Stochastic Methods For Physics Using Java:
An Introduction

PD Dr. F. Petruccione and Dr. P. Biechele

Copyright (c) 2000 Francesco Petruccione and Peter Biechele
Permission is granted to copy, distribute and/or modify this document under the terms of
the GNU Free Documentation License, Version 1.1 or any later version published by the
Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being
LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in
the section entitled "GNU Free Documentation License".
All programs contained herein are under the GNU GPL.

Version of the 8th May 2000
# Contents

## Java

0 Introduction to Programming in Java 5
  0.1 Programming in Java 5
  0.1.1 General Considerations 5
  0.1.2 Programming Languages – Why Java? 5
  0.1.3 Java 6
  0.1.4 Brief History of Java 8
  0.2 Tools for Writing and Using Java 9
    0.2.1 Programs: 9
    0.2.2 Java Packages: 10
  0.3 Basic Elements of Java 11
    0.3.1 The “HelloWorld” Program – Applications and Applets 12
    0.3.2 Variables 18
    0.3.3 Casting and Type Conversions (Wrapper Classes) 19
  0.4 Packages and Import Statements 20
    0.4.1 Packages 20
    0.4.2 The jar Tool 21
    0.4.3 Basic Java Organization 21
    0.4.4 Import statement 22
    0.4.5 Compiling Projects 23
  0.5 Simple Arithmetics, Conditional Statements and Loops 23
    0.5.1 Simple arithmetics 23
    0.5.2 Loops 24
    0.5.3 Conditional Statements 26
  0.6 Arrays, Matrices and Strings 28
    0.6.1 Arrays in Java 2 30
  0.7 Parameters from the Command Line or a HTML File 30
    0.7.1 Parameters from the command line 30
    0.7.2 Parameters from a HTML file 31

1 Object Oriented Programming 33
  1.1 A Classical Example: The Buffon Needle 33
  1.2 The Traditional (Procedural) Approach 34
  1.3 The Object Oriented Approach - Classes and Objects 36
    1.3.1 Definition of Objects 37
    1.3.2 The Code of the Object Oriented Approach 38
    1.3.3 Class variables, Constants and Modifiers 39
    1.3.4 The Constructor 40
    1.3.5 Methods and Class Methods 41
  1.4 Another Example: Calculating the Mean 42
    1.4.1 “Program in One File” Approach 43
    1.4.2 “Traditional Procedural” Approach 43
## CONTENTS

1.4.3 Object Oriented Approach ............................................. 44
1.5 Interfaces and Abstract Classes ........................................ 46
1.6 Extending (Inheritance) and Overloading (Overriding) Classes ........ 46
1.7 The System Class: Screen-Output and Keyboard-Input .................. 47
  1.7.1 Easy Input and Java Rocks printf() ............................ 48
1.8 Passing Arguments to Methods ......................................... 50
1.9 Structure and Overview of Java ........................................ 52
  1.9.1 Packages in Java 1.1 and Java 2 ................................. 52
  1.9.2 Reserved words in Java ........................................... 54
1.10 Name Conventions in Java ............................................. 54
1.11 Java Documentation .................................................. 55
1.12 Applications and Applets Revisited ................................... 55
  1.12.1 Applications ..................................................... 55
  1.12.2 Applet .......................................................... 56
  1.12.3 Programs as Applets and Applications .......................... 57
1.13 Higher Mathematics in Java .......................................... 59
  1.13.1 Standard Mathematical Functions in Java ....................... 59
  1.13.2 Numerical Libraries - The JNL .................................. 61
  1.13.3 The JSci/JavaSci Package ....................................... 63
  1.13.4 JNT, Lapack for Java - JamPack and Jama ....................... 66
1.14 Debugging in Java .................................................... 66
1.15 Advanced Java Features .............................................. 66
1.16 Online References .................................................... 67
1.17 Exercises ............................................................ 68

2 Plotting with Java  ................................................................ 71
  2.1 The Radioactive Decay ................................................. 71
    2.1.1 Random Numbers in Java ....................................... 72
    2.1.2 The Simulation Code ............................................. 73
  2.2 The Most Easy Plot – The AWT and Applet Packages ................. 74
  2.3 Ptplot – Extending Java’s Graphics Capabilities ....................... 83
  2.4 Plot Methods in the Simulation Package ............................... 86
  2.5 Printing in Java and with Ptplot ..................................... 95
  2.6 Advanced topics ........................................................ 97
    2.6.1 3D plots in Java – Java3D ....................................... 97
    2.6.2 Using (system dependent) external programs like gnuplot ...... 99
  2.7 Exercises ............................................................ 101

II Introduction to Stochastic Variables .................................... 103

3 Stochastic Variables ....................................................... 105
  3.1 The Nature of Probabilities ............................................. 105
    3.1.1 The Axiomatic Interpretation ................................... 105
    3.1.2 The Relative Frequency Interpretation ......................... 105
    3.1.3 The Ensemble Interpretation .................................... 107
  3.2 The Definition of Stochastic Variables ................................ 108
    3.2.1 Further Characterization of Stochastic Variables .......... 109
    3.2.2 Some Important Random Variables ........................... 109
    3.2.3 Multivariate Random Variables ................................. 112
  3.3 The Random Variables Transformation Theorem ...................... 113
    3.3.1 The Addition of Stochastic Variables ......................... 114
    3.3.2 One-to-One Transformations .................................... 114
    3.3.3 The Central Limit Theorem ..................................... 115
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.4</td>
<td>The $\chi^2$–Distribution</td>
<td>117</td>
</tr>
<tr>
<td>3.4</td>
<td>Examples</td>
<td>117</td>
</tr>
<tr>
<td>3.4.1</td>
<td>File Input/Output in Java</td>
<td>117</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Exception Handling in Java</td>
<td>126</td>
</tr>
<tr>
<td>3.4.3</td>
<td>The Discrete–Time Random Walk</td>
<td>127</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Generation of Gaussian Random Numbers</td>
<td>131</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Estimation</td>
<td>131</td>
</tr>
<tr>
<td>3.5</td>
<td>Beyond this Chapter</td>
<td>134</td>
</tr>
<tr>
<td>3.6</td>
<td>Exercises</td>
<td>134</td>
</tr>
<tr>
<td>4</td>
<td>Data Analysis</td>
<td>139</td>
</tr>
<tr>
<td>4.1</td>
<td>Estimation</td>
<td>139</td>
</tr>
<tr>
<td>4.2</td>
<td>Simple Monte Carlo Evaluation of Integrals</td>
<td>139</td>
</tr>
<tr>
<td>4.3</td>
<td>Beyond this chapter</td>
<td>143</td>
</tr>
<tr>
<td>4.4</td>
<td>Exercises</td>
<td>143</td>
</tr>
<tr>
<td>5</td>
<td>Sampling of Probability Distributions</td>
<td>147</td>
</tr>
<tr>
<td>5.0.1</td>
<td>A Random Number Generator</td>
<td>147</td>
</tr>
<tr>
<td>5.1</td>
<td>The Generation of Uniformly Distributed Random Numbers</td>
<td>149</td>
</tr>
<tr>
<td>5.2</td>
<td>The Transformation Method: Invertible Distributions</td>
<td>154</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Exponential Distribution</td>
<td>154</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Gaussian Distributed Random Numbers</td>
<td>155</td>
</tr>
<tr>
<td>5.3</td>
<td>The Acceptance–Rejection Technique</td>
<td>157</td>
</tr>
<tr>
<td>5.4</td>
<td>Variance Reduction: Importance Sampling</td>
<td>159</td>
</tr>
<tr>
<td>5.5</td>
<td>Sampling of Polymer Configurations</td>
<td>162</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Ideal Chains</td>
<td>162</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Real Chains</td>
<td>165</td>
</tr>
<tr>
<td>5.6</td>
<td>Exercises</td>
<td>176</td>
</tr>
<tr>
<td>III</td>
<td>Stochastic Processes</td>
<td>187</td>
</tr>
<tr>
<td>6</td>
<td>Markov Processes and Master Equations</td>
<td>189</td>
</tr>
<tr>
<td>6.1</td>
<td>Stochastic Processes</td>
<td>189</td>
</tr>
<tr>
<td>6.2</td>
<td>Markov Processes</td>
<td>190</td>
</tr>
<tr>
<td>6.3</td>
<td>The Differential Chapman–Kolmogorov Equation</td>
<td>191</td>
</tr>
<tr>
<td>6.3.1</td>
<td>The Generator of a Markov Process</td>
<td>191</td>
</tr>
<tr>
<td>6.3.2</td>
<td>The Differential Chapman–Kolmogorov Equation</td>
<td>192</td>
</tr>
<tr>
<td>6.4</td>
<td>The Liouville Equation</td>
<td>193</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Example: Classical Statistical Mechanics</td>
<td>195</td>
</tr>
<tr>
<td>6.5</td>
<td>The Master Equation</td>
<td>196</td>
</tr>
<tr>
<td>6.6</td>
<td>Stochastic Simulation</td>
<td>198</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Radioactive Decay</td>
<td>202</td>
</tr>
<tr>
<td>6.6.2</td>
<td>The Poisson Process</td>
<td>203</td>
</tr>
<tr>
<td>6.6.3</td>
<td>The Continuous Time Random Walk</td>
<td>206</td>
</tr>
<tr>
<td>6.7</td>
<td>The Fokker–Planck Equation</td>
<td>208</td>
</tr>
<tr>
<td>6.7.1</td>
<td>The Wiener Process</td>
<td>210</td>
</tr>
<tr>
<td>6.7.2</td>
<td>The Ornstein–Uhlenbeck Process</td>
<td>216</td>
</tr>
<tr>
<td>6.8</td>
<td>Lévy or Stable Distributions</td>
<td>218</td>
</tr>
<tr>
<td>6.8.1</td>
<td>The Cauchy Process</td>
<td>222</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Lévy Processes</td>
<td>225</td>
</tr>
<tr>
<td>6.8.3</td>
<td>The Numerical Generation of Levy Distributed Random Variables</td>
<td>228</td>
</tr>
<tr>
<td>6.9</td>
<td>Fractal Space Processes</td>
<td>228</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Levy Flights</td>
<td>228</td>
</tr>
</tbody>
</table>
A Summary of Java

A.1 Basic Syntax .................................................. 375
A.2 Structure of a Java program .................................. 375
A.3 The java.lang.System class .................................. 376
A.4 Mathematics .................................................. 377
  A.4.1 The java.lang.Math class ................................ 377
  A.4.2 JNL .......................................................... 377
  A.4.3 JavaSci ...................................................... 377
  A.4.4 Others ...................................................... 377
A.5 Random Numbers .............................................. 377
A.6 Keyboard input and Screen Output ............................ 377
A.7 File I/O .......................................................... 377
A.8 Jplot ............................................................ 377
A.9 AWT ............................................................... 377
A.10 Conversions and Casting ...................................... 377
A.11 Threads .......................................................... 377
A.12 Printing .......................................................... 377
A.13 Modifiers ........................................................ 377
A.14 Debugger ........................................................ 377
A.15 JDE and Emacs ................................................ 377

B Listings and Tables ............................................. 379
B.1 Listing of the ShowTrace.java Program ...................... 379

C Listings for the Exercises ...................................... 381
C.1 Listings for Chapter 1 ............................................ 381
  C.1.1 Calculation of \( \pi \), Exercise 1.1 ....................... 381
  C.1.2 Photoabsorption, Exercise 4.1 ......................... 384
  C.1.3 Monte-Carlo-Integration ................................... 384
  C.1.4 Euler Constant ............................................. 388
  C.1.5 The Standard Deviation ................................... 389
C.2 Listings for Chapter 2 ........................................... 390
  C.2.1 Random number generator check ......................... 390
  C.2.2 Galton Board ............................................... 391
  C.2.3 Poisson Distribution ....................................... 392
C.3 Listings for Chapter 3 .......................................... 393
  C.3.1 Random number generator ................................. 393
  C.3.2 Acceptance-rejection method .............................. 394
  C.3.3 Importance Sampling ........................................ 395
  C.3.4 First passage times ........................................ 396
  C.3.5 Scaling Behaviour of Random Walk in 2D and 3D ........ 397
  C.3.6 Percolation in 2D ........................................... 399
  C.3.7 First passage times ........................................ 402
C.4 Listings for Chapter 4 .......................................... 403
  C.4.1 One-Step Processes ......................................... 403
  C.4.2 Quantum Harmonic Oscillator ............................. 408
  C.4.3 Growth of competitive population ......................... 408
  C.4.4 Random Telegraph Process ................................ 408
  C.4.5 Monomolecular Chemical Reaction ......................... 408
  C.4.6 The Payroll Process ........................................ 409
C.5 Listings for Chapter 5 .......................................... 409
C.6 additional Listings ............................................. 410
  C.6.1 Random Walk 1D ............................................ 410
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of the Java language execution model.</td>
</tr>
<tr>
<td>2</td>
<td>An HTML file gets loaded into the browser. The browser itself formats the text and if it finds an applet mark, it starts the applet.</td>
</tr>
<tr>
<td>3</td>
<td>The output of the <code>javadoc</code> command from the JDK 1.1.</td>
</tr>
<tr>
<td>4</td>
<td>The output of the <code>javadoc</code> command from the JDK 1.1.</td>
</tr>
<tr>
<td>5</td>
<td>The output of the <code>javadoc</code> command from the JDK 1.2.</td>
</tr>
<tr>
<td>1.1</td>
<td>The Buffon needle problem. Definition of the variables $x$ and $\phi$.</td>
</tr>
<tr>
<td>1.2</td>
<td>The Buffon needle problem. The $x$-$\phi$ plane (schematically).</td>
</tr>
<tr>
<td>1.3</td>
<td>A graphical overview of the access control of variables and objects/classes in Java.</td>
</tr>
<tr>
<td>1.4</td>
<td>The file structure of the JDK distribution, both the binary and the documentation package.</td>
</tr>
<tr>
<td>1.5</td>
<td>The line of execution in an application and an applet in the appletviewer or the Netscape Navigator 4.08. For the application only the part above the first line and below the second line are actually the parts, which can not be avoided. The remaining part is just provided to show you how to write an application, which can be used as an applet, too.</td>
</tr>
<tr>
<td>2.1</td>
<td>The peer architecture of the AWT in Java.</td>
</tr>
<tr>
<td>2.2</td>
<td>This figure shows the inheritance hierarchy of the Applet class.</td>
</tr>
<tr>
<td>2.3</td>
<td>The hierarchy of the AWT package.</td>
</tr>
<tr>
<td>2.4</td>
<td>The output of the most easy window with a button using the AWT.</td>
</tr>
<tr>
<td>2.5</td>
<td>The default behaviour of the painting methods of components in the AWT package.</td>
</tr>
<tr>
<td>2.6</td>
<td>The output of the easypplot version of the radioactive decay program. The exact solution is a black line, the simulation a red line.</td>
</tr>
<tr>
<td>2.7</td>
<td>The class hierarchy of the Ptplot package.</td>
</tr>
<tr>
<td>2.8</td>
<td>The output of the RadioactiveDecay ptplot.java program.</td>
</tr>
<tr>
<td>2.9</td>
<td>Two realizations of the stochastic process of the radioactive decay. The first one with linear $y$ axis scaling and the second one uses a logarithmic $y$ axis scaling. The blue lines are the exact solution and the red ones are the simulations. The parameters of the simulation were chosen to be $N_0 = 100$; $\mu = \lambda \Delta t = 0.01 \text{s}^{-1}$; $\Delta t = \text{1s}$; $t_{\text{end}} = \text{300s}$.</td>
</tr>
<tr>
<td>2.10</td>
<td>The same as figure 2.4, but with different parameters: $N_0 = 1000$; $\mu = \lambda \Delta t = 0.03 \text{s}^{-1}$; $\Delta t = \text{1s}$; $t_{\text{end}} = \text{100s}$.</td>
</tr>
<tr>
<td>2.11</td>
<td>The distribution of the number of decays using 100 realizations. The simulation was run for $N_0 = 100$ and $\lambda = 0.001$.</td>
</tr>
<tr>
<td>2.12</td>
<td>The distribution of the number of decays using 1000 realizations. The simulation was run for $N_0 = 100$ and $\lambda = 0.001$.</td>
</tr>
<tr>
<td>2.13</td>
<td>An example of the 3D plotting capabilities of JSci.</td>
</tr>
<tr>
<td>2.14</td>
<td>The configuration of the system.</td>
</tr>
<tr>
<td>3.1</td>
<td>Simulation of the evolution of the relative frequency of throwing a 4 in play of die.</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the <code>io</code> package in Java 1.1 and related classes.</td>
</tr>
<tr>
<td>3.3</td>
<td>Connecting two streams together to get advanced functionality.</td>
</tr>
<tr>
<td>3.4</td>
<td>One realization of a one–dimensional random walk.</td>
</tr>
<tr>
<td>3.5</td>
<td>Another realization of a one–dimensional random walk.</td>
</tr>
</tbody>
</table>
3.6 The distribution of the end–points of the one–dimensional random walk. the program wrdn was run for nstep=100 and nreal=1000. ................................. 130
3.7 The distribution of the Gaussian random numbers generate with the help of the program cltgen. The number of random numbers drawn was chosen to be n = 1000. ................................. 132
4.1 The estimation of pi for n=10,100,1000,10000. The error bars correspond to the standard deviation of the mean of the estimate. ................................. 142
4.2 The scoring method. The continuous line represents the function $\sqrt{1-x^2}$. ................................. 143
5.1 Successive values in a series of random numbers generated for $a=5$, $c=3$, $M=8$. Note that the even numbers are always one less then the odd ones! ................................. 150
5.2 Successive values in a series of 3000 random numbers generated for $a = 65539$, $c = 0$, $M = 2^{31} - 1$. ................................. 152
5.3 Correlation between successive values in a series of 3000 random numbers generated for $a = 65539$, $c = 0$, $M = 2^{31} - 1$. ................................. 152
5.4 Correlation between successive values in a series of 3000 random numbers generated for $a = 65539$, $c = 0$, $M = 2^{31} - 1$. ................................. 153
5.5 Correlation between successive values $R(i),R(i+1),R(i+2)$ in a series of 3000 random numbers generated for $a = 65539$, $c = 0$, $M = 2^{31} - 1$. ................................. 153
5.6 Histogram of 1000 exponentially distributed random numbers with mean 1 generated according to the transformation method. The continuous line represents the expected exponential distribution. ................................. 156
5.7 Histogram of 1000 Gaussian distributed random numbers with mean 0 and variance 2 generated according to the Box–Muller method. The continuous line represents the expected Gaussian density. ................................. 157
5.8 Histogram of 5000 random numbers distributed according to $\rho(x) = 3x^2$ generated with the von–Neumann acceptance–rejection technique. The continuous line represents the exact density $\rho(x)$. ................................. 160
5.9 Ten realizations of a two–dimensional random walk on a square lattice. ................................. 165
5.10 Attrition problem: The average number of trials necessary to generate polymers of a given length increases exponentially with the length of the polymers. ................................. 167
5.11 The flow diagram of the program saw2.m. ................................. 170
5.12 Five realizations of a two–dimensional self–avoiding random walk on a square lattice generated by the simple sampling technique. ................................. 174
5.13 The mean square end–to–end distance of a self–avoiding random walk generated by the simple sampling technique as a function of the polymer length. ................................. 175
5.14 The CPU time for generating 10 realizations of a self–avoiding random walk by the simple sampling technique as a function of the polymer length. ................................. 175
5.15 Five realizations of a two–dimensional self–avoiding random walk on a square lattice generated by the importance sampling technique. ................................. 176
5.16 The mean square end–to–end distance of a self–avoiding random walk generated by the importance sampling technique as a function of the polymer length. ................................. 177
5.17 The CPU time for generating 10 realizations of a self–avoiding random walk by the importance sampling technique as a function of the polymer length. ................................. 177
5.18 The mean square end–to–end distance of a self–avoiding random walk estimated from a sample of 500 realizations by the importance sampling technique as a function of the polymer length. ................................. 178
5.19 The CPU time for generating 500 realizations of a self–avoiding random walk by the importance sampling technique as a function of the polymer length. ................................. 179
6.1 Overview of the theor of stochastic processes. ................................. 194
6.2 Flow chart of a stochastic simulation of a one–step process. The symbols used are explained in the text. ................................. 200
6.3 Stochastic simulation of radioactive decay. The initial number of decaying nuclei is n=100. tend is 30 and the ensemble average was taken over 10 realizations. The decay rate is γ=0.1. .............................................. 203
6.4 Stochastic simulation of the Poisson process. The one–sided random walk starts at nstart=0. tend is 30 and nreal=1. The jump rate is q=1. ............................................................. 205
6.5 Stochastic simulation of the Poisson process. The one–sided random walk starts at nstart=0. tend is 30 and nreal=20. The jump rate is q=1. ............................................... 206
6.6 Stochastic simulation of the Poisson process. The one–sided random walk starts at nstart=0. tend is 30 and nreal=20. The jump rate is q=10. ................................. 207
6.7 Stochastic simulation of the continuous time random walk. The random walk starts at nstart=0. tend is 30 and nreal=20. The jump rate is q=1. .................. 208
6.8 Flow diagram of the program wiener.m ................................................................. 213
6.9 One realization of the Wiener process. The parameters used in the simulation are xs–start=0, tend=50, deltat=0.01, and nreal=1. ................................. 214
6.10 One realization of the Wiener process. The parameters used in the simulation are xs–start=0, tend=50, deltat=0.01, and nreal=1. ................................. 215
6.11 One realization of the Wiener process. The parameters used in the simulation are xs–start=0, tend=5, deltat=0.01, and nreal=1000. ......................... 215
6.12 The flow diagram of the simulation of the Ornstein-Uhlenbeck process. ........... 217
6.13 One realization of the Ornstein–Uhlenbeck process. The parameters used in the simulation are xstart=5, tend=10, deltat=0.01, nreal=1, q=1, and D=1. .......... 219
6.14 The average over 10 realizations of the Ornstein–Uhlenbeck process. The parameters used in the simulation are xstart=5, tend=50, deltat=0.01, nreal=10, q=1, and D=1. 219
6.15 Two possible realizations of the Cauchy process. .................................................. 225
6.16 The Weierstrass function G(κ) for M=0,1,2,3,4,5 in different colors. The parameters are a=2,b=3. .............................................................. 230
7.1 Results of the simulation of the Ornstein–Uhlenbeck process with the stochastic Euler method for different values of the time step. The parameters of the Ornstein-Uhlenbeck process are q=1, D=1. The simulation was run from tstart=0 to tend=4 for 50000 realizations. The timesteps used are deltat=0.2, 0.1, 0.05, 0.025. .... 258
7.2 Histogram of the invariant density of the stochastic differential equation with multiolicative noise. The simulation was run from tstart=0 to tend=4 for 50000 realizations. The initial condition was chosen to be xstart=0.5. The timestep used was deltat=0.01 and the multiplicative noise constant was epsilon=1. ....... 262
7.3 Histogram of the invariant density of the stochastic differential equation with multiolicative noise. The simulation was run from tstart=0 to tend=4 for 50000 realizations. The initial condition was chosen to be xstart=0.5. The timestep used was deltat=0.01 and the multiplicative noise constant was epsilon=3. .............. 262
7.4 The potential U(x) = −a x^2/2 + b x^4/4 for a = 1 and b = 1. ................................ 264
8.1 Plot of the Lennard–Jones potential V_{LJ}. The potential is characterised by the length scale σ and by the energy ε. ...................................................... 275
8.2 Plot of the hard sphere potential. .............................................................. 275
8.3 Plot of the square well potential ................................................................. 276
8.4 Plot of the soft sphere potential for ν = 1 and ν = 12. ........................................ 276
8.5 Periodic boundary conditions. ........................................................................... 280
8.6 The minimum image convention. ....................................................................... 281
8.7 The most important event classes in Java and their structure. In these classes you can find the events available in Java. There are many events also outside of the AWT, which are not relevant for us. ................................................ 285
8.8 The most important AWT listeners and their class structure. Look at the API documentation of the listeners to find all the available methods to be overriden, this shows you what kind of actions are possible to detect. ................................. 286
8.9 The available adapter interfaces for the AWT listener. These classes ease the writing of listeners by only overriding the methods you need, you just implement the appropriate event adapter interfaces. .......................................................... 286
8.10 Qualitative behaviour of the pair correlation function \(g(r)\) for a Lennard–Jones fluid. ................ 289
8.11 The pressure ........................................................................................................... 291
8.12 The temperature as a function of time in Molecular Dynamic simulation of a micro-canonical ensemble. .............................................................. 292
8.13 Plot of the velocity autocorrelation function of a Lennard–Jones fluid. ......................... 292
8.14 The velocity field in a plane Couette flow. .............................................................. 296
8.15 Moving periodic images for the simulation of a plane Couette flow. ......................... 296

9.1 The graphical solution of Eq. (9.2). ............................................................................. 306
9.2 Configurations of the two dimensional Ising model on a 100 \(\times\) 100 lattice at \(\beta/\beta_c = 0.5, 0.7, 0.9, 0.95, 0.98\). Notice the growth of correlations from high temperatures to the critical region. ................................................................. 310
9.3 Magnetisation as a function of the reduced temperature \(kT/2J\) for \(L = 40, L = 50\) and \(L = 100\). .......................................................................................................................... 311
9.4 Magnetic susceptibility. .............................................................................................. 311
9.5 The energy as a function of T. .................................................................................... 311
9.6 The specific heat as a function of T. ............................................................................ 312
9.7 Estimates for \(\sigma^2(m)\) obtained with the blocking method. .............................................. 314
9.8 Finite size scaling behaviour of the two dimensional Ising model on \(L \times L\) square lattices. (Exact solutions of Ferdinand and Fisher??). ................................................................. 315
9.9 Helical boundary conditions ....................................................................................... 317
9.10 The stages of the Swendsen–Wang algorithm for a 6 \(\times\) 6 array with helical boundary conditions. IM WESLICHEN DIE FIGUR AUS MACKEOWN, S.375 MIT ANDEREN RANDBEDINGUNGEN. ................................................. 318

10.1 The three levels of description: macroscopic, mesoscopic, and microscopic. ................. 322
10.2 The different networking models and cables used to connect the CPUs in a “Parallel Computer”. The years just represent the “standard” network used for most permanently connected systems. This is by no means a complete overview, but it should give an impression of what can be expected. The speed denoted above the network protocol is the theoretical maximum value. In reality you can be lucky, if you accomplish 1/10th of this value. ................................................... 327
10.3 Possible software models to be used for parallel programming. For a description of the used abbreviations see table 10.3. Difficult/Easy programming refers to the time needed to implement parallel algorithms using a certain model. .............................................. 329
10.4 Amdahl’s law with some examples for the value of the serial part \(f\) of a program. ................ 331
10.5 A multitasking operating system employing a Java Virtual Machine, which runs multiple threads. ............................................................................................................ 337

A.1 The class structure of a Java program, either application or an applet. ............................... 376
List of Tables

1 Performance Table. ................................................. 7
2 Primitive data types in Java. The byte and char data types have been introduced in Java 1.1 and in Java 2 there was a void type added. ........................................ 18
3 Options of the jar command. The jar tool is included in the JDK. ........................................ 22
4 A comparison of the different memory allocation commands in different languages. ........ 28

1.1 Overview of some available modifiers in Java, see also Figure 1.3. For a complete overview take a look at page 230-234 in Flanagan [1997]. ........................................... 40
1.2 All methods belonging to the (abstract) java.lang.Object class. .................................. 47
1.3 All possible modifiers to be used in the format string given to the printf()/sprintf()/fprintf() methods supplied by the Lava Rocks package. ........................................ 49
1.4 Overview of the mathematical methods available in Java 1.1 in the java.lang.Math class. . 60
1.5 A short list of JNL classes supplied with JNL 1.0 revision f for the new JDK 1.2. There are two 1.0f versions around, one which works with both Java 1.1 and Java 2 and one which has some trouble with Java 2. ........................................... 62
1.6 An overview of the most important methods supplied by the Complex class of the JNL. z represents a complex number (Complex z). .............. 63
1.7 Some of the interesting classes and methods in the JSci/JavaSci package. .................. 65

2.1 A list of most of the defined AWT components of the Java 1.1 API and the Java2 API (Swing). In Java2/Swing all the AWT Components are also available with Swing by just adding a capital J to the beginning of the components (e.g. JButton instead of Button, JComponent instead of Component). ........................................... 79
2.2 All possible layouts in the AWT package of Java 1.1. .................................................... 79
2.3 List of some of the methods contained in the Graphics class. All method arguments are of type integer, unless otherwise stated. More methods are available in the much more powerful Java2D API coming with Java2. ........................................ 80
2.4 Overview of all the Ptplot methods in the Plot and PlotBox classes. .......................... 88

3.1 Some of the important exceptions in the java.lang.Throwable class. A detailed list is in the API documentation. RTE means RuntimeException and therefore do not have to be catched. ........................................... 127

5.1 Series of random numbers for the linear congruential generator of the form \( I_{n+1} = (5I_n + c) \mod 8 \) ........................................... 151
5.2 Monte–Carlo estimates of the integral (5.8) using the standard method \( p(x) = 1 \) and the importance sampling method \( p(x) = a \exp(-x) \) ........................................... 161
5.3 Values of \( \langle R^2 \rangle \) as functions of \( N \), for two–dimensional random walks generated by the simple sampling (ss) and by the importance sampling (is) technique. .................. 176
5.4 Mean square end–to–end distance estimated by importance sampling from a sample of 500 realizations. ........................................... 178

7.1 Multiplication table for products of stochastic differentials. ................................. 245
7.2 Results of the simulation of the Ornstein–Uhlenbeck process with the stochastic Euler method for different values of the time step. The parameters of the Ornstein-Uhlenbeck process are $q=1, D=1$. The simulation was run from $t_{\text{start}}=0$ to $t_{\text{end}}=4$ for 50000 realizations. The timesteps used are $\Delta t=0.2, 0.1, 0.05, 0.025$. ........ 258

8.1 The system of units used in molecular dynamics simulation of particles interacting via a Lennard–Jones potential. The numerical values for $\sigma$, $\varepsilon$ and $m$ are for argon. The quantity $k$ is Boltzmann’s constant and has the value $k = 1.38 \times 10^{-23} J/K$. The unit of pressure is for a two–dimensional system. ....................... 282

8.2 A list of important properties to be read out by a Java program. ....................... 283

9.1 Comparison of the critical exponents for the 2 and 3 dimensional Ising model with mean field theory. ................................................................. 308

10.1 Abbreviations used in the area of networking models. ......................... 327

10.2 Abbreviations used for describing computer hardware. ......................... 328

10.3 Abbreviations used for the models in parallel programming. ..................... 329
Listings

Listings Java/HelloWorld_Application.java .................................................. 12
Listings Java/HelloWorld_Applet.java .......................................................... 14
Listings Java/call_HelloWorld_Applet.html .................................................. 14
Listings Java/HelloWorld.java .................................................................. 22
Listings Java/DataMean.java .................................................................. 24
Listings Java/DiceGame.java ................................................................. 27
Listings Java/DataMeanArray.java ......................................................... 29
Listings Java/ParamCommandLine.java ....................................................... 31
Listings Java/ParamApplet.html ............................................................ 31
Listings Java/ParamApplet.java .............................................................. 32
Listings Java/BufferProcedural.java ....................................................... 34
Listings Java/Data.java ........................................................................ 37
Listings Java/Needle.java ...................................................................... 38
Listings Java/TestFinal.java ................................................................. 40
Listings Java/Buffon.java ..................................................................... 41
Listings Java/Moments_all.java ............................................................ 43
Listings Java/Moments_procedural.java .............................................. 44
Listings Java/Moments_procedural.java .............................................. 44
Listings Java/MomentsData.java ........................................................... 45
Listings Java/Moments_object.java ........................................................ 47
Listings Java/System.Class.java .............................................................. 47
Listings Java/testArray.java ................................................................. 51
Listings Java/TestPassingFunctions.java .................................................. 52
Listings Java/Test_Applet.java .............................................................. 56
Listings Java/TestAppletApplication.java ............................................. 58
Listings Java/Test_Roundings.java .......................................................... 59
Listings Java/Test_Roundings.output ....................................................... 60
Listings Java/RadioactiveDecay.java ....................................................... 73
Listings Java/PlotEasy.java ................................................................. 74
Listings Java/RadioactiveDecay_easyplot.java ...................................... 81
Listings Java/Ptplot_Demo1.java ............................................................ 83
Listings Java/Ptplot_Demo2.java ............................................................ 84
Listings Java/RadioactiveDecay_ptplot.java ............................................ 85
Listings Java/RadioactiveDecay_ptplot2.java ............................................ 90
Listings Java/RadioactiveDecay_ptplot2.java ............................................ 92
Listings Java/RadioactiveDecay_printing.java ........................................ 96
Listings Java/Sci3DGraph.java ............................................................... 97
Listings Java/Gnuplot.java ................................................................. 99
Listings Java/Gnuplot.gnu ............................................................... 100
Listings Java/DirectoryListing.java ........................................................ 100
/Listsings/relfreq.m .......................................................................... 106
Listings Java/FileWriteSimple.java ....................................................... 117
Listings Java/FileReadSimple.java .......................................................... 119

XV
Listings/Java/StringBufferDemo.java ........................................... 120
Listings/Java/FileSaveFormatted.java ........................................ 121
Listings/Java/FileBinary.java .................................................. 122
Listings/Java/FileCheck.java .................................................... 123
Listings/Java/RedirectStandard.java .......................................... 124
Listings/Java/GZIPSarray.java .................................................. 125
/Listings/rwd.m .................................................................. 128
/Listings/rwdtn.m ................................................................. 130
/Listings/cltgen.m ................................................................. 131
/Listings/mcp1.m ................................................................. 141
/Listings/mcpiscore.m .......................................................... 142
Listings/Java/RandomNumber.java .............................................. 147
Listings/Java/UseRandomNumber.java ........................................ 148
/Listings/trandom1.m ............................................................. 149
/Listings/random1.m ............................................................. 151
/Listings/expdistr.m ............................................................. 155
/Listings/gaussdistr.m ......................................................... 156
/Listings/neumann.m ............................................................ 159
/Listings/mciis.m ............................................................... 160
/Listings/rw2d.m ................................................................. 163
/Listings/rw2dsa.m ............................................................... 166
/Listings/rw2dsa2.m .............................................................. 170
/Listings/onestep.m .............................................................. 199
/Listings/decaymaster.m ......................................................... 203
/Listings/poissonmaster.m ...................................................... 205
/Listings/walkmaster.m ........................................................ 207
/Listings/wiener.m .............................................................. 212
/Listings/ornstein.m ............................................................. 217
Listings/Java/CauchyProcess.java .............................................. 223
Listings/Java/SDE.java .......................................................... 250
Listings/Java/OrnsteinUhlenbeck.java ....................................... 256
Listings/Java/NoiseInducedTransition.java ............................... 260
Listings/Java/ScrollPaneDemo.java ........................................... 282
Listings/Java/ClosableFrame.java ............................................. 284
Listings/Java/ButtonListenerTest.java ........................................ 287
Listings/Java/DogFlea.java ...................................................... 332
Listings/Java/ConvertSymphony.java ......................................... 334
Listings/Java/DogFleaCalc.java ................................................. 338
Listings/Java/DogFleaThreads.java ............................................. 339
Listings/Java/ShowTrace.java .................................................. 379
Listings/Java/Pi_CalcPlain.java ................................................. 381
Listings/Java/Pi_Calc.java ..................................................... 382
Listings/photoabsorption.m .................................................... 384
Listings/mcistandard.m ......................................................... 384
Listings/hitandmiss.m .......................................................... 386
Listings/hitandmiss2.m .......................................................... 386
Listings/darts.m ................................................................. 388
Listings/variance.m ............................................................. 389
Listings/momentsrand.m ......................................................... 390
Listings/pokertest.m ............................................................ 390
Listings/galton_board.m ........................................................ 391
Listings/poisson.m ............................................................... 392
Listings/linear_con.m ............................................................ 393
Listings/rejection.m ............................................................ 394
<table>
<thead>
<tr>
<th>Listings/importance.m</th>
<th>395</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listings/first_passage.m</td>
<td>396</td>
</tr>
<tr>
<td>Listings/rw_scaling.m</td>
<td>397</td>
</tr>
<tr>
<td>Listings/percolation.m</td>
<td>399</td>
</tr>
<tr>
<td>Listings/einstein_solid.m</td>
<td>402</td>
</tr>
<tr>
<td>Listings/onestep.m</td>
<td>403</td>
</tr>
<tr>
<td>Listings/onestepfast.m</td>
<td>405</td>
</tr>
<tr>
<td>Listings/qmharmonicmaster.m</td>
<td>408</td>
</tr>
<tr>
<td>Listings/nonlineargrowthmaster.m</td>
<td>408</td>
</tr>
<tr>
<td>Listings/telegraphmaster.m</td>
<td>408</td>
</tr>
<tr>
<td>Listings/reactionmaster.m</td>
<td>408</td>
</tr>
<tr>
<td>Listings/payrollmaster.m</td>
<td>409</td>
</tr>
<tr>
<td>Listings/sdeornstein.m</td>
<td>409</td>
</tr>
<tr>
<td>Listings/rw1d.m</td>
<td>410</td>
</tr>
<tr>
<td>Listings/rw2d.m</td>
<td>411</td>
</tr>
<tr>
<td>Listings/rw2dsa.m</td>
<td>412</td>
</tr>
</tbody>
</table>
Instructions for the use of the CD ROM

Install Toolkit
Part I
Java
Chapter 0

Introduction to Programming in Java

0.1 Programming in Java

0.1.1 General Considerations

This book is about stochastic simulation methods and their applications to physical systems. The material is presented at an introductory level. We do not assume any prior knowledge on probability theory or on the theory of stochastic processes. We assume only the material known from the introductory courses in theoretical physics. The style of the presentation will be as informal as possible and as precise as necessary.

0.1.2 Programming Languages – Why Java?

It is clear that it is not possible to teach simulation methods without performing some numerical experiments in the classroom and that it is impossible for the students to learn stochastic simulation methods without implementing the algorithms. Therefore, the theory and the corresponding algorithms will be presented in a highly interconnected, and, we hope, organic way.

In order to stick to this idea it was necessary to choose a programming language. The obvious criteria for taking such a decision are [1]

1. Efficiency,
2. Understandability,
3. Good graphics,
4. Standard/Portable,
5. Cost,
6. Parallelizability.

The first criterion is of course very important because already simple stochastic algorithms require a considerable amount of CPU time. A secondary aim of the course is to make the student acquainted with a programming language “in action” so they should learn something about a “good programming style” for real-life problems. Visualization of the results obtained allows to understand more easily what is going on physically. So the interface between program and visualization tool/graphical output should be comfortable. Last but not least the availability of the program should be guaranteed. The corresponding compilers should be available at low, in the ideal case at no cost to the students for home exercises. Furthermore, they should be portable on PCs with Windows or Linux operating systems, on Macintosh and on workstations from the UNIX world (IBM AIX, Solaris, SGI Irix, ...). Since almost all universities do have a high-performance parallel computer the language should also allow to demonstrate high-performance parallel algorithms.
Because of our background of convinced Fortran users we considered the following alternatives: our beloved Fortran, MATLAB, a language which is popular in the engineering and applied mathematics communities, and Java, the “Wunderkind” of software developers and of the Internet community. We have not considered C, C++ for the simple reason that we never felt the necessity to learn them.

All the languages considered in some sense satisfy the above criteria. All are more or less portable on different platforms (at different expenses), all allow the use of good visualization tools (at different prices), and, of course, all are clean. But not all are equally powerful.

We checked the power (the efficiency) of the languages considered by the following benchmark, which represent the prototype of a stochastic simulation. We generated 100 trajectories of a typical one-step stochastic process (see Chapter ?? and compared the CPU times obtained by different languages. The result of the benchmarks are summarized in Table 1. The listings of the corresponding programs are presented in the appendix.

Of course, in the above test we have not optimized the algorithm to the different platforms. Nevertheless, the table clearly shows that MATLAB is very slow. Even the compiled version of MATLAB is by a factor of about 100 slower compared to the Fortran code. This is a good reason to disregard MATLAB!

Now we have to decide between Java and Fortran. As can be read off from the Table 1 the criteria of numerical efficiency clearly speaks for Fortran. The argument in favour of Java which compensates the slightly slower performance – Today!, in future this might be different. – is its portability and the free availability of the compiler and of the visualization tools. Java runs on every platform and it is available at no cost. We do not even have to change any line of code to get a faster performance, because we just have to get a faster Just in Time compiler or the new HotSpot technology, which improve performance by factors of 2 to 10 or even more. This will free the programmer from time consuming and difficult optimizaions. There is for example a compiler called High Performance Java by IBM. It generates much faster code on IBM workstation compared to the JDK from SUN and the speed is already comparable to C/C++. For a comparison with this compiler see [?].

Last but not least, we want to mention a further advantage of Java. It seems [?] that there is a great need for Java programmers in various branches of industry today. This need will even grow in future years. So learning Java might be a kind of “life insurance” for students of physics. It will put them in the position to find a good job in the software industry.

0.1.3 Java

In Chapter 0.1.2 we have given some good reasons to choose Java as the programming language for our purposes. Here we want to mention some more technical points, from a computational science point of view, in favour of Java. Some of the points are very technical and are only be understood with knowledge about programming. So for beginners it might be useful to come back again here after learning Java in the next chapters.

SUN Microsystems has described Java as follows [?):

Java: A simple, object–oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high–performance, multi-thread, and dynamic language.

Let us try to understand roughly what is meant by the above adjectives.

Java is simple in the sense that the number of language constructs has been kept as small as necessary. For ease of migration from other languages some basic language elements resemble C or C++. However, some features of these languages which were rarely used and which have been considered to be unsafe have been omitted. For example, in Java there is no goto statement; instead it has labelled break and continue statements. The preprocessor of C has been eliminated; the program you write is the program that the compiler sees. In Java there are no operator overloading and no multiple inheritance features known from C++. But you can use interfaces to simulate multiple inheritance and argument overloading is also possible. One major simplification is that Java does not have pointers! In Java memory is taken care of automatically, so the programmer is not responsible for the management of memory space. In particular, Java implements an automatic garbage collector.

Java is an object–oriented language and you do not have to think in a procedural–based way, as it is the case in Fortran for example. In order to solve problems in Java we are forced to use the notions of
Table 1  Performance comparison for different languages, operating systems (OS), and platforms. The test program is a one-step stochastic process. We create 100 realizations, $g(n) = 0.4n$, $r(n) = 0.5n$ (see Chapter ??). On Windows 95 the JIT from Symantec is included and automatically used, when executing programs with the java command in the JDK. The TYA JIT for Linux is freely available and easy to install. Usage: with the Java virtual machine of the JDK use `-Djava.compiler=tya` or set the environment variable `JAVA_COMPILER=tya`. To avoid using the JIT use option `-nojit` up to JDK1.1.7 on Windows or for all other platforms set the environment variable `JAVA_COMPILER=none`.

<table>
<thead>
<tr>
<th>Language</th>
<th>OS</th>
<th>Software</th>
<th>Machine</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 90</td>
<td>Linux</td>
<td>Nag f90 Compiler V2.2(260)</td>
<td>Pentium 133</td>
<td>2.1 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Nag f90 Compiler V2.2(260) with -O3</td>
<td>Pentium 133</td>
<td>2.4 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Nag f95 Compiler V1.0(436)</td>
<td>Pentium 133</td>
<td>2.3 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Nag f95 Compiler V1.0(436) with -O4</td>
<td>Pentium 133</td>
<td>2.3 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Pallas f95 Compiler V3.0-3</td>
<td>Pentium 133</td>
<td>1.3 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Pallas f95 Compiler V3.0-3 with -O4</td>
<td>Pentium 133</td>
<td>1.4 sec.</td>
</tr>
<tr>
<td>C</td>
<td>Linux</td>
<td>GCC 2.7.2.3</td>
<td>Pentium 133</td>
<td>2.1 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>GCC 2.7.2.3 with -O3</td>
<td>Pentium 133</td>
<td>2.0 sec.</td>
</tr>
<tr>
<td>C++</td>
<td>Linux</td>
<td>egcs-1.0.3</td>
<td>Pentium 133</td>
<td>2.2 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>egcs-1.0.3 with -O3</td>
<td>Pentium 133</td>
<td>1.8 sec.</td>
</tr>
<tr>
<td>Java</td>
<td>Linux</td>
<td>JDK 1.1.7, no JIT</td>
<td>Pentium 133</td>
<td>16 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>JDK 1.1.7, with JIT TYA (V1.3)</td>
<td>Pentium 133</td>
<td>10 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>JDK 1.1.7, no JIT</td>
<td>DEC Alpha 21164/600</td>
<td>6 sec.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>JDK 1.1.7, with Symantec JIT</td>
<td>Pentium 133</td>
<td>9 sec.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>JDK 1.1.7, no JIT</td>
<td>Pentium 133</td>
<td>19 sec.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>JDK 1.2, with Symantec JIT</td>
<td>Pentium 133</td>
<td>8 sec.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>JDK 1.2, no JIT</td>
<td>Pentium 133</td>
<td>22 sec.</td>
</tr>
<tr>
<td>Matlab</td>
<td>Win95</td>
<td>Matlab 5.1</td>
<td>Pentium 166</td>
<td>330 sec.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>Matcom Compiler V3.0 with Borland C++ 5</td>
<td>Pentium 166</td>
<td>70 sec.</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Matlab 5.2</td>
<td>Pentium 133</td>
<td>224 sec.</td>
</tr>
<tr>
<td>Maple</td>
<td>Linux</td>
<td>Maple V Rel. 4</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>Mathematica</td>
<td>Win95</td>
<td>Mathematica V3.0</td>
<td>Pentium 133</td>
<td>28 Min.</td>
</tr>
<tr>
<td></td>
<td>Win95</td>
<td>with Compiler</td>
<td>Pentium 133</td>
<td>26 Min.</td>
</tr>
</tbody>
</table>
classes and objects. Every object has a class that defines its data and the methods that operate on these data. Classes are hierarchically arranged. A subclass inherits the behaviour of its superclass. A class is the basic unit of compilation and of the execution in Java. All Java programs are classes. Of course you do not have to use the object oriented programming style, you can still stick to the procedural style in Java too.

Java is a distributed language, which simply means that it provides a lot of tools for networking. Java is the programming language of the Internet.

Java is an interpreted language. The Java compiler compiles the Java source code into Java byte-code, which is the machine language for the Java Virtual Machine (JVM). The JVM is an abstract machine which runs on each system that supports Java. Programs written in other languages may also be compiled into Java byte-code.

Java is robust. Java contains a feature, called exception handling, which simplifies the task of error handling and recovery.

Java is secure. Since Java has been designed for distributed applications high security standards have been implemented. For example, direct access to memory is not allowed. Java contains four different levels of security checks and enforcements to prevent the introduction of viruses. In particular there is protection against deleting and modifying files.

Java is architecture neutral and portable. The byte-code format is always the same regardless of the platform on which the Java compiler runs. Furthermore, there are no “implementation defined” behaviours in Java. For example, Java specifies the size of each primitive data type. The integer types byte, short, int, long take 8, 16, 32, 64 bit of memory, respectively. This also avoids the use of any preprocessor available to all other languages, excessively used in C and C++ to catch all platform relevant parameters.

Java is a high–performance language. Usually Java is run using an interpreter, the so-called Java Virtual Machine. It is however possible to run Java with a Just In Time (JIT) compiler, which translates the bytecode to native code before the code gets executed. JIT compiling increases the performance of Java considerably.

Java is multi-threaded. It supports multiple threads of execution which can handle different tasks. Multi-threading increases the interactive performance of Java.

Java is dynamic. Any Java class can be loaded into a running Java interpreter at any time.

Java includes the zlib compression library in the 1.1 language specification. These are the freely available compression libraries used in the well-known gzip compressor. That makes it very easy to write and read compressed data.

0.1.4 Brief History of Java

Java started off in 1991 as a project by James Gosling, which at that time was called Oaks. Its purpose was focussed on the use as operating software (OS) for consumer electronic devices. A small group decided to adapt Oak to web technology and released the first version of Hotjava (a web browser, at that time still called WebRunner) in late 1994. After a presentation given by James Gosling about the byte codes used by Oak in 1995, the new language JAVA was officially announced in April 1995 (Java 1.0), including the first official release of Hotjava. The announcement was the source of a hype, because Java is ideally suited for the heterogenous world of networked computers seen today and the Java philosophy allows for “Write Once, Run Everywhere” including graphical capabilities.

Then in January 1996 the first version of the Java Development Kit (JDK) was released by SUN. Soon the language was licensed by many companies, most notably Netscape, which included a Java Virtual Machine (JVM) into their widely used browser Netscape Navigator. Then in early 1997 SUN released the second version of Java: the new language specification Java 1.1 and the development kit for it, JDK 1.1.

Meanwhile most of the browsers adapted the new Java 1.1 language specification. Also many new APIs (Application Programmers Interfaces) like Java 3D or most important the JFC have appeared, some of them have been officially included in the Java 1.2 specifications (now called Java 2). Together with the JDK 1.2, Java 2 appeared in January 1999. But no browser supports this standard right now, although most of the software available for Java and written in Java seem to be already adapted to the new standard.

1The Java Foundation Classes, which consist of the Swing API and many more components like (an almost complete set of) the Internet Foundation Classes (IFC). You can either use Java 1.1 together with the JFC 1.1 for Swing 1.1 or Java 2, which already includes JFC 1.1 for Swing 1.1 and also Java 2D and some more new APIs.
Another change occurred to the licensing of the JDK: it is now almost Open Source Software, which means you can have the source code and change it, you only have to make sure it still conforms to the Java standard.

At the same time SUN released a new project called JINI, which is a “small set of instructions and interfaces” based on Java to be used to drive and use arbitrary electronic devices in a local network—the actual aim of the Oaks/Java project started in 1991. The idea is that every JINI device reports to a “naming service” and tells it, what services it provides to the network. The server can then tell, what services are available at all to somebody at some place. Therefore the device can be taken to any place in the world and used in any JINI network to which it can connect. You do not even need a JVM in the device, you can use a JVM supplied by a different device (e.g. a computer or browser) available in the network. The whole system is based on Java code and the RMI protocol supplied by Java.

0.2 Tools for Writing and Using Java

Although we are describing many different programs in this chapter, the only necessary tools to work out the programs in this book are the JDK, an editor like Emacs/XEmacs maybe with JDE and a WWW browser like Netscape or Internet Explorer.

0.2.1 Programs:

**JDK** The Java Development Kit, distributed freely by SUN. This kit is available for almost all platforms, e.g. Windows, Solaris, Macintosh, Linux, etc. This is the first package to get, to use Java. It consists of a Java compiler, a virtual machine and a debugger. There are of course many other compilers, JVMs and debuggers available, but this is the program to start with. A disadvantage of the JDK especially for Windows user is, that you have to use the command line to use it. There are no graphical interfaces coming with the package to start, compile or debug Java programs. For Linux there is a separate package available, which contains the threaded versions for the JDK 1.1. This is mostly not included in the standard packages for the Linux distributions.

**MRJ/SDK** Apple develops the Java development kit for the Macintosh and distributes two versions: the MRJ runtime environment (in version 2.0 included in MacOS 8.5 and the new version MRJ 2.1.2 from http://developer.apple.com/java/) and the SDK for Macintosh in version 2.1 as of the time of writing also available from the address above. These are versions supporting Java 1.1.7 and Swing/JFC. The new version MRJ 2.2 is just getting available.

**GuavaC** OLD WWW ADDRESS !! Guavac is a free Java compiler written in C++.

**Jikes** Jikes is a Java compiler developed at IBM fully conforming to the Java specifications. It is free and much faster than most other compilers. Thy byte code produced is only slightly different. It is a nice replacement for the javac compiler of the JDK.

**Kaffe** Kaffe is an open source JVM. It is a replacement for the java JVM of the JDK. In version 1.0 it is already almost fully Java 1.1 compatible and it includes some of the Java 2 features. It runs under Windows and Unix systems. There is also a commercial version of Kaffe sold by transvirtual.

**GCJ** This is the GNU compiler for Java. It can compile java source code files into class files (bytecode) and it can even compile class files or source files directly into executable files on the platform running GCJ. GCJ is actually a front end to the free famous GCC/EGCS compiler suite. To compile class files to object code you also need the libgcj runtime library.

**CJ/GJ** CJ is an extension to the Java language that supports generic types. This can be used to for example add primitive complex types to Java. This has been already done and the project is called CJ. The idea is to translate the Java source code including the generic types like the primitive complex type to pure Java 1.1 or Java 1.2 code and then compile it with any Java compiler available.
Jolt The JOLT - Java Open Language Toolkit - Project. Tries to compose a full freely available Java development kit. Should include kaffe, guavac and more.

Emacs/XEmacs This is an editor available for most platforms.
We basically use this very powerful but sometimes confusing editor to do all our programming, text editing and more. It is also available for Windows, but it is mostly used on UNIX machines. We are using the Emacs/XEmacs editor throughout the book, but there is no restriction to any of the programs or examples, if you use a different editor.

mpEdit A freely available editor written completely in Java and therefore available on all platforms.
It has all necessary features to write Java/C or C++ programs.

JDE JDE (Java Development Environment) is an Emacs/Xemacs extension written in Elisp.
It enables you to write Java code in a shorter time, if you are using the emacs or xemacs editor.

Netscape Navigator/Communicator A web browser like the Internet Explorer, but written by Netscape inc. and the source code is freely available and fully conforming to the Java standard.
We used Netscape and the appletviewer of the JDK to test all the applets in this book. Be aware that not everything is supported by all browsers and sometimes you get different results or the browser crashes, although the program did run with the appletviewer.

Java Workshop A commercial Integrated Development Environment (IDE) for Java by SUN. Free trial versions for universities and educational institutions are available. Unfortunately it produces not very readable code for later editing.

Freebuilder A freely available IDE for Java.
Already in a usable state, although it is officially in alpha stage.

Netbeans A commercial IDE for Java written in Java
This is an object oriented IDE for Java. It is free for academic and personal use, but not for commercial use.

Simplicity for Java A commercial IDE for Java written in Java
A very nice and easy to use IDE for Java. It is very easy to get started with it. It can even write event handling code for your graphical user interfaces (GUI). Free trial versions are available from the homepage for Simplicity.

TOBA A Java to C Compiler for Linux
It converts Java source code to C and compiles the C code to get better performance. It runs on Linux, Solaris and IRIX and supports Java 1.1. It seems to be no longer developed (August 99).

Fortran to Java A Fortran 77 to Java Compiler.
You can convert very easily Fortran 77 programs to Java. This is a part of the “Java Access to Numerical Libraries” project at the University of Tennesee in the US. F2j was already capable of translating the Lapack routines to Java.

0.2.2 Java Packages:
All these packages are used extensively throughout this book and they are recommended for own Java projects. For most of the programs presented in this book, you have to install these packages.

simulation ??? (mehr Werbung) This is the package developed during the writing of this book. It provides some basic features, which might be of interest or can ease writing code. All the missing methods and classes, which are essential for writing code for stochastic simulations have been included in this package. Some source code of freely available software have been included for convenience, adhering to the software licenses of course.
0.3. BASIC ELEMENTS OF JAVA

Ptolemy (Ptplot) A package to produce 2D plots from data.
Ptolemy – version 1.0 from January 1999 includes version 2.0 of Ptplot – is actually a whole set of useful packages, but so far only the plot package, called Ptplot is currently fully functional. We will use it extensively for all plotting in 2D. It has a nice zoom feature and it allows for the most important plot styles needed.

JNL The Java Numerical Library is written by Visual Numerics and distributed freely\(^2\). It is also submitted to the standardizing committee for proof to include it in the next release of the Java language specification to become a standard.
The JNL includes the basics for using complex numbers, it provides some important standard functions (hyperbolic functions, Gamma function, etc.) and it provides the basic operations for statistical data analysis (e.g. mean, variance, linear least square fit, etc.). It also provides some basic functionality for linear algebra, like matrix decomposition, determinant, trace, solving linear systems, etc.

JSci A freely available package, which contains many physics constants, mathematical operations, etc.
It includes methods for fast Fourier transformations, ordinary differential equations, complex numbers and much more. We prefer to use the JNL implementation of the complex number classes, because they will more probably become a standard in a future release of Java.

Swing Swing is the replacement for the AWT. It is included in the Java 2 standard and therefore the JDK 1.2. For Java 1.1 you can get a separate package, which works with the JDK 1.1.

Biss-AWT This package is meant to be a replacement for the AWT and can be considered the free alternative to the Swing library. We are not using it in this book.

0.3 Basic Elements of Java

To write a program we first need an editor to type the source code. Second you need a compiler to translate the Java code to byte-code. And last, in contrast to most traditional languages like C, C++ and Fortran, we need a virtual machine (interpreter, called JVM) to execute the byte-code.

So for every platform, where a virtual machine is available, you can execute the byte-code without any compatibility problems. But the “look and feel” can be different: for example Java buttons in Windows look different from buttons in X11/Motif using a UNIX operating system. But if you use the new Swing components supplied with the Java 2 standard or extend the Java 1.1 standard with the Swing components, you can choose the appearance of the graphical look, which then is the same across all platforms. So if you are using the new Java 2 standard (JDK1.2), you should definitely use the new Swing components. If you still use the AWT components (like we do here in this book), you should keep in mind that the look (not the functionality) of the program can be different on different platforms. A big advantage of the Swing components is that they do not use any native code, which the AWT components do.

The main reason for the wide availability of Java is that SUN Microsystems distributes the Java Development Kit (JDK) freely for a number of important platforms (Windows, Solaris/Linux, other UNIX systems). The JDK consists of a compiler (javac), a debugger (jdb) and a virtual machine (java)\(^3\).

The JDK can be downloaded from the Internet from the JDK Javasoft link page. The latest version, as of the time of writing, is 1.2. Throughout the book, we will use the Java 1.1 language version and the JDK 1.1.7, because the Java 2 standard is not yet implemented in any JVM of the web browsers available.

The only additional thing necessary to have a Java programming environment is a text-editor to write the Java programs. Use your favorite editor, e.g. emacs or xemacs, which are also freely available and have nice Java editing modes.

There is also a freely available IDE (Integrated Development Environment) called FreeBuilder. This is a complete environment to write Java programs comfortably. Since it is the first running under the GNU

\(^2\)But please consult the License Agreement coming with the software.
\(^3\)Actually, there are some more components like the javadoc command to create HTML documentation or the jar tool to create zipped packages of class files belonging together - see Chapter 0.3.1
license, it is free, but still in alpha development stage, but it already comprises a lot of features. Of course, there are many commercial IDEs available, like SUN's Java Workshop or Simplicity for Java.

Opposing to other languages, which use the ASCII character set, Java uses the Unicode character set. Unicode consists of characters represented by 2 bytes instead of 1 byte like in the ASCII set. So you can use all the 38885 different characters available in the Unicode set (Version 2) for writing a Java program. This means you can name your variables using for example Japanese or Greek characters. Right from the beginning, Java is an international language.

Java is also case sensitive and doesn’t need any special characters to mark continuation lines as is necessary in Fortran for example.

Comments are used just like in C and C++. You can use either the /* ... */ syntax (borrowed from C) for multi-line comments or the // syntax (borrowed from C++) for single line comments. Additionally you can use /*...*/ for comments to automatically generate a HTML documentation file for the class defined in the file. We will see in Chapter 0.3.1 how this can be achieved.

There are also certain disadvantages, which should not be forgotten to mention: The learning curve is certainly steeper for the beginner for Java than for Matlab or even Fortran. This is of course due to the full use of the object oriented approach used for all Java programs. This clearly shows up when we will introduce e.g. the file and keyboard input/output capabilities of Java. But in the long term, learning Java definitely pays off.

A second point to note is, that for scientists a big concern are always complex numbers. They are currently not supported as primitive types as in Fortran for example, but there are (standardized) packages, which add complex number support to Java (e.g. JNL). And the CJ compiler, an extension to the Java compiler, can even handle primitive complex types.

0.3.1 The “HelloWorld” Program – Applications and Applets

Application

Now let us start with the traditional “Hello World” program written in Java. Type the following code using your favourite text editor and save the program in a file called HelloWorld_Application.java.

Figure 1: Overview of the Java language execution model.
0.3. BASIC ELEMENTS OF JAVA

```java
/* The Hello World Program */
/* your first program! */

/**
 * This is the Hello World application, displaying
 * an interesting message, if executed. */

class HelloWorld_Application
{
    public static void main(String[] args)
    {
        System.out.println("Hello_World");
    }
}
```

If we use the JDK 1.1 or 1.2 under Windows or Unix, we can execute the above example by typing on the command line:

- Compiler: `javac HelloWorld_Application.java`
  
  produces `HelloWorld_Application.class` in the same directory.

- Byte-Code Executor (JVM): `java HelloWorld_Application`

- Output on screen: `Hello World`!

Let us now try to understand the above code. The program consists of three lines.

In the first line the program declares with the help of the `class` statement a class called `HelloWorld_Application`. The identifier following the `class` statement is the name by which the class will be referenced. Each Java program is a class. The definition of the class is included in the curly brackets between line 6 and line 10.

In the second line the `main` method is introduced. The `main` method is declared `void` because the method does not return a value. The `main` method is executed when you run the class as an application. The only parameter, the argument of the `main` method, is an array of `String` objects, here named `args`.

In the third line the method `println()` of the system class `out` is invoked. This method simply prints a string and terminates with a new–line command. Alternatively you could use the `System.out.print()` method, which does the same, but does not print a newline at the end.

You have just written and executed your first application in Java. Please, do not worry if you do not understand everything. You are not expected to understand everything at this stage.

If you are using Emacs/Xemacs and JDE you could have started Emacs and then just used the “JDE New” method in the “Files” menu. Then you have to type the `System.out.println()` code, which could again be accomplished by using the Generate menu in the JDE menu. To compile use the “compile” command in the “JDE” menu and similar the “run” command in the same menu.

**Applet**

Java offers another possibility to execute programs, the so-called Applets. In contrast to the stand–alone Java application which starts with a `main` method and runs until it is completed, the applet is a kind of sub–program which runs under the control of some other program. Usually, applets are (small) Java programs, which are started by a server program, e.g. a WWW browser like Netscape Navigator, the Internet Explorer, HotJava or the appletviewer.

In the case of an applet, you have a browser loading a HTML (Hyper Text Mark-up Language) file. This HTML file contains mark-ups, which tell the browser what applet to load and where to find the applet (see figure 2). You should note that HTML is not a programming language, but it is a mark-up language to produce a text with additional “commands” for the meaning of the text parts. It does not give any details about the formatting or layout of the text. To start an applet, HTML uses some special marks, which we will discuss in a moment.

The big difference between an applet and an application is that applets are not allowed to do certain things. For example applets do not have access to local file systems, so it is not possible to save data on

---

4You need at least version 4.06 or patches for earlier versions to use all features of Java 1.1
5Seems to run Java 1.1 programs since version 4, but does not conform to the Java standards.
6Included with the JDK.
The file system from an applet. It is also not possible for an applet to issue a print command, the process of printing has to be initiated by the server (browser). And an applet is not allowed to do time consuming tasks like long computations. This has to be kept in mind, when writing applets.

The “Hello World” example written as an applet takes the following form:

```java
/* Applet version of Hello World */

import java.applet.*;
import java.awt.*;

public class HelloWorld_Applet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello World",25,50); // write text in window
    }
}
```

Applets have to be derived from the applet class `java.applet.Applet`, which include the principal facilities of the Abstract Windowing Toolkit (AWT) and the interface with X/Motif on Unix, Windows on PC, and Mac-OS on Macintosh platforms. As you can see there is no main method in an applet. Instead if an applet is started by a server, the init() method is executed first. There is also a start() and a stop() method, which are executed if the applet becomes visible or disappears in the server window (e.g. by scrolling in the Netscape Navigator window).

Many methods are available to set up the display in an assigned applet area. The paint() method appearing in the “Hello World” applet is responsible for the visual part of the applet. It uses a canvas (drawing area) with a size defined by the calling HTML file. In our case this HTML file could look like:

```html
<html>
<head><title>Test</title></head>
<body>
Here we could explain the applet to the user.
<br>
<applet code="HelloWorld_Applet.class"
        width=500 height=300>
This message is displayed, if
your Browser is not capable of running applets!
</applet>
</body>
</html>
```
The code parameter given in the HTML file defines the name of the applet to be executed. Because there is no init() and no start() method in our example, the paint() method is called by the browser or appletviewer. The size of the canvas has to be given explicitly in pixels in the HTML file.

To run the applet you can either type

- appletviewer call_HelloWorld_Applet.html or
- use the following URL in the browser:
  PATH_TO_CLASS_AND_HTML_FILE/call_HelloWorld_Applet.html
e.g. /home/user/java/call_HelloWorld_Applet.html

The question, when to use an application and when to use an applet, is difficult to answer. We have decided to write most of the programs as applications and applets in one program. So you can decide if you run it as an application or start it as an applet. Some features are of course not available from an applet and you are missing some functionality of some programs, if you run them as an applet. Time consuming calculations should definitely be written as an application, small programs can be written as an applet. Although this is not mandatory it should be obeyed by a good Java programmer.

In Java 1.1 a new package has been introduced, which allows for so-called “trusted applets”. These are specially signed libraries, which are signed with a key by a person we trust (using the JDK we have to use javakey for that purpose). Only if the signature is valid and we trust that person, the “trusted applet” has access to the local files system – basically it can do everything an application could do on our machine (see ? page 142). This is already implemented with the appletviewer, on other “servers” it might work or not. In the future this will be extended to a more extensive set of rules for the allowed methods of an applet.

**Documenting Java Programs – javadoc** A last remark concerns the documentation for the programs. We have learned, that we can include documentation comments using /** .... */. These comments are processed by using the javadoc command from the JDK. If you run the javadoc command as:

```
javadoc HelloWorld_Application.java HelloWorld_Applet.java
```

you get some HTML files: packages.html, tree.html, AllNames.html, HelloWorld_Applet.html, HelloWorld_Application.html. All these files describe the written classes. The tree.html file gives a tree showing the relatives of the class. The package.html gives the package structure (not used here). The most important file is the AllNames.html, which is an index of all written programs in this package (for an introduction to packages see later).

You can even include special predefined strings to supply the author and many more informations to the javadoc command, either for a class or a method, whereas the documentations for the method can also be supplied for a class:

```java
/**
   @author Peter Biechele
   @version 1.0
*/

.. class ... {
}

/**
   @see <a class name>
   @see <class name>@<method name>
*/

.. method ... {
```
This of course gives us the opportunity to write our explanations/documentation between the /* ... */ commands using HTML constructs like <hr> for a horizontal rule or others - try it. Because this code gets directly included in the HTML files produced by the javadoc command, you can have nicely formatted documentation. But you should avoid using <h1> to <h6> – the heading commands – because javadoc uses them for its own structure.

If you load AllNames.html into a browser, you get the HTML file shown in figure 3. Now you can click on the link HelloWorld_Applet and you get the documentation for that class and analogous for the HelloWorld_Application class (see figure 4). The missing graphics on the right of figure 4 are available in the Java documentation of the JDK and just has to be copied into a subdirectory of the HTML directory called images. Then it looks like the figure on the left.

The javadoc command of the JDK 1.2 produces much nicer pages which look like in figure 5. There are no separate gif pictures needed anymore and the structure of the classes is represented much better.
0.3. BASIC ELEMENTS OF JAVA

public class HelloWorld_Applet
extends java.applet.Applet
public class HelloWorld_Applet
extends java.awt.Panel
|---java.awt.Container
|---java.awt.Component
|---java.lang.Object

Constructor Summary
HelloWorld_Applet()

Method Summary
addNotify
Methods inherited from class java.awt.Panel
setStub, showStatus, start, stop
getParameterInfo, init, isActive, newAudioClip, play, play, resize, resize,
getCodeBase, getDocumentBase, getImage, getImage, getLocale, getParameter,
destroy, getAppletContext, getAppletInfo, getAudioClip, getAudioClip,
Methods inherited from class java.applet.Applet
void

Method Detail
HelloWorld_Applet()

Figure 5: The output of the javadoc command from the JDK 1.2.
0.3.2 Variables

Essentially, Java distinguishes between two types of variables, primitive data types and reference data types.

Primitive data types

We already mentioned that in Java each variable or expression has a definite type and that each type has identical size and behaviour on all Java implementations. Java has built in primitive data types to support integer, floating-point, boolean, and character values. The primitive data types of Java for integers, floating-points, characters and boolean variables are listed in Table 0.3.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Contains</th>
<th>Default</th>
<th>Size</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>signed integer</td>
<td>0</td>
<td>8bits</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>signed integer</td>
<td>0</td>
<td>16bit</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>32bits</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>signed integer</td>
<td>0</td>
<td>64bits</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>floating-point</td>
<td>0.0</td>
<td>32bits</td>
<td>±1.40239846E-45</td>
<td>±3.40282347E+38</td>
</tr>
<tr>
<td>double</td>
<td>floating-point</td>
<td>0.0</td>
<td>64bits</td>
<td>±4.94065645841246544E-324</td>
<td>±1.79769313486231570E+308</td>
</tr>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>false</td>
<td>1 bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>Unicode character</td>
<td>\u0000</td>
<td>16bits</td>
<td>\u0000</td>
<td>\uFFFF</td>
</tr>
</tbody>
</table>

The following comments have to be made. In a Java program every variable must have a type that precedes its name when the variable is declared. For example the integer i may be declared as

```java
int i;
```

and integer or long values can be expressed as

```java
int i = 123;
long l = 1234567889L;  long 12 = 1234567891;
```

Characters are represented by two different data types in Java. One can store only one character and is called char. The other one stores many characters like a word or even a sentence. This second datatype is not a primitive datatype but an object and is called string. It will be discussed later on.

Char values are defined in Java between single quotes, e.g.

```java
char c = ‘C’;
```

Usually characters on a computer are represented internally by a number. This number is between 0 and 255 and the coding of the characters to the numbers is called ASCII\(^7\) code. Because not all characters used for all the different languages in the world are representable by 256 different codes, the codespace has been

\(^7\)American Standard Code for Information Interchange
extended to 65535 numbers and is now called Unicode character set. With this set of numbers corresponding to characters you can code every language of the European countries. There are other Unicode sets to represent more complicated languages like Japanese or Chinese.

A Unicode character in Java is represented by the Unicode escape sequence \uxxxx, where xxxx is a sequence of four hexadecimal digits.

Float and double types have special values that may be the result of certain floating-point operations. For example in the java.lang.Float and java.lang.Double classes the special values POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN (not-a-number) are defined.

Floating point numbers are expressed as e.g.

13. 1.3e1 .13E2

and are considered to be constants of type double unless they are specified with f or F, which makes them then float constants.

In Java strings are not of primitive type and they are not an array of chars like in C. Java provides a String class to deal with sequences of character data. The java string class provides methods to operate on String objects.

All variables in Java are initialized automatically, as soon as they are declared. All primitive types get initialized to zero, the boolean type to false and all objects (remember they are references) are initialized to null. So there is no ambiguity like in other languages, if a variable has a defined value at certain points of the program. Although Java forces you to include sometimes a statement to initialize variables, just to make easier to read source code. We will see an example for this later on, when we discuss loops and calculate averages of sequences of numbers.

Reference data types

All non-primitive data types in Java are objects. They are called also “reference types” because they are handled by reference. For example you may pass the address of an object, which is stored in a variable, to a method. In contrast, primitive types are always passed by value.

Strings and Arrays

There are two special data types belonging to the reference data types: the arrays and the strings. Arrays are objects, which have some special properties and are handled a little bit different from ordinary objects, just because they are used very often (see section 0.6).

Strings are another special data type belonging to the reference data type. In contrast to C and C++, they are not arrays of char variables, but separate objects, which can only be accessed as the whole string or by using special string functions. The strings are also not terminated by a \0 like in C/C++, they just contain the text string assigned to the string object. In the section about arrays, we will learn how to handle Strings and in the section about objects, we will finally hear the full story. Actually we hav met strings in our first program already, because the text between two " is a string.

0.3.3 Casting and Type Conversions (Wrapper Classes)

Java uses a clear strategy to convert the primitive data types to other primitive ones. In a calculation Java always converts (casts) the less precise type to a more precise one. So if you multiply an integer and a float value, it converts automatically the integer to a float and then multiplies both values. If you want to convert a value explicitly you can use the cast operators (like in C and C++). Just write the primitive type in round brackets in front of the value to be converted, e.g. result=(double)a*b casts a to a double value and then multiplies it with b, assigning the result to result. You can also use the wrapper classes to be discussed below, but it is much more complex and should be avoided.

A common problem is to convert from strings to primitive types and vice versa. Because strings are objects and not primitive types, we need a method for the conversion. For that reason Java has so called wrapper classes, subclasses of the java.lang.Number class for all primitive data types. These classes provide all the necessary methods for all types of conversion.
The static .valueOf(String s) method always converts a string to the corresponding wrapper class of a primitive type, e.g.

```java
double a;
String text = "+1.234";

Double D = Double.valueOf(text);
a = D.doubleValue();
```

The fourth line converts the string to the Double class and in the fifth line the doubleValue() method converts the Double wrapper class to a double primitive type. The last line shows how to do it in one line. For the other types you just have to substitute float or int for double, i.e., floatValue(), intValue() and the corresponding wrapper class (Integer for Double, etc.).

Another (easier) method, later used in the ParamApplet.java program is, e.g., the Integer.parseInt(String s) method, which gives back a primitive type integer instead of the wrapper class like valueOf(String s). For other conversions you can use the Long.parseLong(String s) or the Long.parseByte(String s) method.

These methods simplify the conversion a little bit. For the example, see ParamApplet.java in line 11. But this method is only available to Integer, Long and Byte wrapper classes, NOT for Float and Double as of Java 1.1. Fortunately they are included in the Java 2 specification. In the Java 1.1 case you have to use the valueOf(String s) and doubleValue(String s), floatValue(String s) methods.

To convert a double value to a string you have to use the toString(double d) method, which is available for all primitive data types. For example, two ways of doing it are:

```java
double d = 3.1234;
String s1 = Double.toString(d);   // most easy way
String s2 = (new Double(d)).toString();
```

**Convenience Classes of the simulation Package for Conversions** The simulation package provides in the util class methods to convert all primitive types to Strings and Strings to primitive types. Therefore we can for example use

```java
double dum = simulation.Util.stringToDouble("23.4567");
String text = simulation.Util.doubleToString(7.4562);
```

to convert a String to a double or a double to a String. There are analogous methods to convert int, long and float variables.

### 0.4 Packages and Import Statements

#### 0.4.1 Packages

Because Java was designed to be able to load code distributed over the whole internet dynamically, you have to avoid name conflicts between the programs/classes. The Java solution for an Internet–wide unique naming scheme is to put every class in a package. A package is a group of related and possibly cooperating classes. The naming scheme should be based on the internet domain name of the organization at which the package is developed.

If we are not using the package command at all, Java uses the empty package. Then we have to put our programs into the current directory. This is not recommended for medium to complex programs, but for test purposes and very small programs this is very convenient.

The name of the package is given at the beginning of a file before the actual program/class definition starts. So for example if we put the statement

```java
package de.freiburg.simulation;
```
at the beginning of the “Hello World” application, we can compile the application with `javac HelloWorld_Application.java` like before. But to run the application we have to use

```
java de.freiburg.simulation.HelloWorld_Application
```

The program does not start? It cannot, because there is one more thing to know. Java is looking for programs/classes in the directory structure given by the package and the class name. So for the example above we have to put the `HelloWorld_Application.class` file in the directory `de/freiburg/simulation/` and issue the `java` command in the directory, where the directory tree starts.

For example on a Unix machine execute:

```
mkdir de
mkdir de/freiburg
mkdir de/freiburg/simulation
cp HelloWorld_Application.class de/freiburg/simulation/
java de.freiburg.simulation.HelloWorld_Application
```

On a Windows machine we have to change the `mkdir` command into the `md` command and `cp` into `copy`.

We can also use an environment variable called `CLASSPATH` to tell the java executor (JVM) where to find the class files. If for example the `CLASSPATH`-variable includes the directory `/home/user/java` you can start the above example in this directory, if the class is in the subdirectory `de/freiburg/simulation/`. It has to be noted that the entries in a `CLASSPATH` specification may also be ZIP files that contain these classes. On Unix systems the directories in a `CLASSPATH` specification are separated by “:” (on Windows systems by “;”).

For example on a Unix system:

```
export CLASSPATH="$CLASSPATH:/home/user/java"
cd /home/user
java HelloWorld_Application
java de.freiburg.simulation.HelloWorld_Application
```

Only one of the last two lines have to be used, depending on the location of the class file. On Windows you have to change the first two lines to

```
set CLASSPATH="$CLASSPATH;c:\java"
cd c:\
```

All the standard API classes of Java are stored in a central jar file, which is additionally packed to save disk space. These classes are always searched, no matter the `CLASSPATH`-variable is set to. This might be different on some systems and the Java API class file path has to be included in the `CLASSPATH`-variable.

0.4.2 The jar Tool

If you have written a lot of small classes, which all work together (called a project), you can put them all inside a “jar” file and give the jar file to friends instead of the whole bunch of small class files. A jar file is just an archive created by the `jar` program coming with the JDK. It works like the well-known UNIX `tar` command. This is actually a very nice method for packaging applets on the internet, because the jar command also compresses its contents.

For example to view the contents of a jar file you can issue the command `jar tvf lava.jar`. The file `lava.jar` is a file on the CD ROM of the Lava Rocks package. You can use any other `.jar` file you have or can find anywhere. Some of the available options used with `jar` are given in table 3. Jar files are portable from one platform to the other, but as you can see jar is not as powerful as the `tar` command in UNIX.

0.4.3 Basic Java Organization

Java is basically defined in two ways: the first one is the basic set of instructions, like all variables, all arithmetics and conditional statements and loops (see in the next few sections). This is the fundamental part of the language.
**Table 3** Options of the jar command. The jar tool is included in the JDK.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>create jar file</td>
</tr>
<tr>
<td>t</td>
<td>table of contents of jar file</td>
</tr>
<tr>
<td>x</td>
<td>extract files from jar archive</td>
</tr>
<tr>
<td>f</td>
<td>name of jar file comes as first argument after options (the default is the standard input/output)</td>
</tr>
<tr>
<td>v</td>
<td>verbose output</td>
</tr>
<tr>
<td>O</td>
<td>do not compress (used for jars residing in the CLASSPATH)</td>
</tr>
</tbody>
</table>

The second one are the Application Programmers Interfaces (API). They are just packages, which consist of many functions and variables to provide a certain functionality. For example the applet API (package) provides all the necessary functions to build and handle applets in Java.

The big difference to other languages like C for example, where also certain (small) APIs exist, is that these APIs are all included in the Java standard, including the ones giving access to graphical capabilities. There is no place for different vendors to supply different libraries (packages) for the same functionality, but using different calling schemes and therefore making programs non portable.

### 0.4.4 Import statement

Before learning more about the syntax of Java, we have to explain another statement appearing in the part of any Java program before the actual definition of the class: the import-statement. With `import` you can make classes available, so you don’t have to use the fully qualified name to the class, which would be very long sometimes.

If you would like to use the HelloWorld class from above in your programs, you have to change the `main()` method to let us say `hello()`.

**Listing of HelloWorld.java**

```java
package de.freiburg.simulation;

public class HelloWorld {
    public static void hello() {
        System.out.println("Hello World");
    }
}
```

Now you could either type `de.freiburg.simulation.HelloWorld.hello()` (the fully qualified name) in your program or you can use

```java
import de.freiburg.simulation.*;

HelloWorld.hello();
```

to import all classes in the `de.freiburg.simulation` class. You can also use:

```java
import de.freiburg.simulation.HelloWorld;

HelloWorld.hello();
```

if you just want to import one special class. But you can not use `import de.freiburg.*;` and then call the method by using `simulation.HelloWorld.hello();`. You can not split the package name in the import statement.

We have already made use of the `import`-statement in the “Hello World” applet. There we have imported the `java.applet`-class, which defines applets and their behaviour, and the `java.awt`-class, which will be explained later.
0.5. SIMPLE ARITHMETICS, CONDITIONAL STATEMENTS AND LOOPS

There is one class, which is always imported without any import statement: the `java.lang.*` classes. This is the fundamental class of Java and it is implicitly imported for all Java programs, so you do not have to specify it. For example the System class is in `java.lang`, that is why we did not have to use an import statement in the “Hello World” application.

0.4.5 Compiling Projects

If you write a program consisting of many classes and files, you may think that this is a lot of work to compile all of them. Or if you are used to writing code in other languages, you might think of using tools for checking if a program has to be recompiled, if you make changes to some files. One of these tools might be the famous “make” utility. But fortunately this is not needed for Java, because the JDK developers (or to be precise the Java standard) already takes care of these problems.

You have to arrange your code in the different classes according to certain rules (packages). So the directory structure is already a nice tree structure of your project.

If you then want to compile the whole project, but only the files which have changed should be recompiled, you issue the command

```
javac -depend -d base_directory main.java
```

then the Java compiler takes care of all “dependencies”.

0.5 Simple Arithmetics, Conditional Statements and Loops

0.5.1 Simple arithmetics

As we already mentioned Java supports almost all of the standard C operators. The arithmetic operators that operate on numerical types are

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` remainder

The `+` operator can also be used to concatenate strings, as we will see later in an example in section 0.7.

It is important to remark, that in Java integer division truncates toward zero (7/2=3, -7/2=-3 and -7/-2=3).

Java has two special operators for increment `++` and decrement `--`. The expression `i++` is equivalent to `i=i+1` except that `i` is evaluated only once. They can be used as pre- and post-operators, depending on the position of the symbols, e.g. `i++` or `++i`. This does not make a difference, if you just have these statements alone, but in some complex expressions this might make a difference. The postfix (prefix) version of the operator `++` evaluates the value of the operand before (after) the increment operation. For example `i = j++` means setting `i` to `j` and then increment `j` by one. But `i = ++j` means setting `i` to `j+1`.

There is no power operator like `**` in FORTRAN or `^` like in many different programs like TeX/LaTeX. In Java like in C/C++ you have to use (like in C) the `Math.pow()` method of the `Math` class. The `^` operator, used sometimes for the power, is the exclusive or (XOR) operation in Java (either in the logical or the boolean sense). We should not confuse this.

Java supports also a standard set of relational and logical operators, which all yield boolean values. They are listed below

---

8Notice that there is no dependence of classes on other classes in the sense of re-compilation. Java does dynamical run-time linking and no static linking at the end of the compilation process as in all other languages.
CHAPTER 0. INTRODUCTION TO PROGRAMMING IN JAVA

> greater than
\geq greater than or equal to
< less than
\leq less than or equal to
== equal to
!= not equal to

The conditional operators

& & conditional AND
| | conditional OR

operate on boolean expressions only.

Java has also bitwise operators which operate on integers and on boolean types. They allow to perform bit manipulation on data.

& and
| or
\sim not
<< shifts bits left filling with zero bits on the right
>> shifts bits right filling with the highest sign bit on the left-hand side (like a division by 2)
>>> shifts bits right filling with zero bits on the left-hand side, treating the argument as a bitfield

Some examples are in place

1 & 0 // is zero
1 | 0 // is one
\sim10 // is -11, do it by hand to check
2 << 1 // is 4, because bitwise is 2=0010 and 4=0100
2 >> 1 // is 1, because bitwise is 2=0010 and 1=0001
2 >>> 1 // is also 1, like above
2 >> 3 // is 0, because bitwise is 2=0010 and 1=0000

You can use these operators for doing fast divisions by powers by 2, because it is much faster to do a bitshift than a division. For example, instead of writing 16/2 use 16 >> 1 or instead of using 8/4 we can write 8 >> 2.

Last not least we have to mention the fundamental assignment operator =. It may be used in combination with other operators, e.g., += means is incremented by.

0.5.2 Loops

for Loops For our forthcoming applications the most important control statement is the for statement. It is used to loop over a range of values from the beginning to the end. Its syntax is

for (init_expressions; boolean_expr; incr_expressions) {
    statements
}

where init_expressions denotes the initial value of the iterated variable, incr_expressions denotes the increment of the iterated variable. For both you can specify more than one expression separated by commas (as in C). This is the only place, where you can use these comma separated lists. The variables used in the for statement can be integer (long) or float (double) values.

At the beginning of the for loop the boolean expression boolean_expr is evaluated. If its value is found to be true the statement is executed repeatedly with increment incr_expr until the value of the boolean expression is found to be false.

As an example demonstrating the use of for loops we want to write a program to calculate the mean of a given number of random numbers. One possible implementation could look like this:
/** compute the mean of N random numbers distributed uniformly */

public class DataMean {
    public static void main(String[] args) {
        int i, N;
        double mean;
        N = 10000; // set the number of random numbers
        mean = 0;
        for (i = 1; i < N; i++) {
            mean += Math.random(); // draw a random number of type double
        }
        mean /= N;
        System.out.println("The mean of "+N+" random numbers
between 0 and 1 is "+mean+"!");
    }
}

Here we used the class java.util.Random which allows for the creation of random numbers. If we don't supply a seed, as is the case here, it just uses the time to initialize the generator. The initialization takes place in line 10, where a new generator is created. You can check this by running the application more than once and comparing the means — they should not be the same.

The nextDouble() method returns a new random number of type double (for a float use rand.nextFloat()). You can also create normally distributed random numbers with the nextGaussian() method of the Random class. The remaining parts of the program should be self-explaning. You can of course use any expression (e.g. d = d*u + 2) in the last part of the for statement, not only the ++ operator, which is obviously used most often.

Byte-Code of a class file Using the DataMean() program, we want to show the byte-code produced by the Java compiler. In the first line you can see the command to use (javap) and below the output:

Compiled from DataMean.java
public synchronized class DataMean extends java.lang.Object
/* ACC_SUPER bit set */
{
    public static void main(java.lang.String[]);
    public DataMean();
}

Method void main(java.lang.String[])
0 sipush 10000
3 istore_3
4 new #10 <Class java.util.Random>
7 dup
8 invokespecial #12 <Method java.util.Random()>  
11 astore_1
12 dconst_0
13 astore_4
15 iconst_1
16 astore_2
17 goto 32
20 dload 4
22 aload_1
23 invokevirtual #17 <Method double nextDouble()>
26 add
27 astore_4
29 iinc 2 1
32 iload_2
33 iload_3
34 if_icmplt 20
37 dload 4
39 iload_3
40 i2d
41 ddiv
CHAPTER 0. INTRODUCTION TO PROGRAMMING IN JAVA

```java
dstore 4
getstatic #18 <Field java.io.PrintStream out>
new #8 <Class java.lang.StringBuffer>
dup
ldc #2 <String " The mean of ">
invokespecial #13 <Method java.lang.StringBuffer(java.lang.String)>
iload_3
invokevirtual #15 <Method java.lang.StringBuffer append(int)>
ldc #4 <String " random numbers">
invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
ldc #3 <String " between 0 and 1 is ">
invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
dload 4
invokevirtual #14 <Method java.lang.StringBuffer append(double)>
ldc #1 <String ". ">
invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
invokevirtual #20 <Method java.lang.String toString()>
invokevirtual #19 <Method void println(java.lang.String)>
return

Method DataMean()
aload_0
invokespecial #11 <Method java.lang.Object()>
return
```

**while and do–while** Java offers also the possibility to use other loop constructs, the while and the do–while loop. There syntax is

```java
while (boolean_expression)
    statement

and

do
    statement
while (boolean_expression)
```

It is important to observe that in the first construct the boolean expression is evaluated before the statement is executed, while in the second construct the boolean expression is evaluated after the statement has been performed!

In order to give an example we write down an equivalent code using the while statement of the for loop of the DataMean.java program. Lines 13 to 15 have to be replaced by

```java
int i;
while(i<N){
    mean += rand.nextDouble();
    i++;
}
```

### 0.5.3 Conditional Statements

**if-else** The if statement is the fundamental form of conditional control of flow. It allows to choose, whether the statements that follow it are executed or not. Its syntax in Java is

```java
if (boolean_expression) {
    statement1
} else if (boolean_expression) {
    statement2
} else {
    statement3
}
```

First, the boolean expression is executed. If the value is true then statement1 is performed, otherwise if there is the optional else statement statement2 is executed. Of course, if-else constructions can be nested, i.e., an if-else conditional control flow, can be placed within another if-else statement.
0.5. SIMPLE ARITHMETICS, CONDITIONAL STATEMENTS AND LOOPS

The conditional operator ? The conditional operator ? provides a single expression yielding one of two alternatives depending on a boolean expression. To demonstrate its use we write down an if/else Java code first and translate it into an equivalent ? construction. The if/else code reads

```java
if (a<b)
    x=1.0;
else
    x=2.0;
```

The equivalent construction with the conditional operator ? is more compact

```java
x= (a<b ? 1.0 : 2.0);
```

The meaning of the different expressions in the above statement should be obvious.

(labeled) break and continue – goto We already remarked that Java does not have a goto instruction to transfer control to an arbitrary statement in a method. To handle with situations where other languages have a goto Java provides the labelled break and continue statements. Labels are typically used in blocks and loops and precede statements

```java
label: statement
```

The break statement is used to exit from a block, e.g. to break out of a loop. E.g., an unlabeled break terminates the innermost for, while or do.

The continue statement is used only within loops. It skips to the end of the loops body and evaluates the boolean expression that controls the loop. The return statement terminates execution of a method and returns to the invoker. If a method returns no value you can use (you can also just omit it)

```java
return;
```

if the method has a return type, the return must include an expression for a returned type. You can use as many return statements as you like, but only one is executed each time the method gets called.

recursive programming As in most other languages, recursive programming is allowed, although it should be avoided. First because of the clarity of the code and second it has low performance and larger memory consumption. Therefore we do not see any reason to show an example, just avoid using recursive algorithms.

switch/case Another central flow structure is the switch statement. It evaluates an integer expression whose value is used to find an appropriate case label among those listed inside the following block. The switch statement may be used to replace nested if-else statements that determine what is the output for each number. The switch statement works only if the value being tested is a primitive integral type and when the value is tested against constant values. the basic syntax of the switch statement is

```java
switch(expression) {
    statements
}
```

After evaluating the expression, the switch statement executes certain code within the block depending on the integral value of the expression. This information is indicated by the integer label following the case: statement. If there is no case: label that matches the value of the expression, the switch command executes the code following default:, if there is one. Otherwise, switch does nothing.

An example of the use of the switch statement is found in the simple program DiceGame.java

```java
/** Application to show the switch-case statement */
public class DiceGame{
    public static void main ( String [] args){
        int face ;
        for ( int i=1; i<7; i++){
```
face = 1 + (int)(Math.random() * 6);
switch (face % 6)
    {
  case 0: // print 6 if the remainder is zero
      System.out.println("Face equals 6");
      break;
  default:
      System.out.println("Face equals 1, 2, 3, 4 or 5");
      break;
    }
}

Again we want to stress that only an integer, long, char or byte type is possible in the switch statement (no double or float). And the case expressions have to be integral values and not any boolean expressions like i>2.

0.6 Arrays, Matrices and Strings

Before discussing the notion of classes and objects, we want to introduce another reference type: the array. Arrays are actually objects (see Chapter 1), but Java provides many special commands for arrays, which makes them a little bit special.

The first question is how to create arrays and how to destroy them. The destruction is easy to explain: it is done automatically by the garbage collector (like for all objects). This is different to other languages like C, C++ and Fortran 90, where you explicitly have to destroy (free) the allocated memory. To create an array you have to use the new keyword used for creating (instantiating) objects. So to create a one dimensional array, called intarray, with 10 elements you use:

    int intarray[] = new int[10];

This also sets all the elements to zero. But this is only true for primitive types. Arrays of reference types (objects) are created the same way, but the elements consist of references to the elements. The elements themselves are NOT initialized and have to be created too. An example for this is a two dimensional array as we will see soon.

Indices of arrays in Java start with zero as in C and not with 1 as in Fortran. No negative indices are allowed in Java. This is the reason for numbering the chapters in this book starting from 0. The length of an array (meaning the number of elements) is always given by the .length field. For the array above you get the number of elements by using intarray.length. We have already met this notation when we discussed the command line parameters.

You can also create the array and initialize it right away by using (also in Java 1.0):

    int intarray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int[] intarray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

or one step further, create an array of objects (here strings) and create the elements in the same step:

    String stringarray[] = {"a","b","c","d","e","f","g"};
In Java you can even put the brackets behind the type instead of the variable name (not possible in C). So it does not matter if you write `int intarray[]` or `int[] intarray;`.

In Java 1.1 you can also use anonymous arrays:

```java
String[] texts;
texts = new String[] {"a","b","c","d","e","f","g"};
System.out.println(new char[] {'h','e','l','l','o'});
```

So you can create and initialize arrays without even using a variable.

Multidimensional arrays are also supported. Just like in C they are arrays of arrays. For example

```java
double matrix[][] = new double[10][10];
```

creates a 10 by 10 matrix, called `matrix`, meaning that you have created 10 arrays of type `double[10]`. An important point to make is that you do not have to specify all dimensions at once. You can for example create a triangular matrix by submitting:

```java
double matrix[][] = new double[10][];
for (int i=0; i<10; i++) {
    matrix[i] = new double[i+1];
}
```

To access multidimensional arrays you can also use one dimensional array syntax (as in C). If you have a two dimensional array you can access the element `[i,j]` by accessing the element `[i+j*columns]`.

Now let's look at an example using arrays. We have rewritten the program `DataMean` above to calculate the average of random numbers by using arrays.

**Listing of DataMeanArray.java**

```java
public class DataMeanArray {
    public static void main(String[] args) {
        int i, N;
        double mean, sum;
        double[] RandomNumber; // declare an array of 1 dimension
        N=10000; // set the number of random numbers
        RandomNumber = new double[N]; // Instantiate the whole array

        // Generate a lot of random numbers and store them in an array
        for (i=0; i<N; i++) {
            RandomNumber[i]=Math.random();
        } // nextDouble() returns the next random number

        // Calculate the sum of all random numbers
        sum=0;
        for (i=0; i<N; i++) {
            sum+=RandomNumber[i];
        }

        // Calculate the mean of the array
        mean=sum/N;

        System.out.println("The mean of "+N+" random numbers
        between 0 and 1 is "+mean+"!");
    }
}
```

Here we first declare a double array called `RandomNumber` and create it in Line 13. Then we store the random numbers in the array and afterwards calculate the mean. The last important point to address is the copying of arrays. You can not just write
int[] array1 = {1,2,3,4,5};
int[] array2;
array2 = array1; // WRONG ! - ERROR !

This would only copy the reference of the array1 object to the array2 object, not the values, the memory address is the same. To copy the values of arrays you have to use the arraycopy() method of the java.lang.System class. So to copy an array in the above example, you have to write:

    System.arraycopy(array1,0,array2,0,array1.length);

This copies all elements of array1 to array2 staring from element 0. The remaining parts of array2 are not created!

0.6.1 Arrays in Java 2

A new class Arrays in the java.util package has been introduced in Java 2, which is of great interest not only to scientific programmers. It includes methods for sorting arrays of arbitrary type using an improved version of the Quick-sort\(^9\) algorithm. So to sort a whole array of double values into ascending numerical order, you just have to use

    /* JAVA 2 */
    import java.util.*;
    double[] array = new double[1000];
    ....
    Arrays.sort(array,0,1000); // to sort the whole array

You can even sort an array of strings or arbitrary objects, although the algorithm used for the object sorting is allowed to vary from one implementation of the virtual machine to another.

Another new functionality is the fill() method. Often you want to assign a value to a whole array of doubles for example. In Java 2 you can do this using

    /* JAVA 2 */
    import java.util.*;
    double[] array = new double[1000];
    Arrays.fill(array,1.0);

which sets the whole array to 1.

Furthermore there is a comparison method for arrays called

Arrays.equals(double[], double[])

for all data types and last but not least there is a binary search algorithm to find a value in an (sorted) array. So for example to find the index of the array element equal to 2.5 you can use the code:

    /* JAVA 2 */
    import java.util.*;
    double[] array = new double[1000];
    ....
    Arrays.sort(array);
    int index = Arrays.binarysearch( array, 2.5 );

0.7 Parameters from the Command Line or a HTML File

0.7.1 Parameters from the command line

The access of parameters given on the command line is as easy as it is in C and C++. The parameters are stored as strings in Java and are given as the parameters to the main() method of the application. That is the reason for the Syntax:

\(^9\)This is a sorting algorithm, which is very versatile and efficient for most datasets. For details see [?]

public void main(String[] args)

It means that the array args contains the parameters. Each parameter is separated with a space in the command line. Here is an example of a program using command line parameters:

Listing of ParamCommandLine.java

```java
public class ParamCommandLine {
    public static void main(String[] args) {
        int N;

        // Number of parameters on the command line
        N = args.length;

        if (N > 0) {
            // Output all parameters each in a line */
            for (int i = 0; i < N; i++) {
                System.out.println("Parameter No. " + i + ": " + args[i]);
            }
        } else {
            // No parameters are given */
            System.out.println("NO parameters specified!");
        }
    }
}
```

So if you run the program as java ParamCommandLine 12 34 abcd t5 the output on the screen will be

Parameter No. 0 : 12
Parameter No. 1 : 34
Parameter No. 2 : abcd
Parameter No. 3 : t5

and if you don’t supply parameters it will be

NO parameters specified!

We also see the concatenation of strings in the output statement. And because you can only supply one argument to the println() method, you have to concatenate all outputs to one long string.

0.7.2 Parameters from a HTML file

In applets there is no command line to supply parameters. So, in order to transmit parameters from the calling HTML file to the Java applet we have to proceed in a different way. In the HTML file you can specify <PARAM> attributes.

Listing of ParamApplet.html

```html
<TITLE> Test </TITLE> <HEAD> <BODY> Text !
</BODY>
<APPLET code="ParamApplet.class" width=400 height=200>
<PARAM name="NumberOfPoints" value="10000">
<PARAM name="DisplayText" value="This is a test parameter!">
Browser not capable of displaying applet !
</APPLET> </BODY> </HTML>
```
In this case we supply two parameters, called `NumberOfPoints` and `DisplayText` to the Java applet. The value is given in the string behind the keyword `value`. The Java applet to this HTML file could look like this:

```java
import java.applet.*;
import java.awt.*;

/* Create an Applet */
public class ParamApplet extends Applet {
    public int NumberPoints;
    public String text_param;

    /* Get the parameters from the HTML file */
    public void init() {
        NumberPoints=Integer.parseInt(this.getParameter("NumberOfPoints"));
        text_param=this.getParameter("DisplayText");
    }

    /* Display the parameters in the window */
    public void paint(Graphics g) {
        g.drawString("Parameter NumberofPoints is "+NumberPoints,20,50);
        g.drawString("Parameter DisplayText is "+text_param,"",20,80);
    }
}
```

In the `init()` method we get the parameter `NumberOfPoints` and convert it to an integer using a wrapper method. The string of the parameter `DisplayText` doesn't have to be converted. Then in the `paint()` method we display the transmitted parameters on the screen. The output in the appletviewer or in Netscape should look like this:

Parameter `NumberOfPoints` is 10000

Parameter `DisplayText` is "This_is_a_test_parameter!"
Chapter 1

Object Oriented Programming and Advanced Java Features

The biggest step you have to take in mastering Java if you are coming from the Fortran or C community, is to switch to the object oriented paradigm. Although C++ programmers are used to objects, there are quite a number of differences to C++ in Java. That is why Java is closer to C than to C++. Since the notions of classes, objects, and methods are quite abstract we want to introduce them with the help of a few examples. First we discuss a classical example from probability theory, the Buffon needle problem using a procedural program and an object oriented version. A second example will be a class for calculating statistical properties of a set of data points stored in a double array.

1.1 A Classical Example: The Buffon Needle

It seems that the earliest documented application of stochastic simulation methods to the solution of an integral has been advanced by Comte de Buffon. The famous Buffon needle problem has been formulated in 1733 but published only in 1777. It is supposed to be the first experiment, a kind of analogue simulation, in the context of geometric probabilities. The problem can be stated in the following way: A needle of length $2l$ is drawn at random onto a horizontal plane ruled with straight parallel lines. The distance between the lines is $2d$. What is the probability $P$ that the needle will intersect one of these lines?

In fact Comte de Buffon performed the experiment of throwing the needles many times to determine the probability $P$. He also carried out the mathematical analysis of the problem which we want to review shortly.

For convenience we denote by $x$ the distance of the middle point of the needle to the nearest line and by $\phi$ ($0 \leq \phi \leq \pi$) the angle between the needle and this line (see Fig. (1.1)). The quantities $x$ and $\phi$ completely determine the position of the needle. It is evident from Fig. (1.1) that the needle crosses the line only if the condition

$$x \leq l \sin \phi$$

(1.1)

is satisfied.

Let us look at the possible positions of the needle in the $x$–$\phi$ plane (see Fig. (1.1)). All positions lying below the $l \sin \phi$ curve between the abscissa 0 and $\pi$ satisfy the condition (1.1). The surface of this region is immediately found by integration, $F = 2l$. The surface $F$ is a measure for the set of all positions of the needle which cross one line. On the other side it is clear that $\pi d$ is a measure for the surface of all possible

---

1 Georges Loui Leclerc Comte de Buffon (* Montbard (Dijon) 7. 9. 1707, †Paris 16. 4. 1788). He was director of the Jardin des Plants in Paris and since 1753 member of the Académie française. His work “Histoire naturelle”, in which theories about the origin of the earth and of its organisms are discussed, was one of the most famous and translated works of the Age of Enlightenment. Influenced by I. Newton he sustained the scientific method based on observation and experiment. His work contains many ideas which entered future scientific theories.
positions of the half needle. The ratio of the two measures $2l/\pi d$ is the probability we were looking for, i.e.,

$$P = \frac{2l}{\pi d} \quad (1.2)$$

Some years later (?) Laplace \(^2\) recognized that the idea behind the Buffon needle experiment could be used to evaluate $\pi$ from the throws of the needles. Today we would call this a Monte Carlo determination of $\pi$. If we repeat the experiment $N$ times and count the number of times the needle crosses a line $M$, the probability $P$ can be estimated by the relative frequency of hits

$$P \approx \frac{M}{N} \quad (1.3)$$

and hence we have with Eq. (1.2)

$$\pi \approx \frac{2lN}{dM} \quad (1.4)$$

Now, let us try to write a Java code for the simulation of the Buffon needle problem.

1.2 The Traditional (Procedural) Approach

The traditional approach is straightforward. You just draw $N$ needles and check for each one if it crosses a line or not. So there are two subroutines. One creates a new needle with all four coordinates. And the second one is a routine, which just compares the lines with the needle coordinates, whether it crosses the lines or not. You then count the number of crossings and you get the final estimate of $\pi$.

\(^2\)Pierre Simon Marquis de Laplace, * Beaumont-en–Auge 28.3.1749 †5.3.1827 Paris. Laplace was one of the leading french mathematicians of his time. Before being a member in the Académie des sciences and a Professor at the École Normale in Paris (1794) he was an examiner at the Ecole militaire in Paris, where in 1785 he examined Napoleon Bonaparte. The most important contributions of Laplace were in celestial mechanics, cosmology, mathematical physics and, probability theory. In his work “Théorie analytique des probabilités” (1812) he develops for the first time a systematic mathematical treatment of probabilistic problems.
1.2. THE TRADITIONAL (PROCEDURAL) APPROACH

Figure 1.2: The Buffon needle problem. The \(x-\phi\) plane (schematically).

```java
/** Procedural Version of the Buffon Needle problem */

public class BuffonProcedural {
    /** some constants of the problem */
    final static int NumberOfRows = 10;
    final static int NeedleLength = 20;
    final static int TableWidth = 200;
    final static int TableHeight = NeedleLength * NumberOfRows;

    /** The main program */
    public static void main(String[] args) {
        int N = 20000; // number of needles to be drawn
        int cross = 0; // number of crossings, which occured
        double[] needlePos = new double[4]; // positions of the needles

        // The loop over all needles to be drawn
        for (int i = 0; i < N; i++) {
            // create a random needle: call method drawNeedle()
            needlePos = drawNeedle();
            // Is the needle crossing a line ???
            // use the method checkNeedle()
            if (checkNeedle(needlePos) == true) cross++;
        }

        // The variable cross contains the number of intersections
        System.out.println("crossings=\n" + cross + ";\nN=\n" + N);
        // The estimate for pi is printed
        System.out.println("Estimated value of pi=
" + 2 * (double)N / cross);
```
private static double[] drawNeedle() {
    // store the position of the needle: (X0,Y0) to (XI,Y1)
    double[] position = new double[4];
    // dummy variables
    double deltaX, deltaY, ysign;

    // create starting position for the needle
    position[0] = TableWidth * Math.random();
    position[1] = TableHeight * Math.random();

    // create the end points of the needle
    deltaX = NeedleLength * Math.sin(2 * Math.PI * Math.random() - Math.PI);
    deltaY = Math.sqrt(NeedleLength * NeedleLength - deltaX * deltaX);
    ysign = (Math.random() < 0.5 ? -1 : 1);

    // store the end points of the needle
    position[2] = position[0] + deltaX;

    // return the position of the start and endpoints of the needle
    return position;
}

private static boolean checkNeedle(double[] needle) {
    // loop over all lines in the plane
    for (int yLine = 0; yLine <= NeedleLength * NumberOfRows; yLine += NeedleLength) {
        // check if the start and endpoints are on opposite sides of the line:
        // 1) if yes return the method with true
    }

    // if no line is crossed by this needle, return false
    return false;
}

First of all we have made use of the Math.random() method, which draws a random number (a double) between 0 and 1. We also used arrays to store the coordinates and even used arrays as parameters to subroutines (methods). The program basically consists of one loop, which first creates a needle and then checks if the needle crosses one of the lines in the plane. The first part is done in the method drawNeedle() and creates at random the positions of the start and endpoint of the needle. These four coordinates are returned and stored in the array needlePos. The method checkNeedle() uses the positions as arguments and checks if one of the lines is crossed by the needle, defined by the supplied coordinates. The variable cross counts the number of crossings, which have occurred so far. After the loop has finished we print the results on the screen.

1.3 The Object Oriented Approach - Classes and Objects

Before we begin to write the object oriented code we have to introduce some formal aspects of the Java language.
1.3. Definition of Objects

We have already met a lot of object oriented features without discussing them in detail. For example, we already noticed that the fundamental unit of programming in Java is the class. A concise definition of classes and objects in Java could be:

A class is a collection of data and methods that operate on that data. In Fortran or C we call the methods procedures or functions. An object is an instance of the class, meaning it is a thing to work with. The class defines the data necessary for the object and the functions which can operate on them.

In other words, like in other languages you can compute only with primitive types (integer, float, ...) but you can also create and manipulate objects. An example already familiar to us, is the array. If you have an array object you can call the method length to get the number of elements of the array.

So what is the difference to the standard function approach here? Instead of using function arguments you supply the argument by putting them in front of the method separated by a point, e.g., args.length. The missing brackets on the length method is not a miss-print, the Java engineers thought it might ease writing array code, but actually it confuses sometimes. Still it is the most easy demonstration of calling a method of a class.

To calculate the mean of an array of doubles, you can either write a method which takes the array as an argument (like you would in Fortran or C) or you can use the object oriented feature:

```java
import VisualNumerics.math.*;

double[] array = new double[100];
for (int i=0; i<100; i++) { // create a random array
    array[i]=Math.random();
}
double result;
Functions
result=Statistics.average(array);

Object Oriented
Data dat = new Data(array);
result = dat.average();
```

The program for the functional approach is contained in the JNL package and can be used directly. But the second approach needs a new class to be defined by ourself. So here is the code to get the second example working. Store this in a separate file in the same directory and compile it before compiling the code above.

```java
public class Data {
    double[] data;
    
    // The constructor : called by : new Data(array);
    Data(double[] array) {
        data = new double[100];
        for (int i=0; i<array.length; i++) {
            data[i]=array[i];
        }
    }

    // The method for computing the mean (average)
    double average () {
        double sum=0;
        for (int i=0; i<data.length; i++) {
            sum = sum + data[i];
        }
        sum = sum / data.length;
        return sum;
    }
}
```
Which version you prefer seems to be a matter of taste. But because all programs in Java are actually classes we have to understand at least the fundamentals of object oriented programming. Later on we will have learned the beauty and clarity of the object oriented approach and the question will turn into: Why do we need the procedural approach?

Just to remind you again, strings are objects and not primitive types. To use strings you first have to declare and instantiate a string object. Here are three possible ways of doing it:

```java
String text;
text = "Test String";
text = new String();
text = new String("Test String");
```

The first line only declares a string object called text and does not allocate (instantiate) the memory for the value of the object, just the reference to the value. The second line instantiates and defines the String object text. The third line instantiates, but does not define it and the fourth line is like the second line, with the second line being more efficient in memory consumption and speed.

An important example of calling methods of classes or objects is the `length()` method of the `String` class. In contrast to the `length` method for the arrays, we have to write the brackets this time. For example the length of a string can be printed by using

```java
String s = "text test";
System.out.println("The length is:"+s.length());
```

And if you want to convert the letters of a string to lower case, you can use the `toLowerCase()` method, e.g.

```java
String s = "TEST Text";
System.out.println("in lower case letters:"+s.toLowerCase());
```

### 1.3.2 The Code of the Object Oriented Approach

Now we want to write a Java code which allows the simulation of the Buffon needle problem. It is clear that in the problem at hand “needles” will play a central role. Each needle may be described by the $x$– and $y$–coordinates of its two ends. Of course, for each needle we can check whether it crosses a line or not. We draw needles at random so different needles will have different coordinates. However, needles have also generic properties which justify to define needles as a class!

The listing of the `Needle` class can be seen below. We are going to explain basic concepts while discussing this example.

```java
/**
 * A class representing a needle of the Buffon Needle Problem
 **/
5 public class Needle {

   // These are the data fields of the class
   double needleX1, needleX2, needleY1, needleY2;

   /* we have to fix some constants : class variables
    * these can be changed !! */
10  static int needleLength = 20;  // Define length of the needles
   static int numberOfRows = 10;  // Define number of rows
   /* Define the size of the table on which the needles live .
   * These are fixed constants and can not be changed */
15  final static int tableHeight = needleLength * numberOfRows;
   final static int tableWidth = 200;

   /** This is the main constructor :
    * It calculates the position of the Needle object */
```
public Needle() {
    double deltaX, deltaY;
    int ysign;

    // Create the coordinates of starting point
    this.needleX1 = tableWidth*Math.random();
    this.needleY1 = tableHeight*Math.random();
    // Create the coordinates of end point
    deltaX = needleLength*Math.sin(2*Math.PI*Math.random()-Math.PI);
    deltaY = Math.sqrt(needleLength*needleLength-deltaX*deltaX);
    ysign = (Math.random()<0.5?-1:1);
    this.needleX2 = this.needleX1+deltaX;
    this.needleY2 = this.needleY1+ysign*deltaY;
}

/** This is the method CrossInspection which checks whether
   the Needle crosses a line or not. */
public boolean crossInspection() {
    for (int yLine=0; yLine<=needleLength*numberOfRows; yLine+=needleLength) {
        if ((needleY1<yLine && needleY2>=yLine) ||
            (needleY1>=yLine && needleY2<yLine)) {
            return true;
        }
    }
    return false;
}

The main structure is as follows: The first lines define the variables and constants, which a needle should have, like the length and the position of the needle. These are both properties of each individual needle contrary to the table dimensions. These are still properties of the problem, but they belong to a whole set of needles, not only to one. This is visible in the static modifier used in front of the variable definition. So we could actually define more than one table each consisting of many needles and having different table dimensions, let us call them planes.

The next part is the constructor of the class. The constructor is the code, which gets called if a new object of this class is created with the new keyword. Here you call the constructor by using Needle draw = new Needle(); and the constructor just creates a new needle for this collection of needles belonging to this plane.

The third part is a method, which returns a boolean variable, telling us if the needle we are just using is crossing a line of the plane or not. The difference to the procedural approach enters here: How does the method know, which needle to check? This is coded into the calling program, which takes an object of the needle class and let the method crossInspection() operate on this object. Because the object consists of all the information necessary for the method, all the data necessary is already supplied.

Now we are going to talk about these topics in greater detail and try to structure all the ideas we have just encountered.

1.3.3 Class variables, Constants and Modifiers

The needle class has four fields which correspond to the four coordinates which specify the position of the needle. The term fields is used in Java as a synonym for variables. These are the variables different for each needle.

Furthermore, we need some specification for the geometry involved in the problem. They are of course not specific to the individual needle, but for all the needles of a given setup or table. To tell Java that these variables are for all needles in this setup, you supply the static keyword. The static modifier defines
a field (variable), which is belonging to the class and not to the object. So every object of that class has the same value for a `static` field.

The `public` modifier on the other hand defines a field or method, which is special to an object not to the class – this is also called an instance field or method, the `static` version is called a class variable or method.

Some of the fields are defined with the `final` keyword. This is the equivalent to the const keyword in C. Here the variables with the final keyword are not allowed to change anymore, e.g. the table dimensions.

There is a nice feature to be used with the final keyword. You can define a final variable, actually computed in an arbitrary method. The method is then executed before any other code of the program, even before the main method. An example would be:

```java
public class TestFinal {
    public static final double Pi = computePi(); // call method

    public static double computePi() {
        System.out.println("I am doing precalculations!");
        return Math.PI;
    }

    public static void main(String[] args) {
        System.out.println(Pi);
        System.out.println(Pi);
    }
}
```

In Table 1.1 we give an overview of some available modifiers. A graphical representation of the access control of variables and objects is in Figure 1.3.

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>final</code></td>
<td>variables may not be changed, methods can not be overwritten, classes may not be subclassed</td>
</tr>
<tr>
<td><code>public</code></td>
<td>accessible from anywhere</td>
</tr>
<tr>
<td><code>static</code></td>
<td>defines a top-level class, a class variable (field) or a class method</td>
</tr>
<tr>
<td><code>private</code></td>
<td>only within the defining class visible, not in other packages, even if subclass of this class</td>
</tr>
<tr>
<td><code>protected</code></td>
<td>accessible within the package in which it is defined and within subclasses</td>
</tr>
<tr>
<td>(none)</td>
<td>accessible only in its package</td>
</tr>
</tbody>
</table>

It is important to remark that class variables and methods are the closest relatives to global variables in all the other languages. They are accessible from all classes, but still are belonging to a class. So you could have two class methods with the same name, but for different classes.

Having fixed also some constants, e.g. \( \pi \) and the distance between parallel lines, we have to initialize the class. This is done by means of the constructor.

### 1.3.4 The Constructor

An important part of any class is the constructor of the class. This is the method (function), which is called when an object of this class is instantiated. It always has the same name as the class itself. The constructor can have zero or more arguments and you can even have different constructors depending on the parameters provided by the calling syntax.
1.3. THE OBJECT ORIENTED APPROACH - CLASSES AND OBJECTS

### Figure 1.3: A graphical overview of the access control of variables and objects/classes in Java.

The constructor of a class ALWAYS calls the constructor of its superclass, so it is good practice to write `super();` at the beginning of a constructor to indicate this feature. Because the constructor of the superclass also calls the constructor of its super class again, this is called “Constructor Chaining”.

In our example a needle is initialized in the following way. We draw at random the four coordinates `needleX1`, `needleX2`, `needleY1`, and `needleY2` which determine the position of the first needle. Note, that we have made use of the keyword `this`. This keyword is always required when the argument of a method or a local variable in a method have the same name as one of the fields in the class. If the method is simple, as it is the case here, it is not necessary to be that careful.

Here in our example, the constructor just sets the four coordinates and does not need any parameters to instantiate the needle.

#### 1.3.5 Methods and Class Methods

Having initialized the class we now want to define some methods in our `Needle` class. There is only one method in our class, which is the `crossInspection()` method. The return type of methods always has to be specified, if there is no return value you have to use the `void` statement (as in C). The value to be returned is specified by the `return` keyword.

The method `crossInspection` checks whether a needle crosses a line or not. The `crossInspection` method returns a boolean telling you, if the needle crosses a line or not. Note that the variable `cross` has been defined as a class variable with the help of the modifier `static` and gets incremented every time you check a needle, which crosses a line. With this method our `Needle` class is complete.

Next we need to look at the class `Buffon.java` which contains the `main` method and demonstrates how to use the `Needle` class, which is our first example of object oriented coding.

```java
/**
 * This is the program Buffon which simulates the buffon needle using the Needle object.
 * Object Oriented Approach */

import Needle.*;
```
public class Buffon {
    public static void main(String args []) {
        Needle draw;
        int hit; // how many needles hit one of the lines
        final int N=20000; /* variable may not be changed by the program
                        N denotes the number of trials */

        /* In the following loop we draw N Needles.
           To this end we have to create needles with the constructor
           of the Needle class.
           We check whether a Needle crosses a line with the
           crossInspection method of the Needle class. */
        hit=0;
        for (int i=0; i<N; i++) {
            // create a new needle
            draw = new Needle();
            // check if needle crosses a line
            if (draw.crossInspection() == true) {
                hit++;
            }
        }

        /* Finally we print the result */
        System.out.println( "Table width = " + Needle.tableWidth);
        System.out.println( "Table height = " + Needle.tableHeight);
        System.out.println( "Number of Rows = " + Needle.numberOfRows);
        System.out.println( "Needle length = " + Needle.needleLength);
        System.out.println( "Crossings = " + hit + " : N = " + N);
        System.out.println(" Estimated value of \pi = " + 2*(double)N/hit);
    }
}

By defining the Needle class in Java, we have created a new data type. Variables of this type can be declared by

Needle draw;

draw is simply a name that refers to a Needle object (references the object). Creating dynamically an object is done with the help of the new keyword

draw = new Needle();

Next, in a for loop we draw N needles and with the help of the crossInspection method we check whether the needles cross the lines or not and the class variable cross is set accordingly. Finally, the result, i.e. the estimated value for π is printed.

1.4 Another Example: Calculating the Mean

Another example demonstrates how to create a class for calculating the mean of a data set. The mean of a data set $x_i, i = 1, \ldots, N$ is of course defined as

$$ \text{mean } \langle x \rangle = \bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i $$
The mean is also called the first moment as we will see later and there are higher moments. They are defined analogously as

\[ \langle x^n \rangle := \frac{1}{N} \sum_{i=1}^{N} x_i^n. \]

We want to show the different approaches of programming models in Java. You have basically three choices, which are now presented.

### 1.4.1 “Program in One File” Approach

First we define variables to set the moments we want to compute. Then we instantiate and create the data array as an array of random numbers between zero and one. Then after setting all the moments to zero, we compute the moments in a nested loop. The last step is dividing through \( N \) and printing the final result on screen.

```java
import java.util.Random;

public class Moments {
    public static void main(String[] args) {
        Random rand = new Random();
        // Number of points used
        final int N=50000;
        // what moments are to be calculated
        int moments_start=1;
        int moments_end=20;
        // declare and instantiate arrays
        double[] moments = new double[moments_end-moments_start+1];
        double[] numbers = new double[N];

        // create N random numbers
        for (int i=0; i<N; i++) {
            numbers[i] = rand.nextDouble();
        }

        // initialize array of moments
        for (int i=0; i<=(moments_end-moments_start); i++) {
            moments[i] = 0;
        }

        // Calculate all the moments
        for (int i=0; j<N; j++) {
            for (int i=0; i<=(moments_end-moments_start); i++) {
                moments[i] += Math.pow(numbers[j], i+moments_start);
            }
        }

        // display the moments
        for (int i=0; i<=(moments_end-moments_start); i++) {
            System.out.println("Moment #"+(i+moments_start) + " exact: \frac{1.0}{1+i+moments_start} = \frac{1.0}{1+\#moments})");
        }
    }
}
```

### 1.4.2 “Traditional Procedural” Approach

Now we take the computation of the moments and put it into a method. Then we call the method from our main program. Because there is not much difference between the program before and this one, we just
present the differences.

```java
// initialize array of moments and calculate
for (int i = 0; i <= (moments_end - moments_start); i++) {
    moments[i] = 0;
    // Call method for calculation
    moments[i] = calcMoment(numbers, i + moments_start);
}

/** Method for computing the moments */
public static double calcMoment(double[] array, int N, int moment) {
    double result = 0;
    // Calculate all the moments
    for (int j = 0; j < N; j++) {
        result += Math.pow(array[j], moment);
    }
    return result / N;
}
```

### 1.4.3 Object Oriented Approach

There are two programs: The first class is the main program using our own class. This has to be called. The second one is the code for the data class itself and have to be compiled only. You can not start this class.

We have written two different versions to express the difference between a class method and an object method.

```java
/** A class for data sets in 1D
   You can call the computation of the moments
   with this class in 2 different ways. */
public class MomentsData {
    private int N;
    private double[] data;

    /** Instantiate an object holding the data and the size
     of the data set */
    public MomentsData(double[] array) {
        this.N = array.length;
        this.data = new double[this.N];
        System.arraycopy(array, 0, this.data, 0, this.N);
    }

    /** Another possible constructor */
    public MomentsData(double[] array, int N) {
        this.N = N;
        this.data = new double[this.N];
        System.arraycopy(array, 0, this.data, 0, this.N);
    }

    /** The version 1, which uses the data of the object */
    public double calcMoment(int moment) {
        double result = 0;
        // Calculate all the moments
        for (int j = 0; j < this.N; j++) {
            result += Math.pow(this.data[j], moment);
        }
        return result / this.N;
    }
}
```
/** The version 2, which uses a parameter for the data */
public static double calcMoment(double[] array, int moment) {
    double result = 0;
    int N = array.length;
    // Calculate all the moments
    for (int j = 0; j < N; j++) {
        result += Math.pow(array[j], moment);
    }
    return result / N;
}

The class MomentsData includes some new code. First we use two constructors, which have different arguments and can therefore be distinguished by Java. Depending on the calling syntax using the new keyword, you call the first or the second constructor.

Then there are two methods defined. Like with the constructor, they have the same name, but can be distinguished by its argument list. This is called method overloading in Java. So again depending on the calling syntax, you call the first or the second method.

Now we take a look at the main program, which uses the above classes.

```java
import java.util.Random;

/** the object oriented version using two different approaches */
public class MomentsObject {
    public static void main(String[] args) {
        Random rand = new Random();
        // Number of points used
        final int N = 50000;
        // what moments are to be calculated
        int moments_start = 1;
        int moments_end = 20;
        // declare and instantiate arrays
        double[] moments = new double[moments_end - moments_start + 1];
        double[] numbers = new double[N];
        // create N random numbers
        for (int i = 0; i < N; i++) {
            numbers[i] = rand.nextDouble();
        }

        /** Version 1
        // instantiate an object of class MomentsData called dat!
        MomentsData dat = new MomentsData(numbers);

        // initialize array of moments and calculate
        for (int i = 0; i <= (moments_end - moments_start); i++) {
            moments[i] = 0;
            // Call method for calculation !!!!!!!!!!!!!!!!
            moments[i] = dat.calcMoment(i + moments_start);
        }
        // display the moments
        for (int i = 0; i <= (moments_end - moments_start); i++) {
            System.out.println("Moment "+(i+ moments_start)+": "+ moments[i]);
        }
    }
}
```

System.out.println();
40 /* Version 2 **************************************** **/
for (int i = 0; i <= (moments_end - moments_start); i++) {
    moments[i] = 0;
    // Call method for calculation !!!!!!!!!!!!!
    moments[i] = MomentsData.calcMoment(numbers, i + moments_start);
}
// Display the moments
for (int i = 0; i <= (moments_end - moments_start); i++) {
    System.out.println("Moment "+(i + moments_start)+": "+moments[i]);
}

Up to line 21 there is nothing new, we just instantiate and create the data array. Then from line 22 to line 38 the first version using instance methods is used. The second version from line 40 to line 50 is using a class method. But both versions do exactly the same. The only difference is line 24 together with line 30 compared to line 44 in the second version.

1.5 Interfaces and Abstract Classes

Abstract classes/methods are also special classes, which can not be instantiated and contain no code for the (abstract) methods. If you define some methods, which contain no code as abstract, you have to define the whole class as abstract. You can subclass from an abstract class and override the abstract methods. You do not have to override all abstract methods, but then the subclass is abstract too.

An even more important concept in Java is the interface. An interface is basically an (abstract) class, which does not implement all the methods defined in the class. All methods have automatically the public modifier and are abstract. Therefore you can create classes defining abstract methods, which should be implemented somewhere else, maybe system dependent, e.g. if you use the native modifier. Interfaces can be subclassed to create new interfaces.

Most of the GUI classes of the AWT introduced later are actually interfaces or abstract classes and not just ordinary classes. Very useful is this concept for passing methods as arguments to other methods (see section 1.8).

1.6 Extending (Inheritance) and Overloading (Overriding) Classes

Often you want to define subclasses, which should inherit all the methods and fields from another class. This is easily done in Java by extending a given class. The meaning of sub-classing is the notion of data hiding or encapsulation. For example you can write a subclass, which can not access all the variables of the super class, therefore hiding some details.

You can reference the super (parent) class by applying the super modifier in front of a method or variable of the parent class. And the this modifier always refers to the actual class.

You can also have equal names for a variable in the super class and the child class, which means you have to reference the variables explicitly by using the super or this keywords. If you want to refer to a variable two classes up from the actual class, you have to use the notion of “shadowing” [Flanagan, 1997], which is a kind of casting with classes.

If you define a class to be final, it can not be extended. For example the java.lang.System class is a final class.

Different from C++, you can not inherit from more than one class in Java, meaning that there is always only one superclass for each class. The only way of having multiple inheritance is by using interfaces, which we will not cover extensively in our introduction.

If you write a subclass you can overwrite methods already defined in the super class. This is called overloading of methods, analogous to the C++ overloading. But in C++ you can even overload operators
like +, -, etc., which is not possible (in Java 1.1) yet.

A simple example is given by the HelloWorld Applet.java program in section 0.3.1. There we have extended the Applet class of the Java.applet package and therefore making our program a subclass of the Applet class. So we inherited all the methods and fields of that class. Then we overloaded the paint() method to display our message. In the words of object oriented programming, writing an applet is called: defining a subclass of the Applet class and overloading the methods of the Applet class as necessary.

One nice and important feature in Java is, that all classes which do not have an explicit parent, inherit from the java.lang.Object class. So you can call this class the father of all classes. There are only a few methods defined in this (abstract) class, which you can always override. For example the toString() method is in java.lang.Object. If you override this method, you can define your own objects, which can then be printed by the usual println() commands.

As an example let us write a simple class, which

In Table 1.2 all methods defined in java.lang.Object are displayed.

<table>
<thead>
<tr>
<th>Table 1.2 All methods belonging to the (abstract) java.lang.Object class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean equals (Object obj);</td>
</tr>
<tr>
<td>protected Object clone() throws</td>
</tr>
<tr>
<td>ClonedNotSupportedException, OutOfMemoryError;</td>
</tr>
<tr>
<td>public String toString ();</td>
</tr>
<tr>
<td>public int hashCode();</td>
</tr>
<tr>
<td>protected void finalize() throws Throwable;</td>
</tr>
</tbody>
</table>

If you want to find out about all methods and fields available in the java.lang.object class in detail, you just take a look at the Java API documentation of the JDK. There in the package API documentation you click on the package java.lang and then on the class object and you get the full description of the class with all methods and fields.

1.7 The System Class: Screen-Output and Keyboard-Input

Now we are in a place to discuss the System.out.println() statement already used in the “HelloWorld” program. This is calling the println() method of the PrintStream class of the java.io package. And the out is a variable from the System class of the java.lang package, referencing the PrintStream class.

There are also two more variables called err and in for error output and data input. The java.lang.System class in general provides an platform-independent interface to some system functions.

Here an example demonstrating some of the material we have learned and explains how to get any kind of input from the keyboard. It waits for an user input and just echoes the typed characters until you type the word “Java”.

```java
/** A program to demonstrate keyboard input, escape codes and string (object) comparisons */
import java.io.*;

public class System_Class {
    public static void main ( String [] args ) throws IOException {
        /* create object to read lines of text from the user */
        Reader read_buffer = new InputStreamReader(System.in); /* first: a reader object for System.in (keyboard) */
        BufferedReader input = new BufferedReader (read_buffer);
        System.out.println("To exit please type the word \"Java\".");

        for (;;) { // Loop forever until input of the word "Java"
```
System.out.print("Please type a word:");
/* Read a line from the user using readLine() */
String line = input.readLine();
/* If we reach the end-of-file, 
or if the user types "Java", then quit */
if ((line == null) || line.equals("Java")) break;
/* Otherwise just echo the word */
System.out.println("You typed: "+line+"\n");

Let us analyze the program: First look at line 19, where we have used the print() method of the System class. It does the same as println(), but does not jump to the next line after the output.

Then take a look at line 23, where we compare two strings. We have to use the equals() method of the String class to compare the value of two strings and not the references. In line 25 we use the string concatenation operator + for the output.

By the way, you have probably already noticed, that the for(;;) loop in line 17 is a endless loop.

The actual input takes place in line 21, where we assign the input to the string line. The method used is readLine(), which is a method of the BufferedReader class. It reads input until a carriage return is reached. The actual object of the BufferedReader class is created in line 13. To that end we have to create a Reader class for the InputStreamReader called in just mentioned above (this is done in line 11).

Because we are using I/O commands, we have to take care of exceptions which can occur during the I/O. For that reason we have to use the throws IOException statement in the definition of the main class (see line 7).

A last remark concerns the escape codes used in line 15 and line 25. The first one, /" displays a quotation mark and the second one a newline. To include arbitrary formatting (escape) characters to the string supplied to the print() and println() methods, you can use similar to C the \n, \t or \" codes to get a newline, a tab or a backslash in the output. You should also note that, if you use e.g. System.out.println(5+7) you get 12 as output, so if you want to see 5 7 you have to use System.out.print(5+" "+7)

Although this looks very complicated, if we do not want to go into details, just copy this part to a program and reuse the code. But after we get used to object oriented programming we will not have any problems understanding the code above anymore.

1.7.1 Easy Input and Lava Rocks printf()

Another easy solution is to use the EasyIn and the Printf() methods, supplied by our simulation class and the "Lava Rocks"\(^3\) package.

Easy Input You can easily use these methods to get input from keyboard for different primitive data types. For example to input a primitive data types, you just use

```java
import simulation.*;
double d = EasyIn.readDouble(); // reads double from System.in
int i = EasyIn.readInt(); // reads int from System.in
float f = EasyIn.readFloat(); // reads float from System.in
boolean b = EasyIn.readBoolean(); // reads boolean from System.in
```

\(^3\)This is a freely available package, containing some easy to use methods, mostly for C programmers who switched to Java. For details see Rose [1999].
Lava Rocks – printf()/sprintf()/fprintf() To use the Lava Rocks package you could write a code like

```java
import lava.clib.*;

int i=100;
float f=165.234f;
Stdio.printf("%8d and %8.1f : test text\n", new Object[]{
    new Integer(i), new Float(f)
});
```

Remember that you need the `lava.jar` file in the classpath. The output will look like

```
100 and 165.2 : test text
```

If you are familiar with the C routines, then you can find all the modifiers used for formatting the different types of variables in Table 1.3. The modifiers represent the place, where the actual value of the variable has to be inserted before the output is sent to the screen (or file, or string). So you have to make sure that the order of the modifiers and the order of the supplied variables is correct, otherwise you get unpleasant results or strange errors.

There are three additional things to mention: If you use the same format string (the first argument to the printf method) very often, you can speed things up by saving the format string and only use the variable every time, like:

```java
import lava.clib.stdio.*;

PrintfFormatString fmt =
    new PrintfFormatString("%8d and %8.1f : test text\n");
```

Now you can use `fmt` instead of the string in all printf commands and it will be much faster.

There is also an “easier” way of using printf without creating an object array, which is much more inefficient and should be avoided.

And there is a platform neutral code for newlines: Use `%\n` instead of `\n` in the format strings and you always get a newline, no matter which platform you run the program.

The same holds for the other two methods sprintf(), which writes the formatted output to a string-buffer (we will discuss this in section 3.4.1 and in section 10.3.5 in more detail.), and fprintf(), which writes the output to a file (actually a Writer, see section 3.4.1).

### Table 1.3 All possible modifiers to be used in the format string given to the printf()/sprintf()/fprintf() methods supplied by the Lava Rocks package.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Type to be formatted</th>
</tr>
</thead>
<tbody>
<tr>
<td>%bd</td>
<td>byte</td>
</tr>
<tr>
<td>%hd</td>
<td>short</td>
</tr>
<tr>
<td>%d</td>
<td>signed integer</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>%o</td>
<td>unsigned octal integer</td>
</tr>
<tr>
<td>%x / %X</td>
<td>unsigned hexadecimal integer (lower or uppercase)</td>
</tr>
<tr>
<td>%f</td>
<td>float</td>
</tr>
<tr>
<td>%lf</td>
<td>double</td>
</tr>
<tr>
<td>%e / %E</td>
<td>float, scientific notation</td>
</tr>
<tr>
<td>%g / %G</td>
<td>float, same as f or e, depending on value</td>
</tr>
<tr>
<td>%s</td>
<td>String</td>
</tr>
<tr>
<td>%c</td>
<td>character</td>
</tr>
<tr>
<td>%p</td>
<td>object identity hash code in unsigned hexadecimal</td>
</tr>
<tr>
<td>%\n</td>
<td>platform independent line separator</td>
</tr>
<tr>
<td>%n</td>
<td>counts characters</td>
</tr>
</tbody>
</table>
There is another free implementation of the printf command for Java called `Format.java` of the `corejava` package. But it has much less functionality and so we decided to use the Lava Rocks implementation.

### 1.8 Passing Arguments to Methods

We want to review all the aspects concerning the passing of arguments to methods or – for former FORTRAN programmers – passing arguments to subroutines/functions.

**Global Variables** First of all there is of course always the possibility to pass variables as global variables. This is easy to use, but it makes it difficult to follow a program structure. So it is preferred to use arguments to methods in an argument list.

**Primitive Data Types** The important point to note here is that Java always passes primitive data types by value and all the other arguments by reference (reference data types). This means that all primitive variables of the argument list can be changed within the method. The dummy variables used in the method can be viewed as local variables.

**Reference Data Types** The reference data types (e.g. arrays, objects, etc.) are passed by reference and therefore only the memory address is passed to the method. That is again the reason that by using the standard assignment or comparison operators, you do not compare the values of the reference data type, but the memory addresses where the data is stored. Therefore there is the `equals()` operator used for comparing reference data types. By the way there is no method of getting the actual memory address for a reference data type, which would be a severe security problem.

If you pass a reference data type to a method it should be clear, that every change of the dummy variable used for the reference data type used in the method, will result in a change of the passed reference variable. So for example for arrays:

```java
public void main (String [] args) {
    ...  // ... implementation...
    int [] array = {1, 2, 3, 4, 5};
    test (array);
    ...  // ... implementation...
}
public void test (int [] dummyarray) {
    int length = dummyarray.length;
    for (int j = 0; j < length; j++) {
        array[j] = 0;
    }
}
```

In this example we pass an array of integers, where the values are all not zero. In the subroutine we change the values of the `dummyarray` array. But we passed only the memory location, so the change is actually a change in the `array` array. If you print the result after the call to the method, all elements are zero!

If you now think about changing a primitive data type inside a method passed to it, there is none. There is actually no simple way of using reference types for primitive data types to change method parameters.

**Instance Variables** A nice and clear way of passing arguments between methods, is using instance variables. The problem here is that you have to create a new class and define instance variables for it. Then you can instantiate an object of this new class and change the values of the instance variables from the main program or the methods therein. But still you have the problem of passing the instance to the methods.
Multi-Dimensional Arrays as arguments  As a scientist you will always have the necessity to use arrays as arguments to methods. There is actually no trouble with using for example a 1D array to a method, which expects a 1D array. But what if you have a 2D array and the method needs a 1D array as an argument? Should you copy a 1D row or column of the 2D array into a new 1D array and then call the method or is there a better way?

The answer is yes, but not in all cases. You can always omit the last dimensions of an array. For example if you leave out the second dimension of a 2D array, you get a 1D array, which is just the desired row of the array. But there is no way of referencing a column. The following program demonstrates the usage.

```java
/**
 * testArray.java
 */

public class testArray {

    public static void main(String[] args) {
        /* create and instantiate a 2D array */
        double[][] array2;
        array2 = new double[10][10];
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                array2[i][j] = j + 10 * i;
            }
        }
        test(array2[1]); // call the method with a row of a 2D array
        // test(array2[][]); // This is WRONG !!!!
    }

    /** The method prints the 1D array argument to the standard output to check the result. */
    static void test(double[] array1) {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + "=" + array1[i]);
        }
    }
}
```

Methods (Functions) as Arguments  Especially for scientists it is of great importance to know how to pass a function (method) to another method.

The first solution would be to write a method as a class method, therefore accessible from every instance of the class and so you do not need to pass the method as an argument at all.

The second solution is much more versatile and general. But as always you have to do more work. You need an interface, which defines the function you would like to pass to a method (Declaration). Then you need a class, which implements this interface, therefore representing the real function (Implementation). And at last you write your program/class, which instantiates the former class and now you can pass the function by reference to all methods you like (Usage).

Here a short example:
CHAPTER 1. OBJECT ORIENTED PROGRAMMING

Declaration

interface function {
    double f (double x);
}

Implementation

class SquareFunc implements function {
    public double f (double x) {
        return (x*x);  // <----- here is the actual function
    }
}

Usage  This is the program, which uses the two classes/interfaces above.

/**
 * TestPassingFunctions.java
 */
public class TestPassingFunctions {
    public static void main (String[] args) {
        SquareFunc f = new SquareFunc();
        int points = 10;
        double integral = integrate (f, 0.1, points);
        System.out.println("The integral is: " + integral);
    }
}

private static double integrate (function f, double a, double b, int p) {
    double integral = 0;
    double dx = (b-a)/p;
    for (double x=a+dx/2; x<b; x+=dx) {
        integral += f.f(x);
    }
    return integral *dx;
}  // TestPassingFunctions

1.9 Structure and Overview of Java

1.9.1 Packages in Java 1.1 and Java 2

Here we list all the standard packages (APIs) included in the Java 1.1 language standard. The important packages (for our purposes) are written in small caps.

Java 1.1 Packages

JAVA.APPLET  includes the superclass of all Java applets (small package)
JAVA.AWT  the Abstract Windowing Toolkit (large package)
java.awt.datatransfer  provides data exchange between programs
JAVA.AWT.EVENT  event handling for the AWT (Mouse, etc.)
java.awt.image  rarely used classes for image processing (use AWT)
1.9. STRUCTURE AND OVERVIEW OF JAVA

java.awt.peer rarely used interfaces for the AWT

java.beans interfaces and classes for beans programmer

JAVA.IO all the input/output classes (very big)

JAVA.LANG central Java language classes (largest package)

java.lang.reflect part of the Java Reflection API (small)

JAVA.MATH arbitrary precision arithmetic (small)

java.net networking package

java.rmi RMI is Remote Method Invocation. It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine.

java.rmi.dgc RMI distributed garbage-collection (DGC).

java.rmi.registry Methods to access the RMI registry.

java.rmi.server RMI server classes and methods.

java.security A security framework. This includes classes that implement an easily configurable, fine-grained access control security architecture.

java.security.acl / java.security.cert / java.security.interfaces / java.security.spec Additional security features.

java.sql Provides the JDBC (Java Database Connectivity) package. JDBC is a standard API for executing SQL statements.

java.text for writing internationalized programs (date, time, etc.)

JAVA.UTIL useful classes, often used, e.g. millisecond time, calendar, random numbers, vectors, etc.

java.util.zip data compression and decompression classes

Java 2 Packages

Additional API packages in Java 2 are:

java.awt.color Provides classes for color spaces.

java.awt.dnd Provides interfaces and classes for supporting drag-and-drop operations.

java.awt.font Provides classes and interface relating to fonts. It contains support for representing Type 1, Type 1 Multiple Master fonts, OpenType fonts, and TrueType fonts.

java.awt.geom The Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.

java.awt.im Classes and an interface for the input method framework. This framework enables all text editing components to receive Japanese, Chinese, or Korean text input through input methods.

java.awt.image.renderable Classes and interfaces for producing rendering-independent images.

JAVA.AWT.PRINT A general printing API, including document types, page setup and formats and job control dialogs.

java.beancontext A bean context is a container for beans and defines the execution environment for the beans it contains.
java.lang.ref Provides reference-object classes, which support a limited degree of interaction with the garbage collector.

java.rmi.activation Provides support for RMI Object Activation.

java.util.jar For reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file.

javax.accessibility Defines a contract between user-interface components and an assistive technology that provides access to those components.

JAVAX.SWING Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.

java.swingx.* There are 15 more packages in the Swing package, which we are not describing here in detail. Take a look at the Swing tutorial or documentation.

org.omg.CORBA Provides the mapping of the Object Management Group (OMG) CORBA \(^4\) APIs to the Java programming language, including the class Object Request Broker (ORB), which is implemented so that a programmer can use it as a fully-functional Object Request Broker.

org.omg.* 6 more packages are provided for using CORBA with Java. See the API documentation.

### 1.9.2 Reserved words in Java

The following words are reserved for Java and cannot be used for names of classes, variables or methods:

abstract, boolean, break, byte, byvalue, case, cast, catch, char, class, const, continue, default, do, double, else, extends, false, final, finally, float, for, future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outher, package, private, protected, public, rest, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while, widefp

### 1.10 Name Conventions in Java

In order to make the Java codes more readable it is customary to stick to the following name conventions.

**Classes and Interfaces.** The names of classes and interfaces should consist of one or more words which are concatenated. They should describe appropriately the class and the interface. The first letter of the name is an uppercase letter. If the class name consists of more than one word, each word after the first one begins with an uppercase letter, e.g., Needle, Reader, StringToken.

**Methods.** The names of methods are verbs or verb–phrases. They begin with lowercase letters. If the name consists of more than one word, the second and all the following words begin with an uppercase letter. To give some examples:

- methods, which set the value of a variable or return the value of a variable begin with set or get; e.g., setName, getData.
- methods, which check some condition and return a result get the prefix is, e.g., isSmaller.
- methods, which simply perform some conversions are characterized by the returned type and have the prefix to, e.g., toString.

\(^4\)Common Object Request Broker Architecture
If you stick to these rules you are already writing programs, which conform to the Java beans syntax. This means you can take your programs and use them as beans later on with almost no changes (see section 1.15).

**Instance Variables.** Names for instance variables are words, nominal phrases, or short-hand notations. Like names for methods the word begins with a lowercase letter and all following words begin with an uppercase letter, e.g., `next`, `dataVector`, `minValue`.

**Local variables or parameters.** Local variables should get short names. Usually, they are named by sequences of small letters. Typical examples are acronyms (the first letter of each word) of the name of the class for a variable, which keeps a reference to an instance of the class, e.g., `rv` for Random Variable or short-hand notations, e.g., `minx` for the minimal value of the variable `x`. Single letter names should be avoided, unless for temporary variables or variables, which are used in loops or for variables with an uncertain value of some type: `b` for `byte`, `c` for `char`, `d` for `double`, `e` for `Exception`, `f` for `float`, `i`, `j`, `k` for `int`, `l` for `long`, `o` for `Object`, `s` for `String`.

**Constants.** The names of constants may be composed by one or more words. All characters are uppercase letters and the words are joined by `_`, e.g., `PI`, `MIN_VALUE`.

**Packages.** Packages, which are only used locally are identified by a name which begins with a lowercase letter. This word can not be `java`; this keyword is reserved for standard Java classes.

### 1.11 Java Documentation

As the time of writing the JDK (1.1 or 2) is distributed in a twofold way: The first one is the Java Runtime Environment called JRE, which includes the Java virtual machine, the Java plugin for the Internet Explorer and Netscape to run Java applets.

The second one is the complete package – the JRE and additionally all development tools to write Java programs. The API documentation is included in the full package, but not in the JRE. The documentation for the APIs is also available separately.

In the directory `JAVA_HOME`, on UNIX this could be `/usr/local/jdk1.2`, `/usr/local/lib/jdk1.2` or `/usr/lib/jdk1.2` and on Windows this might be `c:\jdk1.2`, you can find all the documentation in the subdirectory `docs`. It is best viewed using a web browser starting with the page `jdk1.2/docs/index.html` (see also Figure 1.4).

### 1.12 Applications and Applets Revisited

After discussing object-oriented programming, we want to recapitulate the basic differences and features of the two possible ways of writing and starting Java programs.

#### 1.12.1 Applications

A Java application is in the traditional language a “normal” program. In Java slang it is a class, which only has to have a `main` method, which has to be `static` and `public`. This is also of course the entrance point, if you start the Java application. Then the program executes sequentially the code given in the main method (of course it goes parallel, if you use threads somewhere in the application - see 10.3). There are no restrictions of any sort using features in an application.
1.12.2 Applet

An applet is a Java class, which can be loaded into a virtual machine and can then be executed by it. For example a web browser could load the applet, check if it is an applet allowed to be started on the machine the browser is running on and then “starts the applet”.

This is basically the most confusing part of an applet: What are the instructions, which get executed when the browser “starts the applet”? To that purpose go ahead and write a simple program (see the program `ShowTrace.java` in the appendix B.1), which only prints out the place where the execution just takes place and “start” the Java class on the command line as an application, in the appletviewer as an applet and in a browser as an applet. What you will get is shown in Figure 1.5.

So for the applets you have to write (implement) at least the `init()` method to get things going. But now you will ask, why did we not supply an `init()` method when we wrote our first HelloWorld applet? The answer is we did, but we did not write it explicitly, because we left it empty. The trick was that for an applet there is always another thread (see Chapter 10.3 for an introduction to threads) running, which takes care of (re)painting the windows (panels) used by the applet. This thread is a method of the AWT package (see Chapter 2.2) and the method is the `paint()` method. And in the HelloWorld applet we have overridden the `paint()` method to display our message.

Actually the threads and methods used for painting and repainting the windows or panels is a little bit more tricky and involved, so we have to postpone the discussion to a later chapter. Here only the basics: There is a method `repaint()`, which calls the `update()` method and that in turn calls the `paint()` method, when there is time to do so. This sounds very complicated in the first place, but it will be resolved later on.

There is another issue to be addressed here in the context of applets. So far we have always written a separate HTML file to use with the appletviewer, which then in turn calls the applet itself. The burden of having two files for one applet can be avoided by putting the HTML Code at the beginning of the Java class file in a comment. Then if you call the appletviewer with the Java class file it executes the HTML code supplied in the Java class file and starts the applet. It even starts in the Netscape Navigator, although it probably makes no sense for large programs embedded in a set of HTML pages kept updatate in a different way as the Java source code. But it certainly eases writing small applets and getting not confused by too many files on your disks.

Here a small example showing the described feature:
1.12. APPLICATIONS AND APPLETS REVISITED

Figure 1.5: The line of execution in an application and an applet in the appletviewer or the Netscape Navigator 4.08. For the application only the part above the first line and below the second line are actually the parts, which can not be avoided. The remaining part is just provided to show you how to write an application, which can be used as an applet, too.

```java
/**
 * test_Applet.java
 */

/*<HTML>
 <APPLET code="test_Applet.class" width=500 height=800>
 </APPLET>
 */

import java.applet.*;

public class test_Applet extends Applet {
    public void init() {
        System.out.println("!!! I am alive !!!");
    }
}
```

You can start it (first you have to compile it) by typing `appletviewer test_applet.java` or in the Netscape Navigator type `file:/home/user/test_Applet.java`. Then on the command line or the terminal with which you started the browser you see the short message.

1.12.3 Programs as Applets and Applications

?? GUIs ....

To write a class to be run as an applet and an application you can see in Figure 1.5 the correct order of the methods, which get called in both cases. As an example, we have used the following template for the bigger programs to have them run as an applet as well as a standalone application. We have to add code to the `run()` and the `Setup()` methods to get a useful program. The `Setup()` method should contain

---

5 You have to substitute the path by an appropriate one for your system and configuration.
all screen setup or preliminary work. In the run() method goes the computational intensive part of the program.

```java
/** Program, which can be run as an applet or an application */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class TestAppletApplication extends Applet implements Runnable {
    // Thread variables
    Thread current, calcThread;
    static TestAppletApplication prg;

    // AWT Variables
    static int XSize=500, YSize=500;
    static Frame f;

    /** Constructor: set up the Window/Panel with the GUI here */
    public TestAppletApplication () { } // empty constructor

    /** this is called by the application directly or by the browser of the applet */
    public void start () {
        calcThread.start (); // calls the run() method !!
    }

    /** stops the calculation thread, if applet is stopped */
    public void stop () {
        calcThread.stop (); // call the stop() method !!
    }

    /** Main wrapper to run as application: display window and start calc thread */
    public static void main (String [] args) {
        f = new Frame ("Test Program");
        f.setSize (XSize, YSize);
        f.show ();

        prg = new TestAppletApplication ();
        // Close Window event
        f.addWindowListener (new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                prg.stop (); System.exit (0); }
        });
        f.repaint ();

        // start program = Applet
        prg.init ();
        prg.start ();
    }

    /** Starts the calculation in the first place.
    Gets called by the main program or directly by the browser. */
    public void init () {
        if (prg==null) {
            prg = this; // do not call the constructor again in the applet
        }
    }
```
1.13. Higher Mathematics in Java

1.13.1 Standard Mathematical Functions in Java

The standard mathematical functions of Java are declared in the java.lang.Math class, which consists of static constants and methods for common mathematical manipulations. It contains the functions sine, cosine, logarithm, exponential, square root and much more. Do not confuse this class with the java.Math class, which was introduced to Java 1.1 for arbitrary precision arithmetic – we are not discussing this, read the API documentation for details.

Java math is always conforming to the IEEE 754 standard and all algorithms used in the math API are guaranteed to produce the same results as those from netlib's Freely Distributable Math Library\(^6\). fdlibm is a C library for machines supporting the IEEE 754 floating point arithmetic initially developed by SUN. Meanwhile Visual Numerics has developed a package implementing the full fdlibm library in pure Java. Therefore Java need no longer depend on a local implementation of the fdlibm library. This makes it much easier to implement Java for more exotic computers or machines.

Some useful constant in the Math class are Math.Pi (π) and Math.E (e). In Table 1.4 we have summarized some of the most useful methods in the Math class. Note, that all angles are in radians and all parameters and return values are of type double unless otherwise stated.

These are all class methods, so they can be called from anywhere. Here are some examples:

```java
a=Math.exp(2.1);  // e to the 2.1
distance = Math.sqrt(Math.pow(x,2)+Math.pow(y,2));  // Euclidean distance
```

There is no import statement to avoid the Math in front of the methods, we always have to use it, although it seems tedious.

Here is a test program for the three rounding methods available:

```java
public class TestRounding {
    public static void main(String[] args) {
        double a1 = 7.49;
        double b1 = 7.5;
        double c1 = 7.51;
        double a2 = -7.49;
        // ...
    }
}
```

\(^6\)Called fdlibm. Netlib is a collection of mathematical software, papers and databases. It is located at the ORNL (Oak Ridge National Lab in Tennessee) and UTK (University of Tennessee at Knoxville), but there are many other mirror sites. Fdlkm is available online from http://www.netlib.org/ or http://www.hensa.ac.uk/ftp/mirrors/netlib/
Table 1.4 Overview of the mathematical methods available in Java 1.1 in the java.lang.Math class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
</tr>
<tr>
<td>acos(x)/asin(x)/atan(x)</td>
<td>arccosine/sine/tangent</td>
</tr>
<tr>
<td>atan2(x,y)</td>
<td>arctangent (x/y)</td>
</tr>
<tr>
<td>sin(x)/cos(x)/tan(x)</td>
<td>sine/cosine/tangent</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential and natural logarithm</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>smallest whole number ( \geq x )</td>
</tr>
<tr>
<td>floor(x)</td>
<td>largest whole number ( \leq x )</td>
</tr>
<tr>
<td>rint(x)</td>
<td>( x ) rounded to the nearest integer; if neither integer is nearer, rounds to the even one.</td>
</tr>
<tr>
<td>round(x)</td>
<td>(int) ( \text{floor}(x+0.5) ) for float ( x )</td>
</tr>
<tr>
<td>pow(x,y)</td>
<td>( y^x )</td>
</tr>
<tr>
<td>min(x,y)/max(x,y)</td>
<td>minimum (maximum) of ( x ) and ( y ) for any numeric type</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root of a double ( x )</td>
</tr>
<tr>
<td>random()</td>
<td>random numbers (use java.util.Random class)</td>
</tr>
<tr>
<td>IEEEremainder(x)</td>
<td>special remainder function</td>
</tr>
</tbody>
</table>

New in Java 2:

- toRadians(angle): converts degrees to radians
- toDegrees(rad): converts radians to degrees

```java
double b2 = -7.5;
double c2 = -7.51;

System.out.println("round: "+Math.round(a1)+"\n+"+a1);
System.out.println("round: "+Math.round(a2)+"\n+"+a2);
System.out.println("round: "+Math.round(b1)+"\n+"+b1);
System.out.println("round: "+Math.round(b2)+"\n+"+b2);
System.out.println("round: "+Math.round(c1)+"\n+"+c1);
System.out.println("round: "+Math.round(c2)+"\n+"+c2);
System.out.println("ceil: "+Math.ceil(a1)+"\n+"+a1);
System.out.println("ceil: "+Math.ceil(a2)+"\n+"+a2);
System.out.println("ceil: "+Math.ceil(b1)+"\n+"+b1);
System.out.println("ceil: "+Math.ceil(b2)+"\n+"+b2);
System.out.println("ceil: "+Math.ceil(c1)+"\n+"+c1);
System.out.println("ceil: "+Math.ceil(c2)+"\n+"+c2);
System.out.println("floor: "+Math.floor(a1)+"\n+"+a1);
System.out.println("floor: "+Math.floor(a2)+"\n+"+a2);
System.out.println("floor: "+Math.floor(b1)+"\n+"+b1);
System.out.println("floor: "+Math.floor(b2)+"\n+"+b2);
System.out.println("floor: "+Math.floor(c1)+"\n+"+c1);
System.out.println("floor: "+Math.floor(c2)+"\n+"+c2);
System.out.println("rint: "+Math.rint(a1)+"\n+"+a1);
System.out.println("rint: "+Math.rint(a2)+"\n+"+a2);
System.out.println("rint: "+Math.rint(b1)+"\n+"+b1);
System.out.println("rint: "+Math.rint(b2)+"\n+"+b2);
System.out.println("rint: "+Math.rint(c1)+"\n+"+c1);
System.out.println("rint: "+Math.rint(c2)+"\n+"+c2);
```

and the output of this program is:

```
\begin{verbatim}
round : 7 7.49  floor : 7.0 7.49
round : -7 -7.  floor : -8.0 -7.4949
round : 8 7.5  floor : 7.0 7.5
round : -7 -7.  floor : -8.0 -7.5 5
round : 8 7.51  floor : 7.0 7.51
round : -8 -7.  floor : -8.0 -7.5151
ceil : 8.0 7.4  rint : 7.0 7.49  9
```

- The `Math.round()` method rounds a floating-point number to the nearest integer.
- The `Math.ceil()` method returns the smallest integer value that is greater than or equal to the argument.
- The `Math.floor()` method returns the largest integer value that is less than or equal to the argument.
- The `Math.rint()` method returns the double value that is closest to the argument and is equal to a mathematical integer.

These methods are part of the Java programming language's standard library, providing a set of mathematical functions that developers can use in their code.
For most of our simulations, we will need routines (methods) to calculate the mean, variance or other statistical measures of a times series for example. Unfortunately these easy and important methods are not part of the standard Java language. Solutions to overcome this problem are presented in the next sections.

1.13.2 Numerical Libraries - The JNL

The JNL (Java Numerical Library) was designed and implemented by Visual Numerics\(^7\). It is a free library, which supplies additional mathematical functions, complex numbers, statistical operations and some basic vector and matrix operations for scientists (no sparse matrix support), which are missing in the Java standard. They have proposed it as a standard for the next version of Java.

The algorithms used for the JNL are based on the LinPack library\(^8\), which are a collection of programs for linear algebra in C and Fortran.

A short list of some of the functions included in the JNL is given in Table 1.5.

\[^7\text{A well known software company, selling the IMSL math library, PVWave a sophisticated plotting program and much more.}
\text{Visual Numerics Homepage}\]

\[^8\text{See for example at the NetLib repository.}\]
Table 1.5 A short list of JNL classes supplied with JNL 1.0 revision f for the new JDK 1.2. There are two 1.0f versions around, one which works with both Java 1.1 and Java 2 and one which has some trouble with Java 2.

<table>
<thead>
<tr>
<th>Class</th>
<th>example methods</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFun</td>
<td>public static double sinh(double x)</td>
<td>Special Functions</td>
</tr>
<tr>
<td></td>
<td>public static double gamma(double x)</td>
<td>hyperbolic sine function</td>
</tr>
<tr>
<td></td>
<td>public static double erf(double x)</td>
<td>the gamma function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the error function</td>
</tr>
<tr>
<td>Statistics</td>
<td>public static double average(double[] x)</td>
<td>Statistical functions</td>
</tr>
<tr>
<td></td>
<td>public static double normalCDF(double x)</td>
<td>the sample mean</td>
</tr>
<tr>
<td></td>
<td>public static double[] linearFit(double[] x, double[] y)</td>
<td>cummulative normal distribution function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>linear least square estimator</td>
</tr>
<tr>
<td>Complex</td>
<td></td>
<td>for handling complex numbers</td>
</tr>
<tr>
<td>DoubleVector</td>
<td></td>
<td>using vector operations on arrays of type double</td>
</tr>
<tr>
<td>ComplexVector</td>
<td></td>
<td>the same for complex vectors</td>
</tr>
<tr>
<td>DoubleMatrix</td>
<td></td>
<td>using matrix operations on 2D arrays of type double</td>
</tr>
<tr>
<td>ComplexMatrix</td>
<td></td>
<td>the same for complex matrices</td>
</tr>
</tbody>
</table>
Eight more classes (four for double and four for complex) for matrices are provided, which perform matrix decompositions, like Cholesky, LU, QR and SVD decompositions. They can be used for an efficient solution to linear systems of equations.

For a complete list of supplied functions, please consult the online documentation coming with the JNL package.

**Complex numbers** This part of the JNL deserves a separate discussion. We will not need complex numbers in most parts of the book, but for example when we discuss quantum mechanical problems, we have to come back to treating complex numbers in Java.

Because complex numbers are not a primitive data type in Java (yet), and there is no operator overloading like in C++ for example, we cannot write \( a+b \), if \( a \) and \( b \) are complex numbers. We have to create a new object called Complex and define methods to work with these objects. This is exactly the solution followed by the JNL, so we do not need to do it again.

To instantiate complex numbers issue

```java
import VisualNumerics.math.*;
........
Complex c1 = new Complex(1,2); // means c1 = 1+2i
Complex c2 = new Complex(2); // means c2 = 2
System.out.println(c1); // gives 1+2i on screen
........
```

By the way you can change the output character for the complex unit \( i \) by using `Complex.suffix="j";` for example.

To add two complex numbers use one of the two possible methods:

```java
Complex cresult1 = Complex.add(c1,c2);
Complex cresult2 = c1.add(c2);
```

The first version is the static method and the second one is the instance method of the class Complex. There are analogous static methods for `Complex.subtract(c1,c2), Complex.divide(c1,c2)` and `Complex.multiply(c1,c2)` and of course the same with instance methods.

You can extract real and imaginary part from a complex number easily by using

```java
Complex c1 = new Complex(2,5);
System.out.println(" Real part: "+c1.re);
System.out.println(" Imaginary part: "+c1.im);
```

An overview of the most important methods is given in Table 1.6.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex.abs(z)</td>
<td>absolute value</td>
</tr>
<tr>
<td>Complex.argument(z)</td>
<td>the argument of the complex number z</td>
</tr>
<tr>
<td>Complex.conjugate(z)</td>
<td>conjugate of z</td>
</tr>
<tr>
<td>Complex.sqrt(z)</td>
<td>the square root of z</td>
</tr>
<tr>
<td>Complex.pow(z1,z2)</td>
<td>the power of z1 to the z2</td>
</tr>
<tr>
<td>Complex.pow(z,d)</td>
<td>the power of z to the (double) d</td>
</tr>
<tr>
<td>Complex.exp(z)</td>
<td>the natural exponential of z</td>
</tr>
<tr>
<td>Complex.sin(z)</td>
<td>the sine of z</td>
</tr>
</tbody>
</table>

### 1.13.3 The JSci/JavaSci Package

The JavaSci package has been developed at the *Imperial College of Science, Technology and Medicine* in the UK. It is a freely available scientific package and its objective is:
“JavaSci aims to encapsulate scientific principles and ideas in a way that is representative of their underlying structure and usage.”

The package consists of basically two types of classes: representation classes and library classes. The representation classes are instanitateable and represent some physical or mathematical structure, whereas the library classes are not to be instantiated and are used to manipulate the representation classes.

The JSci package is split into a chemistry, a physics and a maths subclass. Then there are many convenience classes and some interesting other classes. The best way to learn the package is to study the API documentation. In Table 1.7 we give a short overview of interesting classes and methods in the JavaSci package used throughout the book.
### Table 1.7 Some of the interesting classes and methods in the JSci/JavaSci package.

<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSci.chemistry.Atom</td>
<td>Used to construct and store an atom with all the data necessary.</td>
</tr>
<tr>
<td>JSci.chemistry.Element</td>
<td>Used to construct and store an element with all the data necessary.</td>
</tr>
<tr>
<td>JSci.chemistry.Molecule</td>
<td>Used to construct and store a molecule with all the data necessary.</td>
</tr>
<tr>
<td>JSci.chemistry.periodictable.TransitionMetal</td>
<td>The full periodic table with all data.</td>
</tr>
<tr>
<td>JSci.maths.Mapping</td>
<td>An interface to define 1D complex or real functions.</td>
</tr>
<tr>
<td>JSci.maths.NMapping</td>
<td>An interface to define N dimensional complex or real functions.</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath</td>
<td>All kinds of array methods to make life easier. Many more as in the Java 2 array class.</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.correlation()</td>
<td>Compute the (linear) correlation between two arrays.</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.norm()</td>
<td>Compute the L2 norm of an array (Euclidean norm or length).</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.normalize()</td>
<td>Normalize the array so that its L2 norm is 1 (up to computational errors).</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.print()</td>
<td>Print a 1D or 2D array to the screen.</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.sumSquares()</td>
<td>Sum the squares of all components; also called the energy of the array.</td>
</tr>
<tr>
<td>JSci.maths.ArrayMath.toString()</td>
<td>Returns a comma delimited string representing the value of the array.</td>
</tr>
<tr>
<td>JSci.maths.FourierMath</td>
<td>Methods for 1D real or complex FFTs.</td>
</tr>
<tr>
<td>JSci.maths.FourierMath.transform()</td>
<td>A 1D complex or double fast Fourier transformation.</td>
</tr>
<tr>
<td>JSci.maths.LinearMath</td>
<td>Eigenvalues, eigenvectors, linear least squares and solving linear systems.</td>
</tr>
<tr>
<td>JSci.maths.NumericalMath</td>
<td>Numerical differentiation, integration, ODEs, Metropolis algorithm</td>
</tr>
<tr>
<td>JSci.maths.NumericalMath.rungeKutta()</td>
<td>Uses the Runge-Kutta method to solve an ODE.</td>
</tr>
<tr>
<td>JSci.maths.WaveletMath</td>
<td>All kinds of methods for wavelets.</td>
</tr>
<tr>
<td>JSci.maths.statistics</td>
<td>A class with many important complicated distributions (Beta, Gamma, T, ChiSquared, etc.)</td>
</tr>
<tr>
<td>JSci.physics.PhysicalConstants</td>
<td>All physical constants you can think of...</td>
</tr>
<tr>
<td>JSci.physics.quantum.DensityMatrix</td>
<td>Calculate the density matrix.</td>
</tr>
<tr>
<td>JSci.util.MatrixToolkit</td>
<td>Methods to create random matrices and some conversions.</td>
</tr>
<tr>
<td>JSci.io</td>
<td>Reading/Writing text files and writing MathML (XML) files.</td>
</tr>
<tr>
<td>JSci.awt</td>
<td>Graphical routines like plotting for the AWT.</td>
</tr>
<tr>
<td>JSci.awt.BarGraph()</td>
<td>A bar graph AWT component.</td>
</tr>
<tr>
<td>JSci.awt.ContourPlot()</td>
<td>A contour plot AWT component.</td>
</tr>
<tr>
<td>JSci.awt.LineGraph()</td>
<td>A 2D line graph AWT component.</td>
</tr>
<tr>
<td>JSci.awt.LineGraph3D()</td>
<td>A 3D line graph AWT component.</td>
</tr>
<tr>
<td>JSci.awt.PieChart()</td>
<td>A pie chart AWT component.</td>
</tr>
<tr>
<td>JSci.awt.ScatterGraph()</td>
<td>A scatterGraph AWT component.</td>
</tr>
<tr>
<td>JSci.swing</td>
<td>The same for the swing package.</td>
</tr>
</tbody>
</table>
The \texttt{JSci.maths.ArrayMath} class also contains methods for calculating the mean, variance, standard deviation, etc. like the JNL. The \texttt{JSci.maths} package contains many methods and classes for handling matrices either integer, double, complex, sparse or diagonal and more. It has methods for handling vectors in all flavours. And it even provides functionality for discrete sets and the construction of the Karhunen-Loeve expansion. There are complex numbers, special functions like hyperbolic ones and in the \texttt{SpecialMath} math class there are even more like the Bessel functions, Airy functions, incomplete Gamma and Beta functions, etc. Another class can handle symmetry groups, one handles chaotic maps and almost everything you wish about wavelets in many different classes.

In the \texttt{JSci.physics} class you get methods and classes to use particles. There is a whole class just defining all kinds of particles known to physicists today. Then you can use the \texttt{JSci.physics.quantum} class to do quantum mechanics calculations easily or you could use the \texttt{JSci.physics.relativity} class to do relativistic calculations.

Since version 0.82 there is the IBM MathML package included to write XML/MathML files from Java. That is the reason why now there is a limitation in the use of the whole package. So go ahead and read the license agreement in the online documentation or at the homepage of JSci.

In our opinion, using the JNL is preferred. Only for methods not contained in this package use the JSci package like for FFTs or for solving ODEs for example. JSci is still in the development stage and it takes some more time to be the package of choice for most of the tasks.

### 1.13.4 JNT, Lapack for Java - JamPack and Jama

JNT

Lapack

### 1.14 Debugging in Java

### 1.15 Advanced Java Features

In this chapter we shortly discuss and present some advanced features, which are of interest to some scientists, but are not essential for finishing this book. But nevertheless some topics deserve to be mentioned for self study. Still we are missing Java features such as graphical interfaces, plotting, parallel execution, file reading and writing and more. These are the contents of the rest of the book and are always contained in the context of a new simulation technique.

**Including Native Code (C and C++)**

To include native code into a Java program, there is the so-called JNI (Java Native Interface). This allows for inclusion of native C or C++ codes. To refer to such a method, you just use

```java
... native CFunction();
```

The code for the method has to be implemented in C or C++.

At the moment there is no direct interface to other languages like Fortran. If you want to include Fortran code you have to use a C wrapper code and then use the JNI with that C wrapper code for inclusion to Java.
Vectors

For a scientist there is a confusing class called `java.util.Vectors`, which has nothing to do with mathematical vectors. This class realizes an array of objects, which can have a variable length. For example you can store names of persons in an vector and add or delete them as you like without taking care of the number and the access.

An interesting usage for this class is an array, which does not have a fixed size. Using the Vector class you can easily build a class handling a flexible size array. For details of the class please refer to the API documentation.

Stack

There is also a class `java.util.Stack`, which utilizes as the name suggests a stack. A stack is a kind of pile, where you can put things on it and take them away, but only the last one put in. It is for example used for many operating system tasks on a lower level. Again take a look at the API documentation for details.

Hashtable

Another often occurring class of the `java.util` package is the hashtable class. It realizes a table with associations between so-called keys and values, which has to be a one-to-one correspondence. So for example you can store the name of a person and its age in a hashtable.

Serialization

One problem of objects is how to “save” objects to disk for example. Or sometimes you want to submit an object to another applet or application using RMI across networks, or you might want to store the object for the next call of the same program (called persistence).

For all these reasons and many more, there is a method called serialization, which transforms an object in a byte stream and back. To save an object to a file for later use, you could easily serialize the object and store the byte stream to a file. Later on you reread the file and deserialize the byte stream back to an object.

If you create your own classes and want objects (instances of your class) to be serializable you have to implement the serializable interface. We do not need this in the context of this book, but it is important to know what is behind this.

Beans

Beans are classes, which conform to certain naming rules for the methods contained in a class. A bean gets a graphical representation (an icon) and can be used in a separate program (often called the beanbox) to create an application by just putting together different beans and changing the resources of the bean.

For example we can use a plot bean, which displays a plot (see ptplot later on). Then you can place this plot bean on the screen and change the title, axes, etc. Then you can add a bean, which creates random numbers and type in the size of the set. Then you can connect the two beans, which automatically sends the data to the plot bean and plots it. This is called visual programming and is used in programs as IBM Data Explorer or AVS/Express to write programs mostly including visual interfaces (GUIs).

You can for example write your own graphical beans and import them into one of the IDEs like Simplicity or Netbeans. Then you can use your own bean to construct a nice GUI.

1.16 Online References

http://java.sun.com/ is the homepage of Java from SUN. A good starting point.

http://www.javasoft.com/

???????
1.17 Exercises

Exercise 1.1 Computation of Pi using Hit and Miss Algorithm

Write a Java program to estimate the number $\pi = 3.1415...$ using the following algorithm:

- Choose two random numbers $r_1$ and $r_2$ between 0 and 1. The two random numbers define a point in a 2D square of length 1.

- Check if the distance of the random point is inside a circle of radius 1. e.g. check if $r_1^2 + r_2^2 < 1$. Count these points in a variable called for example inside.

- Repeat this until you have created and checked $N$ points.

- Compute $4 \times \text{inside} / N$, which is an approximation of $\pi$.

This is the algorithm for the “Hit and Miss” algorithm, which we will discuss in a later chapter in detail.

Run this program with several different values of $N$ and plot the accuracy, the absolute difference between the exact $\pi$ and the estimation, versus the number of random points used.

Exercise 1.2 Compute Euler Constant $e$ using a Darts Board
Bibliography


Chapter 2

Plotting with Java

Among the numerical techniques available to computational physics, stochastic methods, also called Monte Carlo methods, play a central role. They are particularly appealing because of their immediacy, their power and the breadth of applications. Of course, in order to profit from the advantages of such techniques it is necessary to visualize the results of the simulations. It is the aim of this chapter to introduce into the graphical analysis of data with Java.

Our approach will be hands on. We will start by presenting the prototype of a stochastic simulation algorithm: the simulation of radioactive decay. As we already know, Monte Carlo methods rely upon the use of random numbers. The radioactive decay is a physical system in which random events arise naturally. Further on, we will learn that the sampling of random numbers offers an efficient numerical method to compute multidimensional integrals.

The main part of the chapter is devoted to the graphical analysis of the data of the simulation of radioactive decay. We will begin by introducing the foundations of graphical tools of Java. Then, we proceed with the presentation of some more comfortable graphical capabilities which are freely available on the Internet and with the discussion of some more advanced topics. All the tools will be applied to the simulation of the radioactive decay.

2.1 The Radioactive Decay

As an example of a natural stochastic process we consider radioactive decay. Many heavy nuclei are intrinsically unstable and decay to lighter, more stable elements under the emission of $\alpha$-, $\beta$-, or $\gamma$-radiation. The radioactive decay is a statistical process. One cannot foresee at what time the next nucleus will decay. According to the radioactive decay law one can predict the mean number of nuclei, which will decay in a given time interval. Let us denote by $\lambda$ the decay constant, i.e., the fraction of given nuclei decaying per second. Then the average number of decays occurring between time $t$ and time $t + dt$ is given by the relation

$$dn = -\lambda ndt.$$  \hspace{1cm} (2.1)

The quantity $dn/dt$ is called the activity. Its dimension is the Becquerel (1Bq = $1\text{s}^{-1}$). To give an example, the activity of 1g of Radium $^{226}\text{Ra}$ is approximately equal to $3.7 \cdot 10^{10}$ Bq (In an older notation the same activity was named 1 Curie). $\tau = 1/\lambda$ is the mean life time, during which the number of radioactive nuclei drops to $1/e$.

If at time $t = 0$ we have $n_0$ nuclei, it follows from Eq. (2.1) that at the later time $t > 0$ we are left with

$$n(t) = n_0 \exp(-\lambda t)$$  \hspace{1cm} (2.2)

nuclei. The half–life $t_{1/2}$ is easily evaluated from the condition

$$n(t_{1/2}) = \frac{n_0}{2}$$  \hspace{1cm} (2.3)
to be
\[ t_{1/2} = \frac{\ln 2}{\lambda}. \]  
(2.4)

It is important to remark that for each nucleus regardless of the decay mode \( \lambda \) and \( \tau \) are characteristic constants which do not depend upon, e.g., the temperature, the pressure, or chemical reactions.

Let us now turn our attention to the stochastic description of radioactive decay from which we will derive a stochastic algorithm. As we have already noticed a basic ingredient of all Monte Carlo recipes is the use of random numbers. Thus, we have to know how to draw random numbers in our computer program. At the moment it is not important for us to understand how this works using a computer. We only have to know how to access them for now and this will be the subject of Chapter 2.1.1.

We only have to know that almost all programming languages have a random number generator in form of some function in their mathematical library. Shortly we will see how this can be achieved in Java.

Let us assume that the system is made of \( N_0 \) unstable nuclei. The probability \( p \) for a nucleus to decay in the finite time interval \( \Delta t \) is obviously given by

\[ p = \lambda \Delta t \quad (\text{for} \quad \lambda \Delta t \ll 1), \]  
(2.5)

Therefore it is easy to decide whether a nucleus decays with probability \( p \) or not. To do so we have to draw a random number \( R \) uniformly distributed in the interval \([0, 1)\). This random number lies with probability \( p \) in the interval \([0, p\Delta t]\). Therefore, if \( R \leq p\Delta t \) a decay takes place, otherwise it does not. Hence in each time step \( \Delta t \) we have to decide between two cases: a) If a decay takes place we put \( N \rightarrow N-1 \) and \( t \rightarrow t + \Delta t \); b) If no decay takes place we set simply \( t \rightarrow t + \Delta t \).

Thus, schematically the stochastic algorithm to simulate the radioactive decay reads

For \( t=0 \) to \( t \) with step \( \Delta t \)
- For each remaining nucleus
  - Decide if the nucleus decays
    - if (random number < \( p \Delta t \)) then
      - \( N \rightarrow N-1 \)
    - end
  - end loop over nuclei
- end loop over time

Before writing a program in Java to simulate the radioactive decay, let us briefly discuss the generation of random numbers in Java.

### 2.1.1 Random Numbers in Java

In the first two chapters we have already met different possible ways to generate random numbers in Java. The first method \( \text{random()} \) we encountered was contained in the \texttt{Math} class of the \texttt{java.lang} package. The method \( \text{random()} \) creates a single Random object the first time it is invoked and returns pseudo-random numbers for that object for each subsequent call. A better way to generate random numbers is provided by the \texttt{java.util} package, which contains several standard utilities interfaces and classes. This second possibility is to be preferred since it offers more possibilities to control the generation of random numbers. With the help of the constructor

\[
\text{public Random()}
\]

we can create a new Random object. As we will learn in the next Chapter the sequence of random numbers begins with the so-called seed. The class Random automatically chooses a seed according to the current time. If a specific seed is desired, this can be fixed with the help of the constructor

\[
\text{public Random(long seed)}
\]

Furthermore, the method \( \text{public synchronized void setSeed(long seed)} \) which can be invoked at any time resets the sequence of random numbers to start from the given seed. Having instantiated the Random object a pseudo-random number uniformly distributed between 0.0 (inclusive) and 1.0 (exclusive) is returned by invoking the instance method
public double nextDouble()

Similarly, the method nextFloat may be invoked to generate uniformly distributed random numbers of the float type. In later Chapters we will also need uniformly distributed integer valued random numbers. Such pseudo-random numbers between Integer.MIN_VALUE and Integer.MAX_VALUE can be generated in Java with the help of

public long nextInt()

Alternatively it is possible to invoke also the method nextLong(), which generates discrete random numbers uniformly distributed between Long.MIN_VALUE and Long.MAX_VALUE. The simulation package contains two methods to generate an integer valued random number between 0 and N: the nextInteger() or nextInt() method of the Distribution class. In Java 2 there is already a method nextInt() to produce an integer valued random number between 0 and N. So we can always use the nextInt() method in Java 1.1 and 1.2. To switch from Java 1.1 to 1.2, we just have to change the creation of the random number generator object from the simulation class to the java.util.Random class. For later purposes, let us mention that the package java.util also contains a method

public synchronized double nextGaussian()

which returns a pseudo-random Gaussian–distributed double value with mean 0.0 and standard deviation 1.0.

Now we are in the position to write a Java code for the simulation of the radioactive decay.

2.1.2 The Simulation Code

The above algorithm written in Java is shown in the following listing.

```java
import java.applet.Applet;
/** A simple Radioactive decay simulation
 with output to the command line */
public class RadioactiveDecay extends Applet {
    private int N=1000;
    private double t_end=300;
    private double decay_const = 0.02;
    private double dt = 1;
    private double prob = decay_const * dt;

    public int [] N_simu, N_exact;
    public java.util.Random rand = new java.util.Random();

    // empty constructor */
    void RadioactiveDecay () {} } }

/** use a main method to use it as application OR applet */
public static void main(String [] args) {
    RadioactiveDecay decay = new RadioactiveDecay();
    decay.init();
}

/** The actual program */
public void init () {
    int steps , N_save , N;
    double jump;

    steps = (int)(t_end / dt) + 1;
    N_simu = new int [steps];
    N_exact = new int [steps];
    N_simu [0] = N0;
```

The program is straightforward. In line 5 to 10 the relevant variables are defined and initialized. The main time loop starts is line 35 and the loop over all the nuclei is starts in line 36. In line 38 we check if the nuclei is already decayed or not. In order to check the results we evaluate the exact solution in line 41. Finally, we print the result of the simulation to the screen comparing the exact with the simulated solution.

The class RadioactiveDecay has been written in such a way that it can be run as an application as well as an applet. The idea is to extend the Applet class and to define, as it is necessary for applications, a main method. The actual algorithm is implemented in lines 33 to 41.

We are now in the position to perform a simulation. To this end we run the program with the following parameters

\[ N_0 = 1000; \quad \lambda = 0.02 s^{-1}; \quad \Delta t = 1s; \quad t_{\text{end}} = 300s. \]

Running the program it is evident that the screen output is not particularly satisfactory to examine the results of the simulation. Thus, we have reached the point where we feel the necessity to learn something about the graphical possibilities of Java.

## 2.2 The Most Easy Plot – The AWT and Applet Packages

Before trying to plot the data generated with the program RadioactiveDecay we want to discuss the most simple Java code which allows to plot some 2 dimensional data. With the help of this example we will learn the foundations of the graphical tools of Java. The code we want to discuss is the PlotEasy class which can be run as an applet or as an application.

```java
/*<applet code="PlotEasy.class" width=400 height=400> */
import java.applet.Applet;
import java.awt.*;

/** The easiest way to plot a Chart in Java,
 use it as Applet or as an application.
 There is no way to stop the program or to close the window,
 you have to kill the program. */
10 public class PlotEasy extends Applet {
   public static void main(String [] args) {
      Applet a = new PlotEasy();
      Frame f = new Frame("Easy_Plotting");
      f.add("Center",a);
      f.setSize(400,400);
      f.show();
```
2.2. THE MOST EASY PLOT – THE AWT AND APPLET PACKAGES

    a.init();
    }
    
    /* Here we define the function to be plotted */
    double f(double x) {
        return (Math.cos(x/2)+Math.sin(x/7)+2)*getSize().height/4;
    }
    
    /* Here we plot the function because we are using the actual size of the canvas, the plot gets resized each time you resize the window */
    public void paint(Graphics g) {
        for (int x=0; x<getSize().width; x++) {
            g.drawLine(x,(int)f(x),x+1,(int)f(x+1));
        }
    }
    }

As we can see the code starts by importing packages. We already met the first one, the java.applet.Applet package in the applet version of the HelloWorld program. The java.applet package is a small package. It simply contains the Applet class, which is the superclass of all applets, i.e., in order to create our own applet we have to create a subclass of this class and override (overload) some or all of its methods (see chapter 0). The second package we have to import is the java.awt package, where awt stands for the Abstract Windowing Toolkit. This is one of the biggest packages in Java 1.1 and includes all the nice graphics capabilities of Java.\footnote{The discussion is based on Java 1.1, but if we just prepend a capital J to all AWT class names, we can use the new Swing/Java 2 components. We also have to change the import command from import java.awt.*; to import javax.swing.*; You should not use both together in one program.}

Java programs look different on different systems, because they use the AWT, which is an abstract class, just defining the necessary methods and fields to write programs using graphics functionality. To accomplish this, Java uses a so-called peer architecture, see Figure 2.1.

![Figure 2.1](image)

Figure 2.1: The peer architecture of the AWT in Java.

In order to understand the deep relation between applets and the AWT it is instructive to look at Figure 2.2, which shows the inheritance hierarchies of the Applet class. In the Java AWT API only interfaces and abstract methods are defined, the actual program code has to be implemented for each platform in native code. The communication between the native and the AWT objects is realized using the peer interface. We as programmers only have to understand how to use the AWT and do not have to care about the underlying peer interface or even the native implementations. Someone who wants to write a virtual machine for Java has to understand the peer architecture and implement all necessary native code for the AWT.
Applets inherit the drawing and event handling methods from the AWT Components class. Component is the superclass of all GUI components in the java.awt package. Many important methods you have to use are defined in the Component class.

The java.awt.Container class implements a component that contains other components. Container can not be instantiated directly. You always have to use one of its subclasses, such as Panel, Frame or Dialog. Once a Container is created you can set its Layout Manager with setLayout() or add components to it with add(). You can remove components with remove().

The java.awt.Panel class is a Container that is itself contained in a Container. It does not create a separate window of its own. Applets are a subclass of Panel that is contained within a Web browser or an applet viewer. Figure 2.2 shows the hierarchy relations of some (not all!) components and layouts classes of the java.awt package.

An important class of the AWT package is the java.awt.Graphics class, which encapsulates most of the graphics functionality of the Java API. But the Graphics class is an abstract class and does not have a constructor (For abstract classes, see Chapter 1.5). So there is no way to instantiate a Graphics object like new Graphics() (This syntax is WRONG! We can not use it!). To get a Graphics object we can ask for one by calling the getGraphics() method of the Component class. This is only possible, if the actual Component class does have a drawable graphic context, e.g. Canvas or Container objects. It makes obviously no sense to have a drawing area for a Button object for example. More common and most of the time easier is to override the paint() or update() methods of a Component object, because both methods supply the Graphics object as an parameter to the method.

Now we have the necessary background to understand what is going on in the PlotEasy.java program. The class PlotEasy extends the Applet class and has a main method. In the main method we instantiate the Applet PlotEasy and the Frame f. The hierarchy of the Frame class is (see Fig. 2.2) Object → Component → Container → Window → Frame. The Frame class represents an optionally resizable application window with a title bar.

With the help of the add(Component) method in the Container class, we can "add" arbitrary components to our container. E.g. we can add a button or a label to the frame we have created, for example:

Frame f = new Frame("Test Frame");
Button but = new Button("Test");
....
f.add(but); // here we add the Button to the Frame
Figure 2.3: The hierarchy of the AWT package.

```plaintext
java.lang.
object

Graphics
Component

FlowLayout
GridLayout
BorderLayout

Label
Button
Container
Canvas
Panel
ScrollPane
Window
Panel
ScrollPane
Window
Dialog
Frame
f.show(); // finally we show the Frame on the screen

The resulting window is shown in figure 2.4.

![Figure 2.4: The output of the most easy window with a button using the AWT.](image)

A list of all AWT components defined in the Java 1.1 and 1.2 standard is shown in table 2.1.

The arrangement of the components, in Java called layout, and the containers for each container can be defined by using a layout manager. By default it always uses the FlowLayout manager for an application, which just puts the components beside each other in a line and if the window is too small it wraps them to the next line. For an applet the default is the borderlayout manager.

More sophisticated layout managers are available in Java. All possible layout managers are given in Table 2.2. Unfortunately it is a pretty difficult task to use some of the layout managers, but if the desired layout for a program is not too complicated, it is enough to use the easy borderlayout, flowlayout and/or gridlayout managers. If we need more complex layouts, we should use the Swing/JFC packages, which are also available for Java 1.1 as mentioned in the introductory chapter. If we use an IDE like Simplicity for Java or Netbeans, it is much easier to set up our own layouts and windows.

Let us look back at the PlotEasy.java program. With the help of the method setSize(int width, int height) we fix the dimensions of the frame in pixels. Another useful method is the pack() method of the Frame class, which sets the size of the frame to the size necessary to display all contained components of the frame. The show() method displays the frame. As we already know a.init() calls the init() method of the Applet class and therefore starts the applet. There is also a method called setResizable(boolean) for the Frame class, which decides if the window can be resized by the user or not.

In principle the part of the code we just described is not necessary. We included it only to allow the program to be run as an application and as an applet. Try to run the code without the lines 8 to 15 as an applet and it will still work, but it will no longer work as an application, because the main method is missing.

The actual plotting is performed by the paint() method, which draws on the screen – to be precise: in the panel of the Applet class. Applets typically override some of the methods of the Component class of the java.awt package and you have to override at least the init() method, as we have already learned. The simplest way for a Component to draw itself is to put drawing code in the paint() method. In lines 24 to 27 we see a simple example of implementing the paint() method.

The Graphics class of the java.awt package defines methods for drawing different kinds of shapes. The method which we use here is the
drawLine(int x1, int y1, int x2, int y2)
method, which simply draws a line with the current color in the Graphics object g. Other typical methods are shown in table 2.3. There is no drawPoint() method as one might expect, but we can easily use the drawLine() method with same starting and endpoint of course. Much more graphics capabilities have been added by introducing the Java2D package into Java 2.

The method getSize().height and getSize().width, which we use in lines 19 and 25 return the height and the width of the Frame or the panel of the applet. Now that we understand the code let us run the program, either from the command line or using the appletviewer or netscape.

If we try to resize the window, we realize that the plot is drawn again and zoomed to the new window size. This is because we have not overloaded the update() method and the default behaviour of the update() method is to call the paint() method again. In chapter 0 we have learned, that
2.2. THE MOST EASY PLOT – THE AWT AND APPLET PACKAGES

Table 2.1 A list of most of the defined AWT components of the Java 1.1 API and the Java2 API (Swing).

<table>
<thead>
<tr>
<th>Component</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>A button with text on it.</td>
</tr>
<tr>
<td>Canvas</td>
<td>A drawing area.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>A list of clickable choices (more than one can be chosen at a time).</td>
</tr>
<tr>
<td>CheckboxGroup</td>
<td>A list of clickable choices (only one can be chosen at a time).</td>
</tr>
<tr>
<td>Choice</td>
<td>A list of choices, like a pull-down menu.</td>
</tr>
<tr>
<td>Label</td>
<td>A short text to be displayed in the GUI.</td>
</tr>
<tr>
<td>Menu</td>
<td>Many components are available to realize menus.</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>Sometimes called a slider.</td>
</tr>
<tr>
<td>TextComponent.TextArea</td>
<td>Multiple lines of text to be displayed or optionally editable.</td>
</tr>
<tr>
<td>TextComponent.TextField</td>
<td>A line of text, optionally editable.</td>
</tr>
<tr>
<td>Container.Panel</td>
<td>A “box” to contain other components.</td>
</tr>
<tr>
<td>Container.ScrollPane</td>
<td>A container to hold large areas, which cannot be displayed and have to be scrolled using the scroll panes.</td>
</tr>
<tr>
<td>Container.Window.Frame</td>
<td>An optionally resizable window with title and decorations.</td>
</tr>
<tr>
<td>Container.Window.Dialog</td>
<td>A dialog window to ask a question or display a notification.</td>
</tr>
<tr>
<td>JRadioButton</td>
<td>A special Button (only one can be selected).</td>
</tr>
<tr>
<td>JComboBox</td>
<td>Special purpose component to accomplish long lists.</td>
</tr>
<tr>
<td>JList</td>
<td>A group of items displayed in a column.</td>
</tr>
<tr>
<td>JSlider</td>
<td>For entering numeric values, which are bounded.</td>
</tr>
<tr>
<td>JProgressBar</td>
<td>Visible component to show how much of a job has been completed.</td>
</tr>
<tr>
<td>JColorChooser</td>
<td>Choose from a color box.</td>
</tr>
<tr>
<td>JFileChooser</td>
<td>Get a file or path from the user.</td>
</tr>
<tr>
<td>JTable</td>
<td>Display a table of data.</td>
</tr>
<tr>
<td>JTree</td>
<td>Display hierarchical data.</td>
</tr>
<tr>
<td>ToolTips</td>
<td>For every component you can have a balloon help/tool tip.</td>
</tr>
<tr>
<td>Container.JToolBar</td>
<td>Group several components (eg. Buttons) with icons.</td>
</tr>
<tr>
<td>Container.JSplitPane</td>
<td>Two panels in one.</td>
</tr>
<tr>
<td>Container.JTabbedPane</td>
<td>Sharing the same place by many panels, similar to carlayout.</td>
</tr>
</tbody>
</table>

Table 2.2 All possible layouts in the AWT package of Java 1.1.

<table>
<thead>
<tr>
<th>Layout Manager</th>
<th>how it works:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowLayout</td>
<td>The default layout, everything beside each other.</td>
</tr>
<tr>
<td>GridLayout</td>
<td>Put all components/container in a table structure.</td>
</tr>
<tr>
<td>BorderLayout</td>
<td>Use a center and 4 borders to put the components/containers.</td>
</tr>
<tr>
<td>CardLayout</td>
<td>Put components/containers like cards on a stack.</td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>A very versatile but complicated layout manager.</td>
</tr>
<tr>
<td>BoxLayout</td>
<td>Components in a column, where the largest gives the width.</td>
</tr>
<tr>
<td>“Absolute Positions”</td>
<td>New in Java 2, but it is better to use the others.</td>
</tr>
</tbody>
</table>
Table 2.3 List of some of the methods contained in the Graphics class. All method arguments are of type integer, unless otherwise stated. More methods are available in the much more powerful Java2D API coming with Java2.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawLine(x1, y1, x2, y2)</td>
<td>Draw a line or point.</td>
</tr>
<tr>
<td>drawRect(x, y, width, height)</td>
<td>Draw a rectangle.</td>
</tr>
<tr>
<td>fillRect(x, y, width, height)</td>
<td>Draw a filled rectangle.</td>
</tr>
<tr>
<td>clearRect(x, y, width, height)</td>
<td>Clear a rectangle area.</td>
</tr>
<tr>
<td>drawArc(x,y,width,height,Angle0,arc)</td>
<td>Draw a part of a circle.</td>
</tr>
<tr>
<td>fillArc(x,y,width,height,Angle0,arc)</td>
<td>Draw and fill the arc.</td>
</tr>
<tr>
<td>drawPolygon(int[] x, int[] y, nPoints)</td>
<td>Draw a polygon with the given points.</td>
</tr>
<tr>
<td>fillPolygon(int[] x, int[] y, nPoint)</td>
<td>Same, but fills it with a color.</td>
</tr>
<tr>
<td>copyArea(x, y, width, height, dx, dy)</td>
<td>Copy an area by dx/dy.</td>
</tr>
<tr>
<td>drawString(String text, x,y)</td>
<td>Draw text at the position.</td>
</tr>
<tr>
<td>translate(x,y)</td>
<td>Translate the origin.</td>
</tr>
</tbody>
</table>

Component.repaint() / Component.repaint(long milliseconds)  call this

Component.update(Graphics g)  override these

Component.paint(Graphics g)

Figure 2.5: The default behaviour of the painting methods of components in the AWT package.
2.2. THE MOST EASY PLOT – THE AWT AND APPLET PACKAGES

the `update()` method is called everytime the window has to be redrawn, because of some events like
scrolling in the browser window or, like here, resizing the window (see Figure 2.5.).

Having learned the basic graphical tools of the AWT with the help of the most easy plot program, we
can now apply what we have just learned to the simulation of the radioactive decay.

```java
/* <applet code="RadioactiveDecay_easyplot.class" width=500 height=400> Run Applet </applet> */
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/** This is a Radioactive decay simulation using only Java 1.1 features. */
public class RadioactiveDecay_easyplot extends Applet {
    private static int width=500, height=400;
    private int N_0=2000;
    private double t_end=400;
    private double decay_const=0.02;
    private double dt=1;
    private double prob=decay_const*dt;
    public int [] N_simu , N_exact ;

    /** The main method just calls the init() method of the applet and
     opens a window to host the plots */
    public static void main(String [] args ) {
        Applet applet = new RadioactiveDecay_easyplot ( );
        Frame frame = new Frame("Radioactive_Decay");
        frame.addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent e ) {
                System.exit(0); }
        }); // Handle window close requests
        frame.setSize( width , height ); // set size of window
        frame.add("Center", applet ); // add applet to the window
        frame.show(); // display window on screen
        applet.init(); // start applet
    }

    /** The actual program, started by the browser or by the main method */
    public void init() {
        int steps , N_save , N;
        double jump;

        steps = (int)(t_end/dt)+1;
        N_simu = new int [steps ];
        N_exact = new int [steps ];
        N_simu[0] = N_0 ;
        N_exact[0] = N_0 ;

        // Here the actual simulation takes place
        N_save = N_0 ;
        N=N_0 ;
        for ( int t=0; t<t_end; t++) {
            for ( int i=0; i<N_save; i++) {
                jump=Math.random();
                if ( jump < prob ) N--; }
            N_save=N;
            N_simu[t+1]=N;
            N_exact[t+1]=(int)(N_0*Math.exp(-decay_const*t));
        }
    }
}
```
Here we plot the points and repaint it, if the window gets resized. */
public void paint(Graphics g) {
    int x1, x2, y1, y2;
    double scale_x, scale_y;

    // get the size of the available canvas
    int width = this.getSize().width;
    int height = this.getSize().height;

    // Calculate scaling factors to use the whole plotting area
    scale_x = (double) width / t_end;
    scale_y = (double) height / N_0;

    // Plot all the points without axes -- scale it
    for (int t = 1; t < t_end - 1; t++) {
        // simulation
        y1 = (int) (height - N_simu[t - 1] * scale_y);
        y2 = (int) (height - N_simu[t] * scale_y);
        x1 = (int) ((t - 1) * scale_x);
        x2 = (int) (t * scale_x);
        g.setColor(Color.red);
        g.drawLine(x1, y1, x2, y2);

        // exact result
        y1 = (int) (height - (double) N_exact[t - 1] / N_0 * height);
        y2 = (int) (height - (double) N_exact[t] / N_0 * height);
        g.setColor(Color.black);
        g.drawLine(x1, y1, x2, y2);
    }
}

Implementing the graphical facilities in the RadioactiveDecay.java code is easy. The output of the new program can be seen in figure 2.6.

Figure 2.6: The output of the easypplot version of the radioactive decay program. The exact solution is a black line, the simulation a red line.

In the program code we immediately recognize in the lines 56 to 81 the paint() method. The results
of the simulation have to be scaled appropriately to fit in the Frame. The curve is plotted with the method 
\texttt{drawLine()}. Since we want to plot also the exact analytical solution for the mean values in red we set
\begin{verbatim}
g.setColor(Color.red)
\end{verbatim}
in line 74 before drawing the corresponding curve.
It is important to note that in the lines 21 to 24 we have added the code
\begin{verbatim}
frame.addWindowListener(...);
\end{verbatim}
which allows to handle the request to close the window. These “events” are in the \texttt{java.awt.event} 
package and will be discussed later in chapter 8.5.

### 2.3 Ptplot – Extending Javas Graphics Capabilities

The quality of the plot of the simulation results are rather poor compared to high standards we are used to 
today. It is clear, that we could now go on refining the plot with the help of the Java AWT. Although this 
might be an interesting and instructive task, it is not our primary interest in this book. Fortunately, there 
are advanced 2D graphics components which can be used in applets and applications.

One of these packages is Ptplot (you pronounce it pee–tee–plot). The Ptplot package is contained in a 
large project called \texttt{Ptolemy} and is released under the liberal UC Berkley copyright. It has been developed 
by Edward A. Lee, C. Hylands, and W. Wu. You are free to download it together with the full Ptolemy 
package or as a stand alone package at http://ptolemy.eecs.berkeley.edu/java/ptolemy.plot2.0/ptolemy/plot 
where you also find the documentation and many demos of Ptplot.

The components of Ptplot have the following properties:

- plots are embeddable in applets and applications
- you may use binary or ASCII data
- the plots are auto–ranging
- you may label automatically or manually the axes
- logarithmic axes
- live, animated plots
- infinite zooming
- various plot styles (connected lines, scatter plots, bars, ..)
- various point styles (none, dots, points, ....)
- multiple data sets and legends
- color or black and white plots
- error bars.

Before writing the first program using Ptplot take a look at the class hierarchy of Ptplot in Figure 2.7. The 
most important classes are the \texttt{Plot} and the \texttt{PlotBox} classes.

Let us now look at a very simple program in order to demonstrate what we need to invoke the Ptplot 
methods.

\begin{verbatim}
/** <applet code=”Ptplot_Demo1.class” width=400 height=400> Run Applet </applet> */

import java.applet.Applet;
import java.awt.*;
\end{verbatim}
First, we see that we have to import additionally the Ptplot package by using

```java
import ptolemy.plot.*;
```

The class Ptplot_Demo extends the class PlotApplet. Again we want to run the code as an applet as well as an application so the class does have a main method. In lines 14 to 17 we instantiate the new Frame and activate the WindowListener as we did in the RadioactiveDecay_plotEasy.java code. The actual plot routines are in lines 28 to 33. In the init() method we invoke the method super.newPlot() to create a new plot, super.init() to initialize it and plot().setTitle to give the plot a title.

A second possibility of using Ptplot, which we prefer to use, is to extend the Applet class instead of the PlotApplet class and change the lines 28 to 33 to:

```java
// Create a new Plot
Plot plot = new Plot();
```
The difference is that the PlotApplet class realizes a kind of interface for an applet to start plot commands confirming to the pxgraph commands and executes them from parameters given in the HTML file for the applet. Pxgraph is a program for the X windows system to plot data using batch files like Gnuplot. The full Pxgraph functionality is included in the ptplot package and can be used. For further documentation concerning this point, please refer to the ptplot documentation.

Next we want to draw the results of the simulation of the radioactive decay process with the help of ptplot and learn at the same time how to exploit the features of ptplot. The corresponding code can be seen below.

```java
import java.awt.*;
import java.awt.event.*;
import ptolemy.plot.*;

public class RadioactiveDecay_ptplot extends Applet {
    private int N=1000; // initial number of part.
    private double t_end=100; // end time of simulation
    private double decay_const=0.001; // decay constant lambda
    private double dt=1; // time increment for saving
    private double prob=decay_const*dt; // decay probability
    public int[] N_simu,N_exact; // the array for saving results
    public static Plot plot1;

    /** The main routine for running the program as an application */
    public static void main(String[] args) {
        Applet applet = new RadioactiveDecay_ptplot();
        Frame frame = new Frame("RadioactiveDecay using PTPlot");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0); }
        });
        frame.add("Center",applet); // add applet to the window
        applet.init(); // start applet
        frame.pack();
        frame.show(); // display window on screen
    }

    /** The actual main program, started by a browser or by the main method Calculate a radioactive decay and plot the resulting points using the PTPlot classes. Compare with the exact result. */
    public void init() {
        int steps, N_save, N;
        double jump;

        steps = (int)(t_end/dt)+1;
        N_simu=new int[steps];
        N_exact=new int[steps];
        N_exact[0]=N_0;
        N_simu[0]=N_0;
        N_save=N_0;
        N=N_0;
        // Advance : time steps
```
for (int t = 0; t < steps - 1; t++) {
    // transitions until the next measure point (dt intervals)
    for (int i = 0; i < N save; i++) {
        jump = Math.random();
        if (jump < prob) N = ;
    }
    N save = N;
    // save the number of particles in an array
    N simu[t + 1] = N;
    N exact[t + 1] = (int)(N0 * Math.exp(-decay const));
}

// start a new plot and plot the points
plot1 = new Plot();
add(plot1);

int t max = (int)t end;
// set the size of the fonts for title and labels
plot1.setLabelFont("Serif-bold-16");
plot1.setTitleFont("Serif-bold-24");

plot1.setTitle("Radioactive Decay"); // Title of plot
plot1.setMarksStyle("none"); // dots, points or various
plot1.setXLabel("time t"); // set the labels of the axes
plot1.setYLabel("Number of Particles");
plot1.setXRange(0, t max); // set the x range
plot1.setGrid(true); // Grid or not ?
plot1.setYLog(false); // logarithmic plot ?
plot1.setBars(false); // should I use bars ?
plot1.setButtons(true);
// Create the ticks for the axis
for (int i = 0; i <= t max; i += 50) {
    plot1.addXTick(Integer.toString(i), i);
}
for (int i = 0; i <= N0; i += (int)(N0/10)) {
    plot1.addYTick(Integer.toString(i), i);
}

// plot the points and connect them
boolean connect = false;
for (int t = 0; t < steps; t++) {
    plot1.addPoint(0, t, N simu[t], connect);
    plot1.addPoint(1, t, N exact[t], connect);
    if (connect == false) connect = true;
}
}

The output of the code can be seen in Fig. (2.8).

The actual plotting code starts in line 55 and ends in line 79. There we use several methods of
the ptplot.Plot class and of the superclass PlotBox of the Plot class. All methods are called
setMethod where Method is self-explaining. Several other methods are implemented in the Ptplot.PlotBox
class and its child, the Ptplot.Plot class. They are summarized in table 2.4.

Although this is the most common way of using ptplot in this book, there is also the Pxgraph feature
coming with ptplot, mentioned above. For us it will be easier to use ptplot directly in the Java program to
realize plots.

2.4 Plot Methods in the Simulation Package

In the previous section we have seen how to plot the result of the simulation with the help of the ptplot
package. Here, we will present other features that we have added to the simulation package in order to
Figure 2.8: The output of the RadioactiveDecay_ptplot.java program.
Table 2.4 Overview of all the Pplot methods in the Plot and PlotBox classes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>addLegend(int, String)</td>
<td>draw a legend for one plot number</td>
</tr>
<tr>
<td>addXTick()</td>
<td>add a tick to x-axis</td>
</tr>
<tr>
<td>addYTick()</td>
<td>add a tick to y-axis</td>
</tr>
<tr>
<td>setBackground(Color)</td>
<td>set the background color of plot</td>
</tr>
<tr>
<td>setForeground(Color)</td>
<td>set the foreground color of plot</td>
</tr>
<tr>
<td>setGrid(boolean)</td>
<td>draw a grid</td>
</tr>
<tr>
<td>setLabelFont(String)</td>
<td>font for axis labels and legend labels</td>
</tr>
<tr>
<td>setTitle(String)</td>
<td>title of graph</td>
</tr>
<tr>
<td>setTitleFont(String)</td>
<td>set title font</td>
</tr>
<tr>
<td>setXLabel(String)</td>
<td>label of x axis</td>
</tr>
<tr>
<td>setYLabel(String)</td>
<td>label of y axis</td>
</tr>
<tr>
<td>setXLog(boolean)</td>
<td>x axis logarithmic scaling</td>
</tr>
<tr>
<td>setYLog(boolean)</td>
<td>y axis logarithmic scaling</td>
</tr>
<tr>
<td>setXRange(double, double)</td>
<td>x range of the plot</td>
</tr>
<tr>
<td>setYRange(double, double)</td>
<td>y range of the plot</td>
</tr>
<tr>
<td>addPoint(int, double, double, boolean)</td>
<td>add a point to the plot, the boolean variable decides, if the point gets connected with the last one. The integer variable is the plot number.</td>
</tr>
<tr>
<td>addPointWithErrorBars(int, double, double, double, double, boolean)</td>
<td>add a point to the plot with errorbars. Th additional vars. specify the lower and higher y coordinate of the error bar.</td>
</tr>
<tr>
<td>setBars(boolean)</td>
<td>bar plotting on or off</td>
</tr>
<tr>
<td>setBars(double, double)</td>
<td>define width and offset for bar plotting and enable bar plotting.</td>
</tr>
<tr>
<td>setImpulses(boolean)</td>
<td>plot impulses</td>
</tr>
<tr>
<td>setMarksStyle(String)</td>
<td>none, points or various</td>
</tr>
</tbody>
</table>
2.4. PLOT METHODS IN THE SIMULATION PACKAGE

make the plotting from inside a Java program easier.

**plot2D()**  The `plot2D()` method has several different argument lists. The main purpose of the method is to ease plotting of data contained in an array. If you have for example an array, which you want to plot versus the index of the array, just issue

```
simulation.Plotting.plot2D(array);
```

This command creates a new frame and plots the data into this frame using ptplot. If you also want to specify the x-coordinates, just use

```
simulation.Plotting.plot2D(arrayX, arrayY);
```

If we do not want a new frame, but want to add the points to an existing ptplot `Plot` object, we can use

```
simulation.Plotting.plot2D(plot, plotNo, arrayX, arrayY);
```

where `plot` is an object of type `Plot` and `plotNo` is an integer, referring to the ptplot dataset to be used. We can avoid any unnecessary arguments if we like. For details look at the simulation package documentation.

**barPlots()**  There is also a convenience method for plotting bargraphs with ptplot. To use a new frame and plot a bargraph for given boundaries of the bars (points) and given heights of the bars, we can use

```
simulation.Plotting.barPlot(points, heights, barWidth);
```

The `barWidth` specifies the width of the bars to be drawn as a double value. Again there is another version, which does not create a new frame, but uses an existing `Plot` object.

```
simulation.Plotting.barPlot(plot, plotNo, points, heights, barWidth);
```

**errorBars()**  Plotting an errobar plot is very important for scientists. Fortunately ptplot contains functions for errorbar plotting. We have only added again some convenience routines to make it easier. Ptplot is made for assymmetric error bars, but mostly we are faced with symmetric error bars. So to avoid writing the same lines over and over again, we can use

```
simulation.Plotting.errorBarPlot(x, y, yerr);
```

to plot arrays `x` and `y` containing data with a given symmetric error `yerr` in the y-coordinate into a new Frame. For assymmetric error bars use

```
simulation.Plotting.errorBarPlot(x, y, yerrUp, yerrDown);
```

Again there are two more methods doing exactly the same, but do not create its own frame. And if we do not specify the x-coordinates, the y array is plotted versus the index of the array.

**Function Plotting**  Sometimes we want to plot analytical functions for comparisons. To that end we have written a method, which takes a function as an argument and plots the function in a specified range. For example, if we want to plot a \( \sin(x)/x \) function, we use

```java
public class Test {
    /** a function */
    public double function(double x) {
        if (x==0) {
            return 1; }
        else {
            return Math.sin(x)/x; }
    }

    public static void main(String[] args) {
        Test fct = new Test();
        Plotting.plotFunction(fct, 1000, -50, 50);
    }
}
```
The $x$ range is from $-50$ to $50$ and the method uses 1000 points in this interval. If you just supply the function, the method uses a default interval of $[0, 1]$ and 1000 points. If we want to specify a `Plot` object, we have to specify it as the first argument to the method.

**histogram()** A last very important plotting feature needed very often is the plotting of a histogram to estimate for example a probability distribution, as we will see later. Usually you have a data set and want to sort the data according to its values. For example we have data measured from a thermometer giving the temperature in your living room over the last 12 months. Now we want to know how often did we have between 15 and 16 degrees Celsius, how often between 16 and 17, and so on. For this we just have to count the number of temperatures falling in between the given range. This is exactly what is called a histogram.

Because plotting a histogram is actually a two step process, we have to use a separate class to get a histogram plot. Assume we have a data set in a double array called `array` containing $N$ numbers.

- Computing the histogram is done in the `Histogram` class:

  ```java
  Histogram histo = new Histogram(array); // create a histogram
  histo.estimate(); // compute the histogram from the data
  ```

- Plotting or displaying the histogram can be done by using

  ```java
  histo.plotInFrame(); // plot the histogram
  histo.print(); // display the histogram as numbers
  ```

The methods presented above compute the bins to be used by calculating the mean $\mu$ and the variance $\sigma^2$ of the data and choose 20 equal width bins across the range $[\mu - 2\sigma, \mu + 2\sigma]$. If we want to choose our own bin positions and sizes we have to use one of the other methods supplied by the `Histogram` class. For all the methods and fields available, please use the documentation of the simulation package.

**The Radioactive Decay** In order to demonstrate the application of the simulation package features we consider again the simulation of the radioactive decay.

Some results of simulations are plotted in Fig. (2.4) and (2.4). As we already know from Fig. 2.4 the results of the simulation fluctuate around the expected curve. This is of course not astonishing since the exact result holds only for mean values. In order to achieve a better agreement with the decay law it is necessary to run the simulation several times and to take the average over the different realizations of the decay process. This can easily be achieved by a simple modification of the program `RadioactiveDecay_ptplot.java`. We introduce an additional input variable, the number of realizations `nreal` and accordingly implement a loop over the different realizations. This can be best seen in the listing of the new program `RadioactiveDecay_ptplot2.java`.

```java
N_exact=new int[steps];
N_exact[0]=N0;

for (int r=0; r<nreal; r++) {
    N_simu[0]=N0;
    N=N0;
    N_save=N0;
    // Advance : time steps
    for (int t=0; t<steps-1; t++) {
        // transitions until the next measure point (dt intervals)
        jump=Math.random();
        if (jump < prob*N0) N--;
        // save the number of particles in an array
        N_simu[t+1] += N;
        if (r==0) N_exact[t+1]=(int)(N0*Math.exp(-decay_const*t));
```
2.4. PLOT METHODS IN THE SIMULATION PACKAGE

Figure 2.9: Two realizations of the stochastic process of the radioactive decay. The first one with linear y axis scaling and the second one uses a logarithmic y axis scaling. The blue lines are the exact solution and the red ones are the simulations. The parameters of the simulation were chosen to be $N_0 = 100; \ p = \lambda \Delta t = 0.01 s^{-1}; \ \Delta t = 1s; \ \tau_{end} = 300s$

Figure 2.10: The same as figure 2.4, but with different parameters: $N_0 = 1000; \ p = \lambda \Delta t = 0.03 s^{-1}; \ \Delta t = 1s; \ \tau_{end} = 100s$
At the end of the realizations loop we perform the average. This is seen in lines 59 to 62.
Note, furthermore, that in order to speed up the program we have modified slightly the algorithm so that we can save a loop. We have exploited the fact that the probability to observe one decay in time \( \Delta t \) is proportional to number of nuclei which did not already decay, i.e.,

\[
p = \beta \Delta t
\]

where \( \beta = \lambda N \) and \( \Delta t \) must be small enough so that \( \beta \Delta t \ll 1 \).

What about \( \Delta t \) ?????????????????

With the help of the `Histogram` class, we now want to look at the statistics of the number of decays in a given interval.

It is now easy to verify that the number of decays in a given interval is distributed according to the Poisson distribution. To this end we have counted the number of decays in a given interval. This is accomplished in the lines xy. At the end of the program we plot in a histogram the distribution of the number of decays and overlay the expected Poisson distribution.

The results of this test is seen in Fig. xy. In fact it can be shown that the number of decays in a given interval are distributed according to a Poisson distribution. From the elementary rules of combinatorics we know that the probability to observe \( n \) decays in time \( t = m\Delta t \) is therefore given by

\[
P = p^n (1 - p)^{m-n} \binom{m}{n}.
\]

Inserting the definition of \( p \) the above expression can be cast in the form

\[
P = \left( \frac{\beta t}{m} \right)^n \left( 1 - \frac{\beta t}{m} \right)^{m-n} \frac{m!}{(m-n)n!}.
\]

Performing the limit \( \Delta t \to 0 \) (i.e. \( m \to \infty \)) and considering that

\[
\left( 1 - \frac{\beta t}{m} \right)^m \to \exp(-\beta t),
\]

\[
\left( 1 - \frac{\beta t}{m} \right)^{-m} \to 1,
\]

and

\[
\frac{m!}{(m-n)n!} \to m^n
\]

we obtain the result

\[
P = \frac{\mu^n \exp(-\mu)}{n!},
\]

where \( \mu = \beta t \). The above distribution is the well know Poisson distribution.

Listing of the full Program

```java
/* <applet code="RadioactiveDecay_plots2.class" width=800 height=450>
Run Applet </applet> */
import java.awt.*;
import java.awt.event.*;
```
import ptolemy.plot.*;
import simulation.*;

public class RadioactiveDecay_ptplot2 extends java.applet.Applet {
    private static int width=800, height=450;

    private int nreal = 1000; // number of realizations
    private int N_0=100; // initial number of part.
    private double t_end=100;  // end time of simulation
    private double decay_const=0.001; // decay constant lambda
    private double dt=1;  // time increment for saving
    private double prob=decay_const*dt; // decay probability

    public int[] N_simu, N_exact, N_decay; // the array for saving results
    public static Plot plot1, plot2;

    /** The main routine for running the program as an application */
    public static void main(String[] args) {
        java.applet.Applet applet = new RadioactiveDecay_ptplot2();
        Frame frame = new Frame("RadioactiveDecay using PTPlot");
        frame.addWindowListener(new WindowAdapter() {
            // Handle window close requests
            public void windowClosing(WindowEvent e) { System.exit(0); }
        });
        frame.setSize(width, height); // set size of window
        frame.add("Center", applet); // add applet to the window
        frame.show(); // display window on screen
        applet.init(); // start applet
    }

    /** The actual main program, started by a browser or by the main method
    Calculate a radioactive decay and plot the resulting points using
    the PTPlot classes. Compare with the exact result. */
    public void init() {
        int steps, N, N_save;
        double jump;

        steps = (int)((t_end/dt)+1);
        N_simu=new int[steps];
        N_decay=new int[nreal];
        N_exact=new int[steps];
        N_exact[0]=N_0;

        for (int r=0; r<nreal; r++) {
            N_simu[0]=N_0;
            N=N_0;
            N_save=N_0;
            // Advance : time steps
            for (int t=0; t<steps-1; t++) {
                // transitions until the next measure point (dt intervals)
                jump=Math.random();
                if (jump < prob* N_0) N--;
                // save the number of particles in an array
                N_simu[t+1] += N;
                if (r==0) N_exact[t+1]=(int)(N_0*Math.exp(-decay_const*t));
            }
            N_decay[r]=N_0-N;
        }
    }
// compute mean values
for (int t = 1; t < steps - 1; t++) {
    N_simu[t] = nreal;
}

// start a new plot and plot the points
plot1 = new Plot();

int t_max = (int)t_end; // for plotting tickmarks !!
// set the size of the fonts for title and labels
plot1.setTitleFont("Serif-bold-18");
plot1.setTitleFont("Serif-bold-24");

plot1.setTitle("Radioactive_Decay"); // Title of plot
plot1.setMarksStyle("none"); // dots, points or various
plot1.setXLabel("time_t"); // set the labels of the axes
plot1.setYLabel("Number_of_Particles");
plot1.setXRange(0, t_max); // set the x range
plot1.setGrid(true); // Grid or not?
plot1.setYLog(false); // logarithm is plot?
plot1.setBars(false); // should I use bars?
plot1.setButtons(true);

// Create the ticks for the axis
for (int i = 0; i < t_max; i += 50) {
    plot1.addXTick(Integer.toString(i), i);
}

for (int i = 0; i < N0; i += 100) {
    plot1.addYTick(Integer.toString(i), i);
}

// plot the points and connect them
boolean connect = false;
for (int t = 0; t < t_max; t++) {
    plot1.addPoint(0, t, N_simu[t], connect);
    plot1.addPoint(1, t, N_exact[t], connect);
    if (connect == false) connect = true;
}

add(plot1);

/* Compute the decay distribution */
Histogram histo = new Histogram(N_decay);
histo.setPointsUniform(0, 50);
histo.estimate();
plot2 = histo.plot();
plot2.setBars(false);
plot2.setImpulses(true, 1);
add(plot2);

// plot the exact probability distribution: Poisson
double y;
connect = false;
double dummy = simulation.DataAnalysis.arraySum(histo.getHistogram());
for (int i = 0; i < steps / 2; i++) {
    y = simulation.Distribution.Poisson(i, prob * N0 * N0 * dummy);
    plot2.addPoint(0, i, y, connect);
    if (connect == false) connect = true;
}

Run the program for the following two sets of parameters:
\[
N_0 = 100, p = 0.001 \text{s}^{-1}, \Delta t = 1 \text{s}, t = 100\text{s} \\
N_0 = 100, p = 0.0001 \text{s}^{-1}, \Delta t = 1 \text{s}, t = 100\text{s}
\]
with nreal = 100 and nreal = 1000.
2.5 Printing in Java and with Ptplot

Now that we have learned how to make plots in Java, it is of great importance to get a printed version of our graphics. For this purpose Java has commands for initiating a print job and preparing the output for the printer. Java always produces postscript output. Because the procedure has changed from Java 1.1 to Java 2, we only present the Java 1.1 version not to get confused. In Java 1.0 there have been no methods for printing in Java, we would have to use screen capture programs for example.

Because printing is of course system dependent, it is not as easy as issuing a print command, but it is still manageable. In Java printing is done basically in four steps:

1. Get a Toolkit for the component we want to print. Use method getToolkit() in the java.awt.Component class.

2. Get a PrintJob. Use the getPrintJob(Frame f, String printjobname, Properties printprops) method in the java.awt.Toolkit class.

3. Start “printing”:
   (a) Get the graphics context for the component in question. Use the getGraphics() method of the java.awt.PrintJob class.
   (b) Print the desired part of the component. If we want to print everything contained by the component use the printAll() method, otherwise use just the print() method of the Component class.
   (c) Send the data to the printer or file by using the dispose() method of the java.awt.Graphics class.

4. Finish printing and close dialog box. Use end() method of the java.awt.PrintJob class.
Figure 2.12: The distribution of the number of decays using 1000 realizations. The simulation was run for $N_0 = 100$ and $\lambda = 0.001$.

The program RadioactiveDecay_printing.java demonstrates the use of the printing capabilities of Java.

```java
/* Print the plot in a file or an printer */
// 1.) get a "connection" to the printer
Toolkit toolkit = plot.getToolkit();
// 2.) get a dialog box for the print job
PrintJob job = toolkit.getPrintJob(frame, "Radioactive Decay". Properties ) null);
if ( job != null ) {
    // 3.) get the graphics handle
    Graphics pg = job.getGraphics();
    // print all components contained in com
    plot.printAll (pg);
    // send it to printer
    pg.dispose();
    // 4.) close all necessary stuff
    job.end();
}
```

Of course there are some drawbacks to talk about. First of all with this code we always get the output in a size referring to our actual picture on the screen, it does not use the full paper size. If we want to use the whole page size available we have to scale the component to be plotted to the full size and scale back after printing. This is what we do in the convenience class we have written for easy printing in the simulation package.

So if we want to print a component scaled to the full size, we use the PrintComponent.Dialog() method in the simulation package. It scales our component to the full size — no matter if it should be on a portrait or landscape page — and sends the scaled picture to the printer or a postscript file. After printing it scales back to the size it had before. The whole code above could therefore be substituted by the line

```java
import simulation.*;
```
2.6. ADVANCED TOPICS

Because most of the time we need postscript or encapsulated postscript (EPS) files of the plots or graphics, this is the most common use of the printing features for most scientists.

A second method of the simulation class called DialogNoScaling() can be used to produce unscaled output of the component or container. It has the same syntax as the Dialog() method before.

There is one caveat to mention: if we are going to insert the produced postscript files into TeX/LaTeX, we will have to do some editing on the postscript files. First of all we have to add a bounding box line at the beginning of the postscript file. We can do this by hand: put a line (for a portrait figure)

```
%%BoundingBox: 0 0 595 840
```

at the beginning of the postscript file (We can adapt the numbers to the figure at hand and check the area by viewing it in ghostscript/ghostview.) or we can use a program to automatically calculate and insert a bounding box command (like e.g. epstool, gsview on Windows, etc.). The second change is to remove all the lines between %BeginSetup and %EndSetup except a possible rotate line like 90 rotate 0 - 595 translate. This is necessary, because some strange effects appear in the resulting .dvi or .ps file after "texting".

Another drawback might be the resolution of the postscript file. This happens especially when printing GUI components and is not present when plotting ptplot plots fortunately. Java kind of rasters the screen display and because the screen resolution is much worse than the printer resolution this gives unpleasant results. It also gives strange results if we scale a large GUI to a small A4 or Letter format for printing, which could look very ugly or even miss some of the displayed objects. But as already mentioned, most of the time we only need ptplot plots in postscript files and this works great with the code above.

In Java 2 there has been some changes to the printing interface. A new class has been introduced in Java2: java.awt.print. This class provides much more sophisticated methods and it is even able of handling color models, which is very important for using colors on displays and printers. This is a big step ahead, but for the details we refer to the API documentation, because it is also a bit more difficult to understand.

Ptplot 2.2 printing facilities ????? (EPS)

Printing from an Applet: Usually applets are not allowed to initiate print jobs, unless a SecurityManager, another important Java class, explicitly allows for it. The browser or the appletviewer have to initiate the print job, and they only allow it, if the security manager does. But the person who executes the applet can always use the printing facilities of the browser and plot the whole panel visible in the browser, but this is from outside the Java program.

What we can do is, write a main method in the applet and start it as an application. Do the plots and print them from the application.

If we use a security manager and are allowed to print, we need a frame for our applet, because the getPrintJob() command only takes a frame and not an applet as first argument. So just put a frame into the applet and then put the applet inside the frame.

2.6 Advanced topics

2.6.1 3D plots in Java – Java3D

Sometimes we are forced to use 3D plots to visualize our data and therefore it is natural to ask for a package to accomplish three dimensional plots. Unfortunately to our knowledge there is (yet) no freely available package in Java for 3D plots. Ptplot can only handle 2D plots and will not be extended to 3D in the future. JSci (see chapter 1.13.3) can at least handle some very basic 3D plots.

Because it is the only (free) way of producing 3D graphs yet, an example in Figure 2.13 introduces some of these features. It is actually not what we expected, but it is a beginning. We can also use the mouse to rotate the 3D graph in the right half of the window. The graph on the left is a contour plot. The details are explained in the API documentation to the JSci package.
import JSci.awt.*;
import java.awt.*;

public class JSci3DGraph {

    public static void main(String[] args) {
        ContourPlot contourGraph;
        LineGraph3D line3DGraph;
        final int N = 20;
        double[][] array = new double[N][N];

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                array[i][j] = Math.random();
            }
        }

        Frame f = new Frame("Test");
        f.setSize(400, 500);

        line3DGraph = new LineGraph3D(array);
        contourGraph = new ContourPlot(array);

        f.setLayout(new GridLayout(2, 2));
        f.add(contourGraph);
        f.add(line3DGraph);
        f.show();
        // simulation.PrintComponent.Dialog(f, "Print It");
    }
}

There is also a program called SciVis\textsuperscript{2}, which is completely written in Java and allows for 2D and

\textsuperscript{2}http://kopernik.npac.syr.edu:8888/scivis/
3D plots in many different ways. But up to now, there are only C and Fortran interfaces to supply data to it. We can also read from files, but first we have to understand the data format, which is described in the users guide. The authors told us they are developing a Java interface to supply data directly to SciVis.

At the moment, the best solution is to write the data to a file and use an external program available to us. The most common denominator would possibly be “Gnuplot”, which is available for many different platforms and can create a lot of different 3D plots. And in the next chapter we will learn how to call gnuplot inside a Java program.

Since Java 2, there is a new (external) API – not included in the JDK 1.2 distribution, we have to get it separately – called Java3D. This is a full implementation of three dimensional routines to produce all kinds of 3D scenes and objects and even move these scenes. With this API it seems to be possible to write a 3D plotting program in the near future. So the lack of 3D plotting features in Java should be hopefully gone soon.

2.6.2 Using (system dependent) external programs like gnuplot

There is a last way of obtaining 3D plots “in Java”: We can use an external program like Gnuplot or any other command line tool and call it from a Java program. This of course is not system independent and is not recommended unless we are desperately needing it.

In Java there is a class called Runtime in the java.lang package, which consists of all kind of methods to change and use the environment we are running our Java program in. This can be used to start a subprocess of the running Java program. So starting an external process from a Java program can be done in two steps:

- Get a Runtime object of the running Java program.
- Start a new process as a subprocess of the given Runtime object. We have to use the exec(String) instance method of the Process class for this.

We can also read the standard output of the subprocess and use it in our Java program. To demonstrate how it actually works, we give two examples:

1. A Java program starts a Gnuplot program, which plots a sine curve.

```java
/**
 * Gnuplot.java
 *
 * @author Peter Biechele
 * @version 1.0
 */

import java.io.*;

public class Gnuplot {

    public static void main(String[] args) throws IOException {

        // get a Runtime object
        Runtime r = Runtime.getRuntime();

        // start the process: gnuplot
        Process p = r.exec("gnuplot_Gnuplot.gnu");
    }

} // Gnuplot
```
We also need the Gnuplot program to be executed:

```
plot sin(x)
pause 5
```

2. A Java program executes a “ls -al” command on a UNIX machine, which just gives the directory listing. For a Windows system we just use the “dir” command instead.

```
/**
 * DirectoryListing.java
 */

5 * Created: Wed Jun 30 17:51:32 1999
*
* @author Peter Biechele
* @version 1.0
 */

import java.io.*;

public class DirectoryListing {

    public static void main(String[] args) throws IOException {

        // get a Runtime object
        Runtime r = Runtime.getRuntime();

        Process p;

        // start the process: UNIX
        p = r.exec("ls -al");

        // start the process: Windows
        // p = r.exec("dir ");

        // wait for the process to finish
        try {
            p.waitFor();
        } catch (InterruptedException e) {
        }

        System.out.println("Exited Process!");

        // read the standard output from the process
        BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));

        int c;
        while ((c = in.read()) > 0) {
            System.out.print((char)c);
        }
    }
}
```

This time we also wait for the execution of the process to finish and then read the standard output and display it on the screen.
2.7 Exercises

Use the Java method `Math.random()` to solve the following problems (do not care about the quality and the algorithm of the random number generator, for now):

**Exercise 2.1 Photoabsorption**

Consider the absorption of photons passing through a gas in two dimensions. We model the gas by introducing slabs of width $dx$ and density $n$ (in particles per area), which absorb the incident photons. The slab particles have a cross-sectional area of $\sigma$.

So the probability of a photon to be absorbed in the slab will be ($M$ is the number of particles in the slab of the height $dy$)

$$P(\text{Photon absorbed}) = \frac{M \sigma}{\frac{\sigma n dx dy}{dy}} = \frac{\sigma n dx}{dx} = \sigma n dx.$$ We have assumed that there is no overlap between the cross-sections of the slab particles.

![Figure 2.14: The configuration of the system.](image)

Write a program to simulate this process on the computer. Take $N$ incident photons and watch the number of particles left over against the slabs passed in a diagram. Do this simulation several times and calculate the ensemble-average. What process we know is similar to this behaviour and what takes the place of the spatial dimension in that case?

**Exercise 2.2 Monte-Carlo Integration – Speed and Accuracy**

Write a program for the calculation of the following integral:

$$I = \int_{0}^{1} \frac{1}{1+x^2} dx.$$  

1. using the hit and miss method
2. using the standard method
For both algorithms, calculate the mean and the standard deviation. Also use the analytical result of the integral to calculate \( \pi \). Compare the accuracy of both algorithms using the approximations of \( \pi \). Compare the speed of the two programs by using the \texttt{cputime} function in Matlab. (e.g. type the following to time the random number generator: \texttt{t=cputime; x=rand(1000); cputime-t})

To this end, create a table and a plot with the two parameters (\( n \): the number of intervals and \( m \): the number of realizations) against the accuracy (use at least 5 values). To save time, we can first check for a good \( n \) and then do the plots only against \( m \). For the speed, plot the cputime against the achieved accuracy for many different \( m \).

**Exercise 2.3 Eulers Constant using Monte-Carlo Algorithm**

Suppose throwing \( N \) darts randomly at a dart board, which has been divided into \( R \) equal size regions. The probability of hitting one region is \( p = 1/R \). Then the probability of hitting an empty region (not already occupied by a dart) is \((1-p)^N\). Using the binomial distribution, we can get the probability for hitting a region with \( m \) darts. If we choose the number of regions equal to the number of darts thrown on the board, we have \( p = 1/N \) and therefore

\[
P(\text{hitting an empty region}) = \left(1 - \frac{1}{N}\right)^N.
\]

Because the above series converges to \( e \) for \( N \to \infty \), we can use the following method to get an approximation of the Euler constant:

(i) Throw randomly a large number of darts (say \( N \)) on a board, which has been divided into \( N \) equal size regions.

(ii) Count the number of empty regions (call it \( N_0 \)).

(iii) The fraction \( N/N_0 \) is a good estimate of the Euler constant \( e \).

Write a program for that algorithm and check the results. We can also use \( N\) \( N_1 \), if \( N_1 \) is the number of regions with the occupancy of one dart. Check this, too. What \( N \) do we need to get the same accuracy using the formula? And how many terms of the series for \( e \) (\( \sum_{i=0}^{\infty} 1/i! = e \))?
Part II

Introduction to Stochastic Variables
Chapter 3

Stochastic Variables

Since the notion of random variables will be essential for the understanding of stochastic methods this chapter will be devoted to the introduction of the fundamental concepts of probability theory.

3.1 The Nature of Probabilities

In the previous chapter we have already made use of probabilistic notions in an intuitive way. However, we have not asked the following question: What are probabilities? How can we formulate the notion of probability in such a way that it is useful for physical applications?

Essentially, there are three possible definitions of probability Brody [1993]: a) the axiomatic interpretation, b) the frequency interpretation, and c) the ensemble interpretation.

3.1.1 The Axiomatic Interpretation

The axiomatic definition Feller [1950] of probabilities has been proposed by Kolmogorov in 1933. The formal objects to which we want to attribute probabilities are called events and are subsets of a basic set $\Omega$ which is called the event space or in physical applications the phase space. If the event $e$ belongs to $\Omega$, so does its complement $\Omega - e$ also; the null event $\emptyset$ is therefore also in $\Omega$. Events containing only one member of $\Omega$ are called the elementary events of $\Omega$.

A function $P(e)$, called the probability of $e$ can be assigned to each event $e$ in $\Omega$. The function $P(e)$ has the following properties:

(i) $P(e) \geq 0$ for all $e$ in $\Omega$;
(ii) $P(\Omega) = 1$;
(iii) If $e_1, e_2, \ldots$ are in $\Omega$ and are pairwise disjoint, i.e., $e_i \cap e_j = \emptyset$ when $i \neq j$, then $P(e_1 \cup e_2 \cup \ldots) = P(e_1) + P(e_2) + \ldots$.

It follows immediately from the above three axioms that

(iv) If $\bar{e}$ is the complement of $e$, i.e., the set of all events which are not in $e$, then $P(\bar{e}) = 1 - P(e)$;
(v) $P(\emptyset) = 0$.

3.1.2 The Relative Frequency Interpretation

In his attempt to axiomatize probability theory, von Mises introduced in 1919 the notion of a Kollektiv, which stands for a single infinite sequence of random events such as the outcomes of throwing a coin. He defined then the probability of some event to be the limit of its relative frequency in such a series of observations when the series becomes infinitely long (the Kollektiv) Compagner [1991]; Brody [1993]. If we denote by $n$ the number of data in the series, by $m(e)$ the number of times the event is observed in it, then the probability $P(e)$ is defined as

$$P(e) = \lim_{n \to \infty} \frac{m(e)}{n}.$$  \hfill (3.1)
Of course, such a series of events must have the property that any infinite subsequence in it must have the same limit.

The problem with this definition is the following one: How can any sequence of experimental data, which will be always be finite, have the properties of such a Kollektiv? In practice the relative frequencies for subsequencies will always differ from that in the main sequence.

To illustrate the problems with the frequency interpretation of probabilities we consider the following example. We throw a die \(n\) times and look at the relative frequency \(m(4)\) of the outcome of throwing a 4. This experiment will be simulated with the help of the following program.

### Listing of the program relfreq.

```matlab
% Program relfreq: simulation of the evolution of the
% relative frequency of throwing a die
clear; help relfreq;
R = rand(400,1);

n = 0;
for i = 1:400
    if (R(i) > 0.5 & R(i) < 2/3)
        n = n + 1;
    end
m(i) = n/i;
end
plot(1:400,m,'x');
title('Evolution of the relative frequency ');
xlabel('n'); ylabel('relative frequency ');
hold on;
plot([1 400],[1/6 1/6],'-');
hold off;
```

The result of an experiment for up to 400 throws is shown in Fig. (3.1.2). Running the program again we observe another approach to the asymptotic value. We recognize immediately the difficulties with von Mises definition of probability.
3.1.3 The Ensemble Interpretation

We know from statistical mechanics the notion of ensemble. An ensemble is a collection of a large number \( N \) of equally prepared systems (equal models). A simple example is the microcanonical ensemble, which is the ensemble of all microstates in phase space, which are characterized by fixing the macroscopic values for the energy \( E \), the volume \( V \) and the number of particles \( N \).

The abstract concept of an ensemble allows naturally the definition of a mean value. We have to consider two cases:

(i) The ensemble contains a finite, discrete number of models: Let \( n \) be the number of models and \( Q(i) \) the interesting quantity in model \( i \). The ensemble mean value \( \langle Q \rangle \) is then defined as

\[
\langle Q \rangle = \frac{1}{n} \sum_{i=1}^{n} Q(i).
\]

(ii) The models are characterized by some continuous parameter, i.e., the initial positions of molecules in a gas: If we name the continuous parameter \( \omega \), the phase space as \( \Omega \), and \( n(\omega) \) a weight function characterizing the ensemble, then

\[
\langle Q \rangle = \frac{\int_{\Omega} Q(\omega) d\omega}{\int_{\Omega} d\omega}.
\]

Usually, the function \( n(\omega) \) can be derived on the basis of the theoretical model on which the ensemble relies upon.

Let us consider as a simple example from equilibrium statistical mechanics a gas consisting of \( N \) particles. The microstates of the system are the points \( (q,p) = (r_1,q_1,...,r_N,q_N) \) in the 6–dimensional phase space. The probability to find the microsystem at time \( t_0 \) in a volume element \( dV = d^{3N}q d^{3N}p \) around \( (q,p) \) is given by

\[
dw(q,p) = \rho(q,p) d^{3N}q d^{3N}p,
\]

where \( \rho(q,p) \) is the distribution function. For the microcanonical ensemble the distribution function \( \rho \) simply reads

\[
\rho(q,p) = \left\{ \begin{array}{ll}
    c = \text{const}, & \text{for } E - \Delta \leq H(q,p) \leq E, \\
    0, & \text{otherwise} \end{array} \right.
\]

(3.5)

Probabilities can be introduced as a special kind of ensemble average. Let \( A \) be a property, which the members of the ensemble may have or not and let us define an indicator function

\[
\chi_A(\omega) = \left\{ \begin{array}{ll}
1, & \text{if member labelled } \omega \text{ has property } A \\
0, & \text{otherwise} \end{array} \right.
\]

(3.6)

The probability of \( A \) in the ensemble is simply defined as the ensemble average of \( \chi_A(\omega) \)

\[
P(A) = \langle \chi_A(\omega) \rangle = \frac{\int_{\Omega} \chi_A(\omega) d\omega}{\int_{\Omega} d\omega}.
\]

(3.7)

The probability is the relative weight in the ensemble of those members that have the property \( A \). In the case of a discrete ensemble the above definition of probability reduces to a sum over the members of the ensemble having the property \( A \)

\[
P(A) = \frac{1}{n} \sum_{i=1}^{n} \chi_A(i),
\]

(3.8)

i.e., the probability is the relative frequency of the members having the property \( A \) in the ensemble.

It is clear that from the above definition of probability it is possible to derive the Kolmogorov axioms of probability theory.
It is important to stress that the ensemble is a purely theoretical construction and has to be adapted to
the physical situation of interest as we will see in the future chapters. Furthermore, it is to be noted that the
ensemble interpretation allows the definition of time dependent probabilities, i.e.,

\[ P(A, t) = \langle \chi_A(t) \rangle, \quad (3.9) \]

which are of fundamental importance while studying stochastic processes.

### 3.2 The Definition of Stochastic Variables

A stochastic variable \( X \) is an object which is defined by a space of states (space of events, phase space) and
by a probability density over this set. The space of state may be discrete, e.g. the numbers 1,2,3,4,5,6 for
a play of dice or the number of molecules in a chemical reaction, as well as continuous, e.g. the velocity
of a Brownian particle. Of course, the space of state may also be discrete and continuous at the same time,
e.g., the energy of an electron in the presence of some binding centers. When sampling a one dimensional
continuous stochastic variable \( X \), the probability to find some value in the infinitesimal interval \((x, x + dx)\)
will be expressed symbolically by

\[ P(x)dx \equiv \text{Prob}\{X \in [x, x + dx]\}, \quad (3.10) \]

which defines the probability density \( P(x) \) associated with the stochastic variable \( X \). It follows immediately
from the above equation and from the addition law of probability theory that the probability to sample a
value of \( X \) in the interval \([a, b]\) is given by

\[ \int_a^b dx P(x) = \text{Prob}\{X \in [a, b]\}. \quad (3.11) \]

It is evident from Eqs. (3.10) and (3.11) that the probability density is non-negative, i.e. \( P(x) \geq 0 \) and that it is normalized

\[ \int_{-\infty}^{\infty} dx P(x) = 1. \quad (3.12) \]

For later convenience we remark that the probability density may contain also sums over \( \delta \)-functions. For example \( P(x) \) can also have the form

\[ P(x) = \sum_n p_n \delta(x - x_n) + \bar{P}(x), \quad (3.13) \]

where \( \bar{P}(x) \geq 0 \), \( p_n \geq 0 \), \( \bar{P} \) integrable and the normalization condition is

\[ \sum_n p_n + \int dx \bar{P}(x) = 1. \quad (3.14) \]

The distribution function \( F \) of the stochastic variable \( X \) is defined by

\[ F(x) \equiv \text{Prob}\{X \leq x\}. \quad (3.15) \]

The density function and the distribution function are related by the equation

\[ F(x) = \int_{-\infty}^{x} dx' P(x'), \quad (3.16) \]

or equivalently by \( F'(x) = P(x) \).
3.2. THE DEFINITION OF STOCHASTIC VARIABLES

3.2.1 Further Characterization of Stochastic Variables

A stochastic variable is completely defined by the space of states and by the probability density function. However, it is helpful to introduce some other quantities in order to characterize them.

The expectation value, i.e., the average, of any function \( f(X) \) with respect to the stochastic variable \( X \) is denoted by \( \langle f(X) \rangle \) and is defined by

\[
\langle f(X) \rangle = \int dx f(x) P(x),
\]

(3.17)

Of particular importance are the moments of a distribution. The \( m \)-th moment \( \mu_m \) is defined as \( \langle X^m \rangle \). Of course, \( \mu_1 \) is the mean. The variance \( \text{Var}(X) \) is defined as

\[
\text{Var}(X) \equiv \langle (X - \langle X \rangle)^2 \rangle = \mu_2 - \mu_1^2,
\]

(3.18)

and is the square of the standard deviation \( \sigma \).

Another important quantity is the characteristic function \( G(k) \). It is defined as

\[
G(k) = \langle \exp(ikx) \rangle = \int \exp(ikx) P(x) dx,
\]

(3.19)

and has the obvious properties

\[
G(0) = 1 \quad \text{and} \quad |G(k)| \leq 1.
\]

(3.20)

The characteristic function is also called the moment generating function, because expanding the exponential function in a Taylor series we get

\[
G(k) = \sum_{n=0}^{\infty} \frac{ik^n}{n!} \langle X^n \rangle.
\]

(3.21)

Thus, if \( G(k) \) is known the moments are easily evaluated as

\[
\frac{d^n}{dk^n} G(k) \bigg|_{k=0} = i^n \langle X^n \rangle,
\]

(3.22)

The same function serves to generate the so-called cumulants which are defined as

\[
\ln G(k) = \sum_{n=1}^{\infty} \frac{(ik)^n}{n!} \kappa_n,
\]

(3.23)

and are combinations of the moments, i.e., the first three cumulants are given by

\[
\kappa_1 = \mu_1, \quad \kappa_2 = \mu_2 - \mu_1^2 = \sigma^2, \quad \kappa_3 = \mu_3 - 3\mu_2 \mu_1 + 2\mu_1^3.
\]

(3.24, 3.25, 3.26)

It can be shown Gardiner [1990] that the cumulant generating function cannot be a polynomial of degree greater than 2, that is, either all but the first two cumulants vanish or there is an infinite number of nonvanishing cumulants.

REMARK ( \( \kappa \) papi given then g(k) unique !!!????????????)

3.2.2 Some Important Random Variables

Let us first introduce and discuss briefly some important continuous one dimensional probability densities.
CHAPTER 3. STOCHASTIC VARIABLES

The Uniform Density

The simplest density is the uniform density which is constant if \( x \) lies within the interval \([a,b]\) and zero otherwise, i.e.,

\[
P(x) = \frac{1}{b - a},
\]

(3.27)

It is easy to check that the mean of the uniform distribution is

\[
\langle X \rangle = \frac{a + b}{2}
\]

(3.28)

and that the standard deviation of a uniformly distributed random variable is

\[
\sigma = \frac{b - a}{2\sqrt{3}}.
\]

(3.29)

As we will see the uniform probability density will play a fundamental role in the forthcoming chapters.

The Exponential Density Function

The exponential density function is defined as

\[
P(x) = a \exp(-ax),
\]

(3.30)

where \( a \) is any positive constant. It is easy to verify that the mean and the standard deviation of an exponentially distributed random variable are equal

\[
\langle X \rangle = \sigma = \frac{1}{a}.
\]

(3.31)

The Gaussian or Normal Density Function

The most important density function for physics is the gaussian probability density. It has the form

\[
P(x) = (2\pi a^2)^{-1/2} \exp\left[-(x - x_0)^2 / 2a^2\right],
\]

(3.32)

for \( a \) positive and \(-\infty < x_0 < \infty\). The mean and the standard deviation of the Gaussian probability density are given by \( \mu_1 = x_0 \) and by \( \sigma = a \), respectively. The characteristic function of the Gaussian density reads

\[
G(k) = \exp(i\mu_1 k - \frac{1}{2} \sigma^2 k^2),
\]

(3.33)

which means that \( \kappa_1 = \mu_1, \kappa_2 = \sigma^2 \) and that all higher cumulants vanish.

The Cauchy or Lorentz Density

The Cauchy or Lorentz density is defined as

\[
P(x) = \frac{1}{\pi} \frac{a}{(x - x_0)^2 + a^2}
\]

(3.34)

for positive \( a \) and \(-\infty < x_0 < \infty\). It is an example of a probability density which does not have a finite variance. In fact, not even the integral defining the mean value converges.

Let us now discuss some typical discrete probability densities. The discrete random variable will be denoted by \( N \).
3.2. THE DEFINITION OF STOCHASTIC VARIABLES

The Discrete Uniform Probability Density

The discrete uniform probability density is defined by

\[
P(n) = \frac{1}{n_2 - n_1 + 1}
\]

(3.35)

for \( n_1 \leq n \leq n_2 \) and zero otherwise. Of course, \( n_1 \) and \( n_2 \) are integer numbers and \( n_1 \leq n_2 \). Its mean value is

\[
\langle N \rangle = \sum_n nP(n) = \frac{n_1 + n_2}{2}
\]

(3.36)

and its variance

\[
\sigma^2 = \frac{(n_2 - n_1)(n_2 - n_1 + 2)}{12}.
\]

(3.37)

The above equations are easily proven with the help of the relations

\[
\sum_{n=1}^{N} n = \frac{N(N+1)}{2}, \quad \sum_{n=1}^{N} n^2 = \frac{N(N+1)(2N+1)}{6}.
\]

(3.38)

The Binomial Distribution

Let us assume that the random variable \( Y \) can take only two values \( \{y_1, y_2\} \), the probability for the value \( y_1 \) being \( p \) and correspondingly for \( y_2 \) \((1-p)\). If we consider \( N \) realizations of the stochastic variable \( Y \) the probability to find the value \( y_1 \) \( N \) times under the \( n \) results is the binomial density \( P(n) \)

\[
P(n) = \frac{N!}{n!(N-n)!} p^n(1-p)^{(N-n)},
\]

(3.39)

for \( 0 \leq n \leq N \). The mean and variance of the binomial density are given by

\[
\langle n \rangle = Np,
\]

(3.40)

and

\[
\sigma^2 = Np(1-p),
\]

(3.41)

It is easy to check the normalization of the binomial distribution since

\[
1 = [p + (1-p)]^N = \sum_{n=0}^{N} \frac{N!}{n!(N-n)!} p^n(1-p)^{(N-n)}.
\]

(3.42)

The Poisson Density

The Poisson density as we already know is defined as

\[
P(n) = \frac{\exp(-a) a^n}{n!},
\]

(3.43)

for \( n > 0 \) and \( a \in R \). The mean value and the variance of the Poisson density are equal,

\[
\langle n \rangle = \sigma^2 = a.
\]

(3.44)

As we already know from the discussion of the radioactive decay the Poisson density is a limit of the binomial probability density for \( N \rightarrow \infty \), \( p \rightarrow 0 \) while \( Np = a = \text{const.} \). Another limit of the Poisson density
which deserves consideration is the limit \( a \gg 1 \): In this limit the Poisson density will be essentially different from zero only for \( n \approx a \). For \( n \gg 1 \) the Stirling formula holds
\[
n! \approx (2\pi n)^{1/2} n^a \exp(-n)
\]
so we can write
\[
\ln \left[ \frac{\exp(-a)a^a}{n!} (2\pi a)^{1/2} \right] \approx (n-a)-n \ln \left( \frac{n}{a} \right).
\]
Setting \( \varepsilon = (n-a)/2 \) and since, for \( \varepsilon \ll 1 \), \( \ln(1+\varepsilon) \approx \varepsilon - \varepsilon^2/2 \) and for \( n \approx a \) we can write
\[
\ln \left[ \frac{\exp(-a)a^a}{n!} (2\pi a)^{1/2} \right] \approx -(n-a)^2/2a.
\]
So that finally for \( a \gg 1 \)
\[
\frac{\exp(-a)a^a}{n!} \approx \exp \left( \frac{(n-a)^2}{2a} \right).
\]
Thus, in the limit \( a \gg 1 \) the Poisson density resembles a Gaussian density with mean \( a \) and variance \( a \).

**3.2.3 Multivariate Random Variables**

Up to now we have considered only one dimensional stochastic variables. Obviously, \( n \) random variables \( X_1,X_2,\ldots,X_n \) which are sampled simultaneously can be interpreted as the components of an \( n \)-dimensional stochastic variable \( X \). Their joint density function \( P_n(x_1,\ldots,x_n) \) is defined through the statement
\[
P_n(x_1,\ldots,x_n)dx_1dx_2\ldots dx_n \equiv \text{Prob}\{X_i \in (x_i,x_i+dx_i) \quad \text{for each} \quad i = 1,\ldots,n\}.
\]
If we look at the subset of stochastic variables \( X_1,\ldots,X_s \), for \( s > n \) we can easily write down with the help of the elementary laws of probability theory the joint density function for this set irrespective of \( X_{s+1},\ldots,X_n \)
\[
P_s(x_1,\ldots,x_s) = \int P_n(x_1,\ldots,x_s,x_{s+1},\ldots,x_n)dx_{s+1}\ldots dx_n.
\]
\( P_s \) is a so-called marginal distribution.

The conditional density \( P_{n-k}(x_1,\ldots,x_k|x_{s+1},\ldots,x_n) \) is the joint density of \( X_1,\ldots,X_s \) given that \( X_{s+1} = x_{s+1},\ldots,X_n = x_n \) and is easily shown to be given by Bayes rule
\[
P_{n-k}(x_1,\ldots,x_k|x_{s+1},\ldots,x_n) = \frac{P_n(x_1,\ldots,x_k,x_{s+1},\ldots,x_n)}{P_{n-s}(x_{s+1},\ldots,x_n)}.
\]
Two subsets \( (X_1,\ldots,X_s) \) and \( (X_{s+1},\ldots,X_n) \) are said to be statistically independent if \( P_n \) factorizes
\[
P_n(x_1,\ldots,x_n) = P_s(x_1,\ldots,x_s)P_{n-s}(x_{s+1},\ldots,x_n).
\]
In this case \( P_s \) is the marginal as well as the conditional probability density.

The definition of moments is easily generalized to the multivariate case
\[
\langle x_1^{m_1} \cdots x_n^{m_n} \rangle = \int x_1^{m_1} \cdots x_n^{m_n} P(x_1,\ldots,x_n)dx_1\cdots dx_n.
\]
Accordingly, the characteristic function is given by
\[
G(k_1,\ldots,k_n) = \langle \exp[i(k_1X_1 + \cdots + k_nX_n)] \rangle.
\]
Again the multivariate Taylor expansion in the variables \( k_i \) generates the moments
\[
G(k_1,\ldots,k_n) = \sum \frac{(ik_1)^{m_1} \cdots (ik_n)^{m_n}}{m_1! \cdots m_n!} \langle x_1^{m_1} \cdots x_n^{m_n} \rangle.
\]
For completeness we mention that the cumulants $\kappa(X_1^{m_1} \ldots X_n^{m_n})$ are defined as

$$\log G(k_1, \ldots, k_n) = \sum \frac{(ik_1)^{m_1} \ldots (ik_n)^{m_n}}{m_1! \ldots m_n!} \kappa(X_1^{m_1} \ldots X_n^{m_n}), \quad (3.56)$$

where the symbol $\sum'$ indicates that we do not have to sum when all $m$ vanish. As an example we give the $n \times n$ covariance matrix $\kappa(X_i, X_j)$

$$\text{Cov}(X_i, X_j) = \langle(X_i - \langle X_i \rangle)(X_j - \langle X_j \rangle)\rangle \quad (3.57)$$

$$= \langle X_i X_j \rangle - \langle X_i \rangle \langle X_j \rangle. \quad (3.58)$$

The diagonal elements of the covariance matrix are, of course, the variances, whereas the off–diagonal elements are called the covariances. With the help of the covariance matrix it is possible to define a correlation coefficient

$$\rho_{ij} = \frac{\kappa(X_i, X_j)}{\sqrt{\kappa(X_i^2) \kappa(X_j^2)}}. \quad (3.59)$$

For $n = 2$ the statistical independence of $X_1$ and $X_2$ can be expressed through one of the following criteria:

(i) All moments factorize, i.e., $\langle X_1^{m_1} X_2^{m_2} \rangle = \langle X_1^{m_1} \rangle \langle X_2^{m_2} \rangle$.

(ii) The characteristic function factorizes, i.e., $G(k_1, k_2) = G(k_1)G(k_2)$.

(iii) All cumulants $\kappa(X_1^{m_1} X_2^{m_2})$ vanish when both $m_1$ and $m_2$ differ from zero. Two variables $X_1$ and $X_2$ are called uncorrelated if their covariance is zero. This condition is weaker than statistical independence.

A typical example of a multivariate density is the density of the multivariate Gaussian distribution

$$p(\vec{X}) = \frac{(2\pi)^{n/2}}{(\det A)^{1/2}} \exp\left(-\frac{1}{2}(\vec{X} - \vec{\mu})^T(A^{-1})(\vec{X} - \vec{\mu})\right), \quad (3.60)$$

where $A$ is a symmetric, positive definite matrix with elements $A_{ij}$. It is straightforward to check that the mean value of $\vec{X}$ is given by

$$\langle \vec{X} \rangle = \vec{\mu}, \quad (3.61)$$

that the covariance matrix is given by

$$\text{Cov}(X_i, X_j) = A_{ij}, \quad (3.62)$$

and that the generating function is

$$G(\vec{k}) = \exp(-\frac{1}{2}k_i A_{ij} k_j + i\mu_k k_i). \quad (3.63)$$

### 3.3 The Random Variables Transformation Theorem

We will discuss in this subsection a very helpful theorem by Gillespie. The proof of the theorem can be found in the book by Gillespie Gillespie [1992] or in his paper Gillespie [1983].

We know already that a stochastic variable is defined by specifying its space of states and its probability density. Here, we consider the $n$–dimensional random variables $X = (X_1, \ldots, X_n)$ which are specified by their joint probability density function $P(x_1, \ldots, x_n)$. Let $f_i$ be functions of the $n$ variables. With the help of the $f_i$ we map the $n$ random variables $X_1, \ldots, X_n$ onto $m$ new random variables $Y_1, \ldots, Y_m$ by

$$Y_i = f_i(X). \quad (3.64)$$

The random variable transformation theorem now states that the probability density of the new stochastic variable $Y$ is given by the expression

$$P(Y_1, \ldots, Y_m) = \int dx_1 \ldots dx_n \prod_{i=1}^m \delta(y_i - f_i(x_1, \ldots, x_n)) P(x_1, \ldots, x_n). \quad (3.65)$$

The integrals extend over the range of all $X_i$. For a proof of the random variable transformation theorem see Gillespie.
### 3.3.1 The Addition of Stochastic Variables

As a first simple example of the application of the random variable theorem we consider the addition of two stochastic variables $X_1$ and $X_2$ with joint probability density $P(x_1, x_2)$. The probability density $P(Y)$ of a new stochastic variable $Y$ which is defined as the sum of $X_1$ and $X_2$

$$Y = X_1 + X_2$$

is then given by

$$P(y) = \int \int \delta(x_1 + x_2 - y)P(x_1, x_2)dx_1dx_2.$$  \hspace{1cm} (3.66)

We can perform the integration over $x_1$ to obtain

$$P(y) = \int P(y-x_1)dx_1.$$  \hspace{1cm} (3.67)

For the special case of two statistically independent random variables $X_1$ and $X_2$ the above equation simplifies to the following expression

$$P(y) = \int P_{X_1}(x_1)P_{X_2}(y-x_1)dx_1.$$  \hspace{1cm} (3.68)

It is now easy to check that the following equations hold for the mean value and for the variance of the new stochastic variable $Y = X_1 + X_2$

$$\mu(X_1 + X_2) = \mu(X_1) + \mu(X_2)$$  \hspace{1cm} (3.70)

and

$$\text{Var}(X_1 + X_2) = \text{Var}(X_1) + \text{Var}(X_2) + 2\text{Cov}(X_1, X_2).$$

The last equation implies that only for uncorrelated stochastic variables we have the simple relation $\text{Var}(X_1 + X_2) = \text{Var}(X_1) + \text{Var}(X_2)$.

The above results for the mean values and variances can easily be generalized to the so called linear combination theorem. For any set of random variables $X_1, \ldots, X_n$ and any set of constants $a_1, \ldots, a_n$ we have

$$\mu \left\{ \sum_{i=1}^{n} a_iX_i \right\} = \sum_{i=1}^{n} a_i\mu(X_i)$$

and

$$\text{Var} \left\{ \sum_{i=1}^{n} a_iX_i \right\} = \sum_{i=1}^{n} a_i^2\text{Var}(X_i) + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} a_i a_j \text{Cov}(X_i, X_j).$$

### 3.3.2 One-to-One Transformations

Let us consider the following application of the random variables transformation theorem. Let $X$ be a random variable with probability density $P(x)$ and let the random variable $Y$ be defined as $Y = f(X)$. Then the density function of $Y$ is given by

$$P(y) = \int_{-\infty}^{\infty} dx P(x)\delta(y - f(x)).$$  \hspace{1cm} (3.71)
3.3. THE RANDOM VARIABLES TRANSFORMATION THEOREM

For the case that the function \( f \) is a smooth, one-to-one transformation the equation \( y = f(x) \) can be solved uniquely for \( x \) as \( x = f^{-1}(y) \). Let us now change the integration variable in Eq. (3.71) from \( x \) to \( z = f(x) \)

\[
P(y) = \int_{-\infty}^{\infty} dz \, f^{-1}(z) P(f^{-1}(z)) \delta(y - z),
\]

where we have made use of

\[
dx = \left| f^{-1}(z) \right| dz.
\]

Integrating over \( z \) yields

\[
P(y) = P(f^{-1}(y)) \left| f^{-1}(y) \right|.
\]

Since \( x = f^{-1}(y) \), then \( |dx/dy| = |f^{-1}(y)| \), and we can rewrite the above equation as

\[
P(y) = P(x) \left| \frac{dx}{dy} \right|
\]

which is a very important formula in the theory of stochastic variables which we will use, e.g., in the next chapter.

The previous result is easily generalized to many dimensions. If the transformations \( Y_i = f_i(X_1, \ldots, X_n) \) for \( i = 1, \ldots, n \) are one-to-one, then the probability density of the variables \( Y_i \) is given by

\[
P(y_1, \ldots, y_n) = P(x_1, \ldots, x_n) \left| \frac{\delta(x_1, \ldots, x_n)}{\delta(y_1, \ldots, y_n)} \right|
\]

BEMERKUNG: ANALOGIE ZUR ANALYSIS !!!!!!!

3.3.3 The Central Limit Theorem

As an essential application of the random variable transformation theorem we prove the central limit theorem. Let us consider \( N \) statistically independent random variables \( X_i \) with the same probability density \( P \) and, of course, the same mean \( \mu \) and the same variance \( \sigma^2 \). With the help of the \( X_i \) we define a new random variable \( Z_N \) as

\[
Z_N = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} (X_j - \mu).
\]  

(3.72)

Since the \( X_i \) are assumed to be mutually statistically independent their joint probability density is \( P(x_1) \cdots P(x_N) \).

By the random variable transformation theorem the probability density of \( Z_N \) is given by

\[
P(z_N) = \int dx_1 \cdots dx_N P(x_i) \delta \left( z_N - \frac{1}{\sqrt{N}} \sum_{j=1}^{N} (X_j - \mu) \right).
\]  

(3.73)

Using the integral representation of the \( \delta \)–function

\[
\delta(x - x_0) = \frac{1}{2\pi} \int ds \exp[is(x - x_0)]
\]  

(3.74)

the \( \delta \)–function in Eq. (3.73) can be written as

\[
\delta \left( z_N - \frac{1}{\sqrt{N}} \sum_{j=1}^{N} (X_j - \mu) \right)
\]  

(3.75)

\[
= \frac{1}{2\pi} \int_{-\infty}^{\infty} ds \exp(is\sqrt{N} \sum_{j=1}^{N} (X_j - \mu)) \exp[-isN^{-1/2}(x - \mu)].
\]  

(3.76)
Inserting the above equation into Eq. (3.73) and changing the order of the \( x \) and \( s \) integrations we obtain
\[
P(z_N) = \frac{1}{2\pi} \int ds \exp(isz_N)
\times \prod_{j=1}^{N} dx_j P(x_j) \exp[-isN^{-1/2}(x_j - \mu)].
\] (3.77)

The above expression can be written more concisely in the form
\[
P(z_N) = \frac{1}{2\pi} \int ds \exp(isz_N)[G(s/\sqrt{N})]^N,
\] (3.78)
where we have introduced the characteristic function \( G \) by
\[
G(\chi) \equiv \int dx P(x) \exp[-i\chi(x - \mu)].
\] (3.79)

The function \( G \) can be expanded in a Taylor series
\[
G(\chi) = G(0) + \chi G'(0) + \frac{\chi^2}{2} G''(0) + O(\chi^3).
\] (3.80)

Since the \( n \)-th derivative of \( G \) is given by
\[
G^{(n)}(\chi) = \int dx P(x) [-i \chi(x - \mu)]^n \exp[-i\chi(x - \mu)],
\] (3.81)

For \( \chi = 0 \) the above expression evidently reduces to
\[
G^{(n)}(0) = (-i)^n \langle (X - \mu)^n \rangle.
\] (3.82)

In particular we find \( G(0) = 1, G'(0) = 0 \) and \( G''(0) = -\sigma^2 \). Therefore, Eq. (3.80) can simply be written as
\[
G(\chi) = 1 - \frac{\sigma^2 \chi^2}{2} + O(\chi^3).
\] (3.83)

Inserting the above equation into Eq. (3.78) and putting \( \chi = s/\sqrt{N} \) we get
\[
P(z_N) = \frac{1}{2\pi} \int ds \exp(isz_N) \left( 1 - \frac{\sigma^2 s^2}{2N} + O\left(\frac{s^3}{N^{3/2}}\right) \right)^N.
\] (3.84)

In the limit \( N \rightarrow \infty \) we have, of course,
\[
\lim_{N \rightarrow \infty} \left( 1 - \frac{\sigma^2 s^2}{2N} \right)^N = \exp\left(-\frac{\sigma^2 s^2}{2}\right).
\] (3.85)

Therefore, for sufficiently large \( N \) we can write Eq. (3.84) as
\[
P(z_N) \approx \frac{1}{2\pi} \int ds \exp(isz_N) \exp\left(-\frac{\sigma^2 s^2}{2}\right).
\] (3.86)

The above integral can easily be evaluated with the help of the formula
\[
\int_{-\infty}^{\infty} dx \exp(ibx) \exp(-a^2 x^2) = \frac{\pi^{1/2}}{|b|} \exp\left(-\frac{b^2}{4a^2}\right)
\] (3.87)

to give
\[
P(z_N) \approx \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{z_n^2}{2\sigma^2}\right).
\] (3.88)

Eq. (3.88) is the central limit theorem. It states that the random variable \( z_N \) asymptotically becomes a Gaussian distributed random variable with zero mean and variance given by \( \sigma^2 \). It is to be remarked that we have only assumed that the random variables \( X_i \) have mean \( \mu \) and variance \( \sigma^2 \). This is the reason for the foremost importance of the Gaussian distribution.
3.4. Examples

In this section, we will learn how to use random variables for a simple example of a Monte Carlo simulation: the discrete-time random walk.

Before we look at the mathematical and physical part of this section, it is time to look at file operations in Java. We want to use these to save all the data produced in the random walk program, which will be too much to store in memory, to the hard disk. Then it can be further analyzed and plotted by other specialized programs available (like GNUPlot, PVWave, Octave, etc.). You could of course also use the graphical capabilities of Java (Pplot) to get the figures, but often in simulations it is of great importance to read and write data to disk for later usage, maybe also in a Java program. Later on in 10.3.5 we will apply these methods for Symphony, a program to solve a problem in parallel.

A short comment upfront: to use file I/O in Java you have to use exception handling, but we still have not discussed it. Because it is still possible to understand I/O operations, we start with this. In the next section we will tell you how to handle and understand exceptions.

3.4.1 File Input/Output in Java

The Java I/O package is one of the largest and for the beginner one of the most difficult core package at first glance. But after a while you feel comfortable and it will be easy to use.

In figure 3.2 you can see the main classes of the package, which are all subclasses of the java.lang.object class. Except the Reader and Writer classes, all classes were already introduced in Java 1.0. In Java 1.1 the two additional classes just mentioned were added, because the InputStream and OutputStream classes are using 8 bit characters and the new classes use 16 bit unicode characters for international language support. In Java 2 only small changes have taken place and therefore you can still stick to the Java 1.1 standard, but should avoid using Java 1.0 I/O syntax if possible. This is also because the Java 1.1 Reader and Writer classes are faster than the old ones. Because File I/O almost always means handling strings, we have to discuss this also in more detail.

ASCII Files

But let us start with the most simple problem: We have an array of double values, for example the x coordinates of the discrete-time random walk discussed afterwards and want to save them in a file in the format time x-coordinate. To that end we need to open a file for writing, write the array in it and close the file.

You probably would tend to look at the File class for reading and writing files, but this class is only responsible for managing files, directories and paths. We can use it to create a name for our file to be created. For file access, you have to use the Reader and Writer classes. So our small sample program would look like:
Figure 3.2: Overview of the `io` package in Java 1.1 and related classes.

Figure 3.3: Connecting two streams together to get advanced functionality.
3.4. EXAMPLES

/** Simple File I/O Program */

import java.io.*;
import java.util.*;

public class FileWriteSimple {
    static Random rand = new Random();

    public static void main(String args[]) {
        String filename = "test.asc"; // set the filename

        double[] array = new double[100]; // array to save
        for (int i=0; i<100; i++) { // create a random array
            array[i] = rand.nextDouble();
        }

        try {
            FileWriter fout = new FileWriter(filename); // open a File stream
            BufferedWriter out = new BufferedWriter(fout); // use a buffer
            // OR BufferedWriter out = new BufferedWriter(new FileWriter(filename)); */

            String dummy;
            for (int i=0; i<100; i++) {
                // write the array
                dummy = Double.toString(array[i]) + "\n"; // convert to String
                out.write(dummy); // write to stream = file
            }
        }
        catch (IOException e) {}
    }
}

First of all you realize that instead of just using the FileWriter class we “pipe” it to a BufferedWriter, which buffers the outgoing data between the disk and the memory. This is always desirable and should therefore always be used. Otherwise you get a very bad performance. Because it is inconvenient to remember all this, we have written a convenience class in our simulation package. So you can have the same program, only lines 17-18 (or 19) have to be changed to (and you have to import the simulation package, see program FileWriteConvenience.java):

FileOut out = new FileOut(filename); // open a buffered File stream

You can use a second argument to this constructor, which is a boolean variable and indicates if the data should be appended to the already existing file or not. The full syntax of the method is therefore FileOut(String filename, boolean append).

In line 23 the double gets converted to a string before it is written to the file. The actual writing of the data to the file is done in line 24 with the help of the write(String s) method in the Writer class.

Now we want to read the contents of the file back in. We use the classes FileReader and BufferedReader as you may have already guessed. The whole program looks like:

/** Simple File I/O Program */

import java.io.*;
import java.util.*;

public class FileReadSimple {
    public static void main(String args[]) {
        String filename = "test.asc"; // set the filename

        double[] array = new double[100]; // array to load
try {
    FileReader fin = new FileReader(filename); // open a File stream
    BufferedReader in = new BufferedReader(fin); // use a buffer
    /* OR BufferedReader in = new BufferedReader(new FileReader(filename)); */
    String dummy;
    int i = 0;
    while ((dummy = in.readLine()) != null) { // read the array
        System.out.println(Double.valueOf(dummy).doubleValue()); // convert to double
    }
    in.close(); // close the stream and file
} catch (IOException e) {}
}

Again you can use the convenience class FileIn from the simulation package, changing again the lines 17-18 (or 19) by:

FileIn in = new FileIn(filename); // open a File stream

To read data we call the readLine() method of the Reader class in line 18. We have used a shortcut notation here to have an assignment and a if clause in one line.

Important remark:
Java always overwrites old files without notification as the default behaviour! If you want to check if an old file already exists under this name, you have to do that by yourself. If you want to append the new data to the old one, use the second argument when creating a new FileWriter instance.

Using a StringBuffer  In the above examples and in all the System.out.println() methods, we make use of the concatenation operator +. So here seems to be the right place to discuss the fundamentals of string construction a bit more in detail.

Often you are in the need of constructing a string from scratch, by putting lots of small pieces (strings) together. Actually the same problem as concatenating strings for (formatted) output. You could always use the Lava Rocks methods, here especially the sprintf() method explained in the next paragraph and in section 10.3.5. But here we want to discuss how this can be implemented in core Java.

The basic feature enabling string concatenation is the concept of a StringBuffer. This is a standard class in the java.lang package and contains methods to manipulate strings. The String class itself (also in the java.lang package) provides basic functionality for fixed (non-changing) strings. As I already mentioned every time you use the concatenation operator, you are implicitly using the StringBuffer methods.

If you want to create a string, which contains all the output you want to write to a file in one string, you could use a StringBuffer. Here is the same class as above for writing a double array to a file, but this time we use the StringBuffer to create a string containing all the output and write this string to the file.

/** StringBuffer Demonstration class */

import java.io.*;
import java.util.*;
import simulation.*;

public class StringBufferDemo {
    public static void main(String args[]) {
        Random rand = new Random();
        String filename = "testbuff.asc"; // set the filename

3.4. EXAMPLES

```java
double[] array = new double[100]; // array to save
for (int i=0; i<100; i++) { // create a random array
    array[i] = rand.nextDouble();
}
```

```java
try {
    FileOut out = new FileOut(filename);
    // create a StringBuffer with length 5000 */
    StringBuffer buf = new StringBuffer(5000);
    for (int i = 0; i < 100; i++) {
        buf.append(arr[i]).append("\n"); // concatenate
    }
    out.write(buf.toString()); // convert to String and write
    out.close(); // close the stream and file
} catch (IOException e) {}
```
CHAPTER 3. STOCHASTIC VARIABLES

```java
FileOut out = new FileOut( filename ); // open a buffered File stream

PrintfFormatString fmt = new PrintfFormatString( "%5.2f %10.6f \n" );
for ( int i=0; i<100; i++ ) { // write the array and time
    Stdio.fprintf( out, fmt, new Object [] { new Double( (i/10.0) ),
        new Double( array[i] ) } );
}
out.close(); // close the stream and file
```

Binary Files We are now familiar with writing ASCII files, but sometimes it is necessary to write or read files in binary format, where the values are stored in the way they are kept in memory of the computer. This makes it faster and most of the time it produces smaller output files than using ASCII output.

The difference to other languages, where you have these two choices too, is that in Java on all platforms the binary format is the same, when it is saved. So here you do not have to mind using binary data files on different machines and operating systems as long as you stick to Java.

All of the basic I/O classes implement writing bytes to a stream, so you could actually use them. But there are two classes, which already implement the conversion of the different data types to a series of bytes and write these bytes to the desired stream. The first one is the DataInputStream class and the second one is the DataOutputStream class. They implement methods like writeDouble(), readDouble(), writeBoolean(), readBoolean(), etc. There are also methods for writing/reading strings (write()/readLine()) in these two classes, but avoid them, because they do not convert the byte stream correctly to character streams. Instead use the above discussed Writer and Reader classes.

As an example we look at the same example as above and write 100 double variables to a file, so the file will be 800 bytes long (on all computers and operating systems).

```java
/** StringBuffer Demonstration class */

import java.io.*;
import java.util.*;
import simulation.*;

public class FileBinary {
    public static void main(String args []) {
        Random rand = new Random();
        String filename = "test.bin"; // set the filename

        double [] array = new double [100]; // array to save
        for ( int i=0; i<100; i++ ) { // create a random array
            array[i] = rand.nextDouble();
        }

        try {
            DataOutputStream out = new DataOutputStream( new FileOutputStream( filename ) );

            for ( int i=0; i<100; i++ ) {
                out.writeDouble( array[i] );
            }
            out.close(); // close the stream and file
        }
    }
```
Handling Files – The File Class

Sometimes you need to delete, rename or copy files or must check if a file is already there or you have to seek in directories. For all these problems you have to refer to the File class (since Java 1.1) of the java.io package. Take a look at the extensive list of methods available in this API and you will certainly find something suitable.

As a short example we want to discuss how to check if a file with a certain filename is already available and how to handle directories and files therein.

```java
import java.io.*;

public class FileCheck {
    public static void main(String args[]) {
        if (args.length == 0) {
            System.out.println("Please supply a filename/directory!");
            System.exit(1);
        }

        String filename = args[0]; // set the filename
        File file = new File(filename);

        /* Check for existence of file */
        if (file.exists() == true) {
            System.out.println("File\n"+filename+" exists!");
        } else {
            System.out.println("\nFile\n"+filename+" does not exist!");
        }

        /* Check for existence AND not directory of a file */
        if (file.isFile() == true) {
            System.out.println("\nFile\n"+filename+" exists!");
        } else {
            System.out.println("\nFile\n"+filename+" does not exist or \n"+"is a directory!");
        }

        /* Check for existence AND not directory of a file */
        if (file.isDirectory() == true) {
            System.out.println("\nFile\n"+filename+" is a directory!");
            String[] filelist = file.list();
            for (int i=0; i<filelist.length; i++) {
```
CHAPTER 3. STOCHASTIC VARIABLES

```java
    System.out.println(filelist[i]);
  }
  else {
    System.out.println("File"+filename+" is not a directory!");
  }
}

Direct File Access  Since Java 1.1 there is also a class called RandomAccessFile, which is for creating and accessing files directly. This means you can store kind of data records and position the file exactly to a certain record instead of reading the whole file sequentially. Because in scientific applications this kind of file access is rather seldom, we do not discuss this feature. You have to consult the online API documentation for details.

Redirecting Standard Input and Output  You can redirect the standard output, input or error to a file for example or whatever you like (since Java 1.1). This is of great use, because often you run long simulations and want to save the standard output or error to a file for later use. This can be done easily on UNIX systems, but still you have to do it manually. In Java you just use methods from the java.lang.System class. For example to redirect the standard output and error to a file called Standard.out of a program, use:

```java
    import java.io.*;

    public class RedirectStandard {
        public static void main(String args[]) {
            try {
                PrintStream out = new PrintStream(
                        new BufferedOutputStream(new FileOutputStream("Standard.out")));
                System.setOut(out);
                System.setErr(out);

                System.out.println("This must go to the file!");

                out.close(); // !!! DO NOT FORGET
            } catch (IOException e) {
            }
        }
    }
```

The similar method for the standard input is called System.setIn(InputStream).

Compressing Files  File compression is not only important for transferring files or applets across the internet, it is also of concern for many scientists, who have large data sets and have to write these sets to disk for postprocessing.

Since the advent of the zlib compression library\(^1\) freely available under the GNU license, it is pretty easy to use compression in your own programs in Fortran or C. But in Java the compression library is already made available in the standard Java API java.util.zip. As the name suggests not only the zlib compression algorithm used in the famous GZIP program – mostly used on UNIX systems using the .gz ending – but also the algorithm of the famous zip program – mostly used on Windows based systems and has the ending .zip – is available in Java directly.

The only drawback is that it is implemented on top of the old Java 1.0 I/O API and sometimes you have to mix the old Java 1.0 and the new Java 1.1 I/O syntax.

\(^{1}\)http://www.zlib.org
The most easy interface is the GZIP one, so for most applications you should use GZIP. The Zip interface allows for multiple program compression in one file, but it is more cumbersome to use. Because most of the time you have to compress a single stream of data, we explain only the GZIP interface.

Let us discuss the problem of saving an array to a file again, but this time we want to store them compressed as a GZIP file.

```java
/** Writing a compressed GZIP file */

again there is a deprecation warning in Java 1.1, which vanishes using Java 2.

import java.util.*;
import java.util.zip.*;
import java.io.*;
import simulation.*; // Convenience

public class GZIPSaveArray {
    public static void main(String args[]) {
        /* create an array to save */
        double[] array = new double[1000];
        Random rand = new Random();
        for (int i = 0; i < array.length; i++) {
            array[i] = rand.nextDouble();
        }

        /* create a binary GZIP output file */
        try {
            /* DataOutputStream.gzout = new DataOutputStream
             * (new GZIPOutputStream
             * (new BufferedOutputStream
             * (new FileOutputStream("test.bin.gz"))) ); */
            // Convenience:
            GZFileOutBin gzout = new GZFileOutBin("test.bin.gz");
            /* store the data in file */
            for (int i = 0; i < array.length; i++) {
                gzout.writeDouble(array[i]);
            }
        }
    }

    /* create an ASCII GZIP output file */
    try {
        /* PrintStream.gzout = new PrintStream
         * (new GZIPOutputStream
         * (new BufferedOutputStream
         * (new FileOutputStream("test.asc.gz"))) ); */
        // Convenience:
        GZFileOutAscii gzout = new GZFileOutAscii("test.asc.gz");
        /* store the data in file */
        for (int i = 0; i < array.length; i++) {
            gzout.println(array[i]);
        }
    }
}
```

The first part stores it as a binary GZIP file `test.bin.gz` and you will see that it does not save
a lot of memory, because we are storing an array with random numbers, which is already (almost) best represented by the binary format. The second part produces an ASCII GZIP file `test.asc.gz`, which is much smaller than the uncompressed file. You can check this by using the GZIP utility and uncompress the files.

Reading of GZIP files works the same, you just have to use the `GZIPInputStream()` constructor and all the file I/O works like usual file handling.

We have also implemented convenience classes for these complicated structures to open a GZIP file for writing and reading. They are called `GZFileOutBin()` (`GZFileOutAscii()` and `GZFileInBin()` (`GZFileInAscii()`) and are used like the ones without compression. You can also take a look at the program `GZIPReadArray.java`.

---

**Reading Files into Applets**  Because of security reasons there is (almost) no way of reading files or writing files from within an applet. One way of doing it would be to use the security features of Java. In this case the browser has to tell the applet explicitly that it is allowed to do file I/O. This can be accomplished by signing an applet, prepared in a jar file. Then the browser can check the signature and if the browser setup allows file I/O for this signature it works.

Property files ?????

A second way would be to use the network features of Java and access a file from a FTP server, probably the same computer the applet was coming from. Of course there is still the problem of a password to be transmitted unsecurely to the FTP server, but it is fairly easy to setup. The code to use is a freely available program called `LinLyn.java`. We have included the code in our simulation package. The documentation is included in the Java file or you can study the simulation package documentation.

### 3.4.2 Exception Handling in Java

We have made extensive use of exceptions in the last section, but now we have to understand what we did there. Exceptions are “thrown” if an exceptional condition or error occurs in a part of a Java program. This can be a division by zero, a file not found error or something else, which the JVM does not know how to handle. So it is up to you to write code, which takes care of unusual conditions, which is called catching an exception – the most common situations are already caught by the standard Java implementation.

If a piece of code produces an exception it is checked if the code block surrounding the exception producing point is able of handling the exception. If not, the exception is propagated up to the next higher (calling) code block (method, class, etc.). If none of the code up to the main method catches the exception, an error message is shown on the command line and the program stops with a stacktrace showing the last commands executed. This propagation is a great simplification, because you can catch all of the exceptions in one place and not distributed through the whole class hierarchy.

The exception handling is located in the `java.lang.Throwable` class. Exceptions are objects and there are two types: Errors and Exceptions. Errors are usually not recoverable and the program has to exit with an error message. From exceptions you can recover, e.g. “Array out of Bounds”, “IO Exception”, etc. The exceptions, which are automatically caught are the `RuntimeExceptions`.

The way of handling exceptions is easily seen in the above examples about file I/O. You put the code, which can throw an exception in a `try { }` clause and use the `catch() {}` clause afterwards to define what has to be done, if an exception occurs in the try block. You can have many catch blocks, each for one special exception.

```java
try {
    // Code which throws exceptions.
} catch (java.lang.Throwable e) {
    // Code which catches/recover from exceptions.
} finally {
    // Code which should be executed, no matter if an exception
    // occurred or not, before global catching of the exceptions
    // takes place.
}
```
The `java.lang.Throwable` keyword in the catch clause has to be exchanged by one of the exceptions in table 3.1.

<table>
<thead>
<tr>
<th>Exception name</th>
<th>RTE</th>
<th>Occurrence/Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOException</td>
<td>N</td>
<td>Some IO problem (has lots of subclasses).</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>Y</td>
<td>array is out of bounds</td>
</tr>
<tr>
<td>ArithmeticException</td>
<td>Y</td>
<td>Integer division by zero</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Y</td>
<td>If a null instead of an object is used</td>
</tr>
<tr>
<td>NegativeArraySizeException</td>
<td>Y</td>
<td>If you try to create a negative sized array</td>
</tr>
</tbody>
</table>

Instead of explicitly catching exceptions you can also just write `throws IOException` after the class definition, before the actual code. Then the exceptions are handed to the next higher instance and if there is no calling class anymore an error is displayed. With this you can propagate all your exceptions to the parent class of all of the classes and catch the exceptions there.

If you ask how do you know what exceptions to catch (if some of the code is not written by you or is very complex), there are only two possibilities: read the whole code and look for exceptions, which can be thrown by it or wait for the exceptions to occur on the standard output, when running the code.

You can even use your own exceptions very easily by using the keyword `throw` and your own name of an exception. That is an easy way of handling exceptional conditions in your own code, like energy turns zero or step size/accuracy gets too large, e.g.

```java
try {
    energy = newEnergy(energy);
    if (energy == 0) {
        throw new EnergyZeroException();
    }
} catch (EnergyZeroException e) {
    // do something about zero energy
} catch (OtherExceptions e) {
    // maybe another exception to be caught
} finally {
    // Always do something here, even if a certain exception
    // has been thrown, but not catched yet
}
```

Although this seems to be a nice way to catch many special occurrences in a Java program, it should be restricted to exceptional conditions and not used as a programming tool. This is because exception throwing is a time consuming task and is not suitable for algorithmic designs.

### 3.4.3 The Discrete–Time Random Walk

Now we come back to the discussion of stochastic variables and take a look at a historical example.

A drunkard leaves a pub. His house is at the end of a straight street. Each time he moves he walks one step to the direction of his home or one step in the opposite direction with equal probability. The question we want to consider in this subsection is: What is the probability for the drunkard to be at home in \( r \) steps?

In a more formal language we want to associate with each step a stochastic variable \( X_i \) \( (i = 1, \ldots, r) \) assuming only the values \(+1\) and \(-1\) with probability \( 1/2 \) each. If he starts at \( n = 0 \), all possible positions are integers \( -\infty < n < \infty \). The position after \( r \) steps will be

\[
Y = X_1 + X_2 + \cdots + X_r.
\]
It is easy to check that \( \langle Y \rangle = 0 \) and since the steps are mutually independent

\[
\langle Y^2 \rangle = r \langle X^2 \rangle = r.
\]

The above relation expresses the very typical behaviour of a diffusive process: The mean squared displacement is proportional to the number of steps. To put it differently the variance of the mean of the velocity tends to zero for long times

\[
\left\langle \left( \frac{Y}{r} \right)^2 \right\rangle = \frac{1}{r} \to 0 \quad \text{as} \quad r \to \infty.
\]

In order to find the probability distribution of \( Y \) we make use of the characteristic function

\[
G_Y(k,r) = [G_X(k)]^r = \left[ \frac{1}{2} \exp(ik) + \frac{1}{2} \exp(-ik) \right]^r.
\]

The probability that \( Y \) has the value \( n \) is the coefficient of \( \exp(ink) \)

\[
p_n(r) = \frac{1}{2} \left( \frac{r}{2n} \right).
\]

In order to make clearer these concepts we want to write a program to simulate the random walker in discrete time steps. The program is called \texttt{rwdt} and its listing can be seen below.

**Listing of the program \texttt{rwdt.m}**

```matlab
% rwdt - Program to simulate a one-dimensional random walk % with discrete time steps.
clear; help rwdt;
n = input( 'Number of time steps (1000) - ');
nreal = input( 'Number of realizations (100) - ');
% Compute stochastic steps
x = sign(rand(n,1)-0.5);
y(1)=0;
for i=1:n-1
  y(i+1)=y(i)+x(i);
end
plot(y,’x’);
title(’One realization of the random walk’);
xlabel(’number of time steps’);
ylabel(’dislocation y’);
```

In the program we have obviously to generate an integer valued random variable which can assume with equal probability the values +1 and -1. One way of generating an \( n \)-dimensional vector \( x \) of such random numbers is

\[
x = \text{sign}(\text{rand}(n,1)-0.5)
\]

We run the program for 1000 steps, i.e., we choose the parameter \( nstep = 1000 \). Two realizations of the one-dimensional random walk can be seen in Figs. (3.4.3) and (3.4.3).

In order to check the theoretical prediction that the mean square displacement is proportional to the number of steps we generalize the program \texttt{rwdt} to allow for the generation of more realizations and the estimation of the mean value and variance. The new program is called \texttt{rwdtn} and generates \( nreal \) realizations of the stochastic process. Its listing can be seen below.
3.4. EXAMPLES

Figure 3.4: One realization of a one–dimensional random walk.

Figure 3.5: Another realization of a one–dimensional random walk.
Distribution of the end points of the random walk

**Figure 3.6:** The distribution of the end–points of the one–dimensional random walk. The program `rwdtn` was run for `nstep=100` and `nreal=1000`.

**Listing of the program `rwdtn.m`**

```matlab
% rwdtn — Program to simulate a one-dimensional random walk
% with discrete time steps. Ensemble average over nreal realizations

clear; help rwdt;
n = input ('Number of time steps (100–1000) — '); 5
nreal = input ('Number of realizations (100–1000) — '); 10

% Compute stochastic steps
for ir = 1:nreal
    x = sign(rand(n,1)–0.5);
y(1)=0;
    for i = 1:n-1
        y(i+1)=y(i)+x(i);
    end
    ym(ir) = y(n);
end
mean = sum(ym)/nreal ; 15
var = sum(ym.^2)/nreal ;
fprintf ('The mean value of y is %g and the variance is %g \n', mean, var)
fprintf ('The predicted values are: mean = %g, variance = %g \n', mean, var)
hist (ym,20);
title ('Distribution of the end points of the random walk'); 20
```

We run the program for `nstep = 100` and `nreal = 1000`. The estimated mean value of 0.274 and the estimated variance of 103.304 are in quite good agreement with the theoretical expected values of 0 and 100, respectively. It is interesting to look also at the distribution of the end points of the random walk. This can be seen in Fig. (3.4.3).
3.4.4 Generation of Gaussian Random Numbers

As a simple demonstration of the central limit theorem we want to generate Gaussian distributed random numbers by adding uniformly distributed ones.

We know that uniformly distributed random numbers on the interval \([0, 1]\) have \(p(x) = 1\) for \(x \in [0, 1]\). Then it is easy to show that

\[
\langle X \rangle = \frac{1}{2}
\]

and

\[
\text{Var}(X) = \int_0^1 x^2 dx - \langle X^2 \rangle = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}.
\]

Now let us consider the transformed random variable \(X'\)

\[
X' = (X - \frac{1}{2}) \sqrt{12}\sigma
\]

which has mean 0, variance \(\sigma^2\), and is uniformly distributed on the interval \([-\frac{1}{12}\sqrt{12}\sigma, \frac{1}{12}\sqrt{12}\sigma]\). Let us now draw \(N\) such random numbers \(X'_1, \ldots, X'_N\) and let us construct the new stochastic variable \(Z\)

\[
Z = \frac{1}{\sqrt{N}}(X'_1 + \ldots + X'_N).
\]

Then the central limit theorem states that the variable \(Z\) is a Gaussian variable with mean 0 and variance \(\sigma^2\).

With the help of the program `cltgen` we want to demonstrate that already for \(N = 12\) we get Gaussian distributed random numbers in a very good approximation.

Listing of the program `cltgen.m`

```matlab
% cltgen - Program to generate Gaussian distributed random numbers
clear; help cltgen;
n = input('Enter number of random numbers z (1000) - ');
for i = 1:n
    x = (rand(12,1)-0.5) / sqrt(12);
    z(i) = sum(x) / sqrt(12);
end
hist(z,20)
title ('Distribution of the generated Gaussian random numbers');
```

In Fig. (3.4.4) we see the distribution of the Gaussian random numbers generated with the help of the program `cltgen`. The number of random numbers \(Z\) was chosen to be 1000.

3.4.5 Estimation

*Experimental data are random numbers!* An experiment provides realizations of some random variable \(X\). We call an \(N\)-fold realization of \(X\) a sample of size \(N\). It is of fundamental importance to distinguish between the estimate for the mean and the variance made on the basis of the sample, which we will denote by \(m\) and by \(s\), respectively and the corresponding quantities for the (infinite) underlying population, the ensemble.

Of course, estimates should be unbiased, i.e., for very large samples the estimate based on the sample size \(N\) should converge to the ensemble averages.
Mean Values

Let us consider \( N \) copies \( X_1, \ldots, X_N \) of a random variable \( X \) and let us build the new stochastic variable

\[
Z = \frac{1}{N} (X_1 + \cdots + X_N).
\]

\( Z \) is the mean value of the \( N \) realizations. Assuming that the \( X_i \) are uncorrelated we obtain for the cumulants of \( Z \)

\[
\kappa_n(Z) = \frac{1}{N^n} \sum_{i=1}^{N} \kappa_n(X_i) = N^{-n+1} \kappa_n(X).
\]

In particular we have

\[
\kappa(Z) = \langle Z \rangle = \langle X \rangle
\]

\[
\kappa_2(Z) = \text{Var}(Z) = \frac{1}{N} \kappa_2(X) = \frac{1}{N} \text{Var}(X)
\]

\[
\kappa_3(Z) = O(N^{-2}) \quad \text{for} \quad N \geq 2.
\]

In other words the mean value of \( Z \) is a random variable with a distribution which has the same mean value as \( X \), but with a variance which is smaller by a factor of \( N \). Up to terms of the order \( O(N^{-2}) \) the distribution of \( Z \) is Gaussian.

Estimating Mean and Variance

Let us consider to have the sample \( x_1, \ldots, x_N \). A natural estimator of the mean value \( \mu \) is the sample mean

\[
m = \frac{1}{N} \sum_{i=1}^{N} x_i.
\]
An estimate for the variance could, in analogy, naturally assumed to be

\[ \hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - m)^2. \]

Unfortunately, the above estimator is biased, because we make use of the already known \( m \) instead of the unknown \( \mu \). As can easily be seen by adding and subtracting \( \mu \) in each term in the above equation we get

\[
\hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu - (m - \mu))^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 - 2(m - \mu) \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu) + (m - \mu)^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 - (m - \mu)^2 \text{.}
\]

Now, by taking expectation values averaging over an infinity of samples of size \( N \) we have

\[
E[\hat{\sigma}^2] = E \left[ \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \right] - E [(m - \mu)^2] = \sigma^2 - \sigma^2_m \text{.}
\]

If we assume, that there are no correlations we have \( \sigma^2_m = \sigma^2 / N \), an unbiased estimate of \( \sigma^2 \) is

\[
s^2 = \frac{N}{N - 1} \hat{\sigma}^2 = \frac{1}{N - 1} \sum_{i=1}^{N} (x_i - m)^2 \text{.}
\]

In computations, if the sample is large, rounding errors can be large because \( (x_i - m) \) is small. In these cases it is convenient to use the "corrected two–pass algorithm" for \( s^2 \)

\[
s^2 = \frac{1}{N - 1} \left\{ \sum_{i=1}^{N} (x_i - m)^2 - \frac{1}{N} \left[ \sum_{i=1}^{N} (x_i - m) \right]^2 \right\} \text{.}
\]

The function of the additional second term which would be identically equal to zero if \( m \) were exact is to correct the rounding errors of the first term Press et al. [1992a].

Confidence Levels

It is important to have also a quantitative characterization of the goodness of the estimation. To this end we assign to every estimation a certain confidence interval, which is to be chosen in such a way that the true value lies within this interval at some predetermined level of confidence. Since we know, by virtue of the central limit theorem, that the mean value is Gaussian distributed a criterion for the error can be directly derived from the geometric properties of the distribution. Assuming that the mean value is \( m \) and that the standard deviation is \( \sigma_m \) then the probability to find the true value in the interval \([m - \sigma, n + \sigma] \) is given by the surface under a normal probability density between \( \mu - \sigma_m \) and \( \mu + \sigma_m \)

\[
\text{Prob}(\mu \in [m - \sigma, n + \sigma]) = \int_{m-\sigma}^{m+\sigma} \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - m)^2}{2\sigma^2} \right) = 0.683 \text{.}
\]

Thus, in 68.3\% of samples a value lying within \( \pm \sigma_m \) of the population mean \( \mu \) would be found. Conversely, there is 68.3\% probability that the interval \([m - \sigma_m, m + \sigma_m] \) contains the population mean.

In general we have...
3.5 Beyond this Chapter

3.6 Exercises

**Exercise 3.1 Random-Number Generator Check [Knuth, 1981]**

To test the random number generator of Matlab, we calculate the first 10 moments of the distribution generated from `rand()`. Compare these with the exact results for a uniform distribution.

Plot a histogram to check for a uniform distribution.

Use the Poker-Test for testing `rand`: Create many series of 5 random numbers between 1 and 13. Then count the fractions of hands with two, three and four identical (numbers) cards. Compare the result with the predictions:

<table>
<thead>
<tr>
<th>hand</th>
<th>number of ways</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>all different (no pair)</td>
<td>1,302,540</td>
<td>0.5017739</td>
</tr>
<tr>
<td>two of a kind</td>
<td>1,098,240</td>
<td>0.42256903</td>
</tr>
<tr>
<td>three of a kind</td>
<td>54,912</td>
<td>0.021128451</td>
</tr>
<tr>
<td>four of a kind</td>
<td>624</td>
<td>0.000240096</td>
</tr>
<tr>
<td>Total number of possibilities</td>
<td>2,598,960</td>
<td>1</td>
</tr>
</tbody>
</table>

For a rigorous check you have to use the chi-squared test for your results. If you are interested, take a look at the book of D. Knuth.

**Exercise 3.2 Galton Board and Pascal Triangle [Whitney, 1990]**

Write a program to simulate a Galton Board on the computer.

That is a device where you introduce a ball at the top. The ball falls down towards the bottom, bouncing off the pins to the right or left at each level. The only random effect is the bouncing at the pins at each level. The probability of bouncing to the right or left is always 0.5. Therefore it is a simple model for a symmetric random walk in one dimension.

How can you extract the number of ways to get to one particular box at the end of the board out of the estimated probabilities above? What is the connection to the Pascal Triangle?

Change the program to simulate an asymmetric random walk in one dimension.

**Exercise 3.3 The Standard Deviation**

Compare the four possibilities to calculate the standard deviation (or the variance):

1. using the definition:

\[
\sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2, \quad \bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i,
\]

2. using the moments:

\[
\sigma^2 = <x^2> - <x>^2, \quad <x^2> = \frac{1}{N} \sum_{i=1}^{N} x_i^2.
\]
3. using the formula ([Scariano und Marlow, 1991]):

\[
\sigma^2 = \frac{1}{N^2} \sum_{i=1}^{N} (x_i - x_j)^2.
\]

4. using the corrected two-pass formula [Press et al., 1992b]:

\[
\sigma^2 = \frac{1}{N-1} \left\{ \sum_{i=1}^{N} (x_i - \bar{x})^2 - \frac{1}{N} \left( \sum_{i=1}^{N} (x_i - \bar{x}) \right)^2 \right\}.
\]

The first and the second method require the computation of the first or the first and the second moments. The third moment doesn’t require any precomputed values at all and the last one uses again only the first moment. The last one corrects for the roundoff errors, encountered when using large sample sizes. The last one is analytically exact only if \(\bar{x}\) would be exact.

Write a program including all four methods and compare the results. Find out which method the Matlab function \texttt{std} uses. Check the calculation with a uniform, a normal and a Cauchy (Lorentz) distribution. Can you find an example where the two-pass algorithm is superior to the other ones?

By the way, the variance is not the only value to estimate the spreading of a sample. Statisticians often use the estimate

\[
\text{ad}e\text{v} = \frac{1}{N} \sum_{i=1}^{N} |x_i - \bar{x}|
\]

as a measure for the distribution width around the mean value.

**Exercise 2** Random variable transformation theorem: (a) Consider the linear transform of \(X\): \(Y = bX + c\). (b) the log–normal distribution. Lit. Gillespie, Am. J. Phys. 51 (1983) 520. lichkeitstheoretische Grundbegriffe; typische Verteilungen (Poisson, Gauss, Binomial);
Bibliography


Chapter 4

Data Analysis

4.1 Estimation

4.2 Simple Monte Carlo Evaluation of Integrals

It is the purpose of this subsection to introduce Monte Carlo Methods in the context of the numerical evaluation of definite integrals. We will see in later chapters that Monte Carlo integration is the method of choice when treating multidimensional integrals numerically. As a typical rule of thumb “classical” deterministic methods are outperformed by Monte Carlo methods for systems with a large number of degrees of freedom. For simplicity and to stress the basic ideas it is convenient at the moment to consider one–dimensional definite integrals of the form

\[ I = \int_{a}^{b} dx f(x), \] (4.1)

Obviously such integrals can be evaluated analytically for many integrands \( f(x) \). However, there are as well many cases for which a numerical evaluation is necessary.

Before introducing the Monte Carlo approach to numerical integration let us remind the basic “classical” deterministic approach to numerical integration. The standard approach is based upon the geometrical interpretation of the integral (4.1) as the area under the curve of the function \( f(x) \) between the points \( a \) and \( b \). In the simplest algorithm this area (see figure) is approximated as a sum over rectangles. To this end the \( x \)--axis is divided into \( n \) equally spaced intervals of width \( \Delta x \),

\[ \Delta x = \frac{b-a}{n} \] (4.2)

whose ends are given by

\[ x_i = x_0 + i \Delta x \] (4.3)

for \( i = 1, \ldots , n \). Of course, \( x_0 = a \) and \( x_n = b \). Thus in the so–called rectangular approximation the integral is evaluated as

\[ I_n = \Delta x \sum_{i=1}^{n-1} f(x_i), \] (4.4)

Of course, other more accurate approximations are possible.

How can we now evaluate the above integral by drawing random numbers? The standard way is based on a very simple idea. From the introductory course in analysis we know that the Mean Value Theorem states that the exact value of the integral \( I \) is given by

\[ I = (b-a)f(\zeta) \] (4.5)
for some value of $\zeta$ in the interval $a \leq \zeta \leq b$. $f(\zeta)$ represents the average value of the function $f(x)$ in the interval $[a,b]$. Thus we could also write

$$I = (b - a) \langle f \rangle,$$  

the inverse of the area

(4.6)

where $\langle \rangle$ denotes the mean value. Let us draw $n$ random numbers which are uniformly distributed in the interval $[a,b]$ and let us sample the corresponding value of $f(x_i) = f_i$. The Monte Carlo estimate $I_n$ of the integral $I$ is then the sample mean, which is given by

$$I_n = \frac{(b - a)}{n} \sum_{i=1}^{n} f(x_i),$$  

(4.7)

where $n$ is the number of trials. Amazingly the form of the above estimate is very similar to the classical formula (4.4). The fundamental difference is that now the $n$ points at which the function $f$ is evaluated are no longer equally spaced but randomly distributed.

There is also the possibility to compute the integral $I$ stochastically with the help of the “Hit or Miss” algorithm. The idea behind this algorithm is again very simple. To be explicit we imagine a rectangle of height $h$ and width $(b - a)$ such that the function $f(x)$ lies within the rectangle (see figure; Gould, p.329). To evaluate the integral we draw randomly pairs of uniformly distributed random numbers $(x_i, y_i)$ such that $a < x_i < b$ and $0 < y_i < h$. In other words the probability to draw a point within the rectangle is given by the inverse of the area $A$ of the rectangle, i.e., $1/(b - a)h$. It is now evident how the area under the function $f$ may be estimated. The fraction of points $(x_i, y_i)$ which satisfy the condition $y_i \leq f(x_i)$ is an estimate of the ratio of the integral $I$ to the area $A$ of the rectangle. Hence, drawing $n$ random pairs the estimate $I_n$ of $I$ by this “scoring” method is given by

$$I_n = A \frac{n_s}{n},$$  

(4.8)

where $n_s$ is the number of “hits”, i.e., of points lying below the curve $f(x)$.

Before writing two simple programs to elucidate the above algorithms it is important to have in mind that both estimates are affected by statistical errors. Let us consider for simplicity the standard method. Since the $f(x_i)$ are random we know from the elementary theory of data analysis that an appropriate measure of the error is given by the variance which is defined by

$$\text{Var}(f) = \langle f^2 \rangle - \langle f \rangle^2 = \langle (f - \langle f \rangle)^2 \rangle.$$

(4.9)

Since we draw a finite number of random numbers we can estimate the mean value by using

$$\hat{f} = \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$  

(4.10)

and correspondingly the estimate of the variance by using

$$\text{Var}(f(x_1), \ldots, f(x_n)) = \frac{1}{n-1} \sum_{i=1}^{n} (f(x_i) - \hat{f})^2 = \sigma_f^2.$$  

(4.11)

The quantity $\sigma_f = \sqrt{\text{Var}(f_1, \ldots, f_n)}$ is also called the standard deviation. In the previous expression we have used the short-hand notation $f(x_i) = f_i$. However, we are not interested in the error of $f$ but in the error of the estimate $I_n$, which is a sum over random numbers.

Repeating the simulation and hence drawing other random numbers we will get another estimate of $I_n$. Therefore, repeating the simulation $m$ times we can estimate the mean of $I_n$ as

$$I = \frac{1}{m} \sum_{j=1}^{m} I_n(f)$$  

(4.12)
4.2. SIMPLE MONTE CARLO EVALUATION OF INTEGRALS

and the corresponding variance as

\[ \text{Var}(I_n(1), \ldots, I_n(m)) = \frac{1}{m-1} \sum_{j=1}^{m} (I_n(j) - \hat{I}_n)^2 = \sigma_f^2 \]  

(4.13)

We will denote the above variance also by \( \sigma_j^2 \). Of course, proceeding this way is not very practical since we have to perform the simulation \( m \) times. A much more economical estimation of the error of the mean of \( I_n \) could be achieved by establishing a simple relation between \( \sigma_j \) and the standard deviation of the individual trials \( \sigma_f \). To this end we introduce the discrepancy \( \delta f_i \) between the individual trial \( f_i \) and its mean \( \langle f \rangle \). The discrepancy \( \delta I_n \) between \( I_n \) and its mean value can be obtained to first order in the \( \delta f_i \) by a simple Taylor expansion (error propagation rules)

\[ \delta I_n = \sum_{i=1}^{n} \frac{\partial I_n}{\partial f_i} \delta f_i \]  

(4.14)

Hence, it follows from the above equation by taking the average over \( \delta I_n^2 \) that

\[ \langle \delta I_n^2 \rangle = \sum_{i,j=1}^{n} \frac{\partial^2 I_n}{\partial f_i \partial f_j} \langle \delta f_i \delta f_j \rangle \]  

(4.15)

It is plausible to assume, that \( \langle \delta f_i \rangle = 0 \) for all \( i \) and that the \( \delta f_i \) are not correlated for \( i \neq j \), i.e., \( \langle \delta f_i \delta f_j \rangle = \langle f_i \rangle \langle f_j \rangle \) and that for \( i = j \) we have \( \langle \delta f_i^2 \rangle = \sigma_f \) for all \( i \) it follows from the above equation that

\[ \sigma_j^2 = \sum_{i=1}^{n} \left( \frac{\partial I_n}{\partial f_i} \right)^2 \sigma_f^2 = \frac{1}{n^2} n \sigma_f^2 = \frac{\sigma_f^2}{n} \]  

(4.16)

and finally we have the useful relation

\[ \sigma_j = \frac{\sigma_f}{\sqrt{n}} \]  

(4.17)

The mean error of the mean scale with 1 over the square root of the number of individual trials. The precision of the estimate thus increases only slowly with the number of trials (remark: central limit theorem: see Chapter 2).

Now we are in the position to write two programs to implement the above stochastic algorithms. In order to be specific we compute the integral

\[ I = \int_{0}^{1} dx \sqrt{1-x^2} = \frac{\pi}{4}, \]  

(4.18)

In other words we want to estimate the number \( \pi \) by Monte Carlo methods.

We begin by the standard method. The listing of an according program can be seen below.

**Listing of the program mcpi.**

```plaintext
% mcpi - Program to estimate pi with the help of the
% standard Monte Carlo method
clear; help mcpi;
n = input( 'Enter number of random numbers n (< 16384)- ');
% Draw n random numbers
r = rand(n,1);
In = 0;
for i = 1:n
```
Figure 4.1: The estimation of pi for n=10,100,1000,10000. The error bars correspond to the standard deviation of the mean of the estimate.

\[
\frac{f(i)}{r(i)} = \sqrt{1 - r(i) \cdot r(i)}; \\
f^2(i) = f(i)^2; \\
In = In + f(i); \\
\text{end}
\]

\[
In = In / n; \\
varf = 0; \\
for \ i = 1:n \\
\quad varf = varf + (f(i) - ln)^2; \\
\text{end}
\]

\[
varf = varf / (n-1); \\
vari = varf / n;
\]

\[
fprintf( '
The MC estimate of pi is \%g \pm \%f \ n', 4 \cdot ln, sqrt(vari));
\]

We run the program for \( n = 10, 100, 1000, 10000 \). The result of the four simulations can be seen in Fig. xy.

Next we write a program for the scoring method.

Listing of the program mcpiscore.m

% mcpiscore — Program to estimate pi by the Monte carlo % scoring method

clear; help mcpiscore;
4.3 Beyond this chapter

4.4 Exercises

Use the Matlab function `rand()` to solve the following problems (don't care about the quality and the algorithm of the random number generator, for now):

```matlab
% Example code for the Monte Carlo estimation of pi
n = input('Enter number of random pairs to draw (<16394) - '); % Input number of random pairs
rx = rand(n,1); % Generate random x-values
ry = rand(n,1); % Generate random y-values
In = 0; % Initialize counter
for i = 1:n
    if (rx(i)*rx(i) + ry(i)*ry(i) <= 1) % Check if point is inside the unit circle
        In = In + 1; % Increment counter if inside
    end
end
In = In / n; % Calculate estimate of pi
plot(rx, ry, 'x') % Plot random points
hold on;
x = [0:0.01:1]; % Generate x-values for the function
y = sqrt(1-x^2); % Calculate y-values for the function
plot(x,y); % Plot the function
hold off;
fprintf('The MC estimate of pi is %g \n', 4*In) % Print the estimate
```

Again we run the simulation for n=10, 100, 1000, 10000. The result of a simulation can be seen in the next Figure.
**EXERCISE 4.1 Photoabsorption [Reif, 1967]**

Consider the absorption of photons passing through a gas in two dimensions. We model the gas by introducing slabs of width \(dx\) and density \(n\) (in particles per area), which absorb the incident photons. The slab particles have a cross-sectional area of \(\sigma\).

So the probability of a photon to be absorbed in the slab will be (\(M\) is the number of particles in the slab of the height \(dy\))

\[
P(\text{Photon absorbed}) = \frac{M\sigma}{dy} = \frac{\sigma ndx dy}{dy} = \sigma ndx.
\]

We have assumed that there is no overlap between the cross-sections of the slab particles.

Write a program to simulate this process on the computer. Take \(N\) incident photons and watch the number of particles left over against the slabs passed in a diagram. Do this simulation several times and calculate the ensemble-average. What process you know is similar to this behaviour and what takes the place of the spatial dimension in that case?

**EXERCISE 4.2 Monte-Carlo Integration – Speed and Accuracy**

Write a program for the calculation of the following integral:

\[
\int_{0}^{1} \frac{1}{1+x^2} dx.
\]

1. using the hit and miss method

2. using the standard method

For both algorithms, calculate the mean and the standard deviation as discussed in the lecture. Also use the analytical result of the integral to calculate \(\pi\). Compare the accuracy of both algorithms using the approximations of \(\pi\). Compare the speed of the two programs by using the `cputime` function in Matlab. (e.g., type the following to time the random number generator: \(t=cputime; x=rand(1000); cputime-t\))

To that end, create a table and a plot with the two parameters \(n\): the number of intervals and \(N\): the number of realizations) against the accuracy (use at least 5 values). To save time, you can first check for a good \(n\) and then do the plots only against \(N\). For the speed, plot the cputime against the achieved accuracy for many different \(N\).

**EXERCISE 4.3 Eulers Constant using Monte-Carlo Algorithm [Mohazzabi, 1998]**

Suppose throwing \(N\) darts randomly at a dart board, which has been divided into \(R\) equal size regions. The probability of hitting one region is \(p = 1/R\). Then the probability of hitting an empty region (not already occupied by a dart) is \((1-p)^N\). Using the binomial distribution, you can get the probability for hitting a region with \(m\) darts. If you choose the number of regions equal to the number of darts thrown on the board, we have \(p = 1/N\) and therefore

\[
P(\text{hitting an empty region}) = \left(1 - \frac{1}{N}\right)^N.
\]

Because the above series converges to \(e\) for \(N \to \infty\), we can use the following method to get an approximation of the Euler constant:

(i) Throw randomly a large number of darts (say \(N\)) on a board, which has been divided into \(N\) equal size regions.

(ii) Count the number of empty regions (call it \(N_0\)).

(iii) The fraction \(N_0/N\) is a good estimate of the Euler constant \(e\).

Write a program for that algorithm and check the results. You can even use \(N/N_1\), if \(N_1\) is the number of regions with the occupancy of one dart. Check this, too. What \(N\) do you need to get the same accuracy using the formula? And how many terms of the series for \(e\) \(\left(\sum_{i=0}^{\infty} 1/i! = e\right)\)?
Bibliography


Chapter 5

Simple Sampling of Probability Distributions Using Random Numbers

This Chapter is devoted to the following question: How can we generate sequences of random numbers which are distributed according to some given distribution?

A simple answer to this question would be to exploit some intrinsically random physical process. For example, one could record a sequence of the decay times of some radioactive substance and use this truly random sequence of numbers in a Monte–Carlo simulation. Although tables of millions of such true random numbers exist in practice this approach turns out to be very impractical. Monte–Carlo simulations need very long sequences of random numbers, so that we have to find more efficient ways to generate them. This requirement is satisfied by so–called pseudo–random numbers. Pseudo–random numbers are generated numerically with the help of some simple algorithm on some computer. Consequently, they are reproducible. This is, however, not a drawback. In fact, the reproducibility may be very useful if we want to check our simulation algorithms.

Pseudo–random numbers are, the name already underlines it, not truly random. However, their statistical properties are very similar to the statistical properties of truly random numbers. So, for all practical purposes pseudo–random numbers appear to be random. Let us now see how such pseudo–random numbers can be generated.

5.0.1 A Random Number Generator

Since the concepts we just introduced are quite abstract it may be useful to see them in action with the help of a second example. So let us write a class, which calculates random numbers uniformly distributed between 0 and 1. To this end we write a class RandomNumber. It has three fields (variables), two of them are defined with the final keyword.

```java
/** A simple and bad random number generator */

public class RandomNumber {
    /** These are the constants, defining ( quality of )
        the generator */
    private int a=65539;
    private long M=(int)Math.pow(2,31)-1;
    /** The seed has to be a class variable */
    private long R1;

    /** Constructor sets the seed of the generator */
    RandomNumber( int Seed ) {
        R1=Seed;
    }
```
/** here we draw the next random number */
public double nextRand () {
    R1=(long)(a*R1)%M;
    return (double)R1/(double)M;
}

/** set the parameters of the generator */
public void setParameters ( int a, long M ) {
    this.a=a;
    this.M=M;
}

/** Set the seed */
public void setSeed ( long seed ) {
    this.R1=seed;
}

/** return the parameters */
public int getA () {
    return this.a;
}
public long getM () {
    return this.M;
}

/** get the seed */
public long getSeed () {
    return this.R1;
}

In our example, we just set the seed in the constructor and as you can see you have to supply the seed, when you instantiate the generator. Most of the constructors don’t even need a parameter to instantiate the object.

In the program we have made use of the Math.pow() method, which is defined in the Math class of the standard Java API java.lang and it computes the power of the first argument to the second one (see section 1.13.1).

Additionally we have included some routines to show how private variables can be handled. Because there is no way for programs to access the fields of the class from the outside of this class, you have to supply methods to read or write the fields. This is actually the way you have to write Java beans discussed shortly in section 1.15.

Then the method nextRand() is defined and returns the next random number calculated using a congruential method¹.

Now we have to write a program (class), which uses this class to calculate the average of some random numbers.

/** A small test program, to test the Random Number class */
public class UseRandomNumber {
    public static void main ( String args [] ) {
        final int Seed=123;
        final int N=100000;
        double [] numbers = new double[N];

        // create a new Random Number Generator
        RandomNumber rand = new RandomNumber(Seed);

        for ( int i=0; i<N; i ++ ) {
            numbers[i]=rand.nextRand (); // draw random numbers
        }

        // Calculate the average
        double avg=0;

        ¹See later in this book for details about how to generate random numbers.
5.1. THE GENERATION OF UNIFORMLY DISTRIBUTED RANDOM NUMBERS

for (int j = 0; j < N; j++) {
    avg += numbers[j];
}
20
    avg /= (double)N;
System.out.println("The Mean is: "+avg);
}

The class is called UseRandomNumber and just contains the main method. First we instantiate (create) an object of class RandomNumber. Then we create an array and in a loop we create N random numbers using the nextRand() method of our class applied to the object we just created. The remaining part has already been explained.

Remember that we have to put both programs in one directory called simu, because we have used the package command. Here it was just to get you acquainted with these terms.

5.1 The Generation of Uniformly Distributed Random Numbers

We will begin with the generation of uniformly distributed random numbers on the interval $[0,1)$. In the following we will often omit the prefix pseudo.

The best known algorithm for the generation of uniformly distributed random numbers is the linear congruential method, which given an initial integer "seed" value $I_1$ produces random integers recursively using the formula

$$I_{n+1} = (aI_n + c) \mod M,$$

where $a$, $c$, and $M$ are integer constants which have to be chosen appropriately. The randomness of the above algorithm results from the fact that after some multiplications with $a$ the result exceeds $M$ and is consequently truncated. Since the integers $I_n$ lie between 1 and $M$ a random number $R$ uniformly distributed between 0 and 1 is obtained as

$$R = \frac{I_n}{M}.$$

Unfortunately, MATLAB does not have integer arithmetic so the above algorithm has to be implemented using the \texttt{rem} (remainder) function instead of the modulo function. A corresponding code could be

$$I(n+1) = \text{floor}(\text{rem}(a*I(n) + c, M)) .$$

The MATLAB function \texttt{floor} rounds towards minus infinity. In order to get familiar with this algorithm we want to generate a sequence of pseudo–random numbers for the following parameters: we choose the multiplier to be $a = 5$, the increment $c = 3$, and the modulo $M = 8$. Obviously the longest period of random numbers will have the length 8. The generation of the random sequences will be achieved with the program \texttt{trandom1}.

Listing of the program \texttt{trandom1.m}

```matlab
% trandom1 - Program to demonstrate the generation of random numbers
% using the linear congruential method
clear; help trandom1 % clear the memory and print header
seed = input('Enter the seed (1) - '); m = input('Enter the modulus (8) - ');
a = input('Enter the multiplier (5) - ');
c = input('Enter the increment (1<=c<7) - ');
% Set starting value
R(1) = seed;
```
The series of generated random numbers

Figure 5.1: Successive values in a series of random numbers generated for a=5, c=3, M=8. Note that the even numbers are always one less than the odd ones!

10  \% Generate vector of 2m random numbers
for  j=1:2*m
    R(j+1) = floor ( rem (a*R(j)+c,m) );
end
\%
R=R/M;
15 disp ('The generated series is :')
disp (R)
plot (R,'x')
title ('The series of generated random numbers ');
xlabel ('Term, i ');
ylabel ('Value ');

Run with the above parameters the program generates the sequence

1,0,3,2,5,4,7,6,1,0,3,2...

which has also been plotted in Fig. (5.1). It might be instructive to run the program keeping the multiplier \(a\) and the modulo \(M\) fixed while changing the increment \(c\). The result of these runs are summarized in Table 5.1.

It is evident that a wrong choice of the constants leads to a very poor random sequence.

It can be shown Knuth [1981] that in the case \(c=0\) the full period, 1 to \(M-1\) can be achieved by choosing \(M\) as a prime number and for \(a\), a primitive element modulo \(m\), i.e., for all prime divisors, \(p\), of \((M-1)\),

\[ a^{(M-1)}/p \mod M \neq 1. \]

For the case of \(c \neq 0\) the full period is obtained if the following three conditions are satisfied:
(i) \(c\) and \(M\) are relatively prime,
(ii) \(a \mod p = 1\) for each prime factor \(p\) of \(M\),
(iii) \(a \mod 4 = 1\) if 4 divides \(M\).

It is evident that the greater the modulus the longer the period. For example the MATLAB random number generating function uses

\[ a = 16807; c = 0; M = 2^{31} - 1. \]
5.1. THE GENERATION OF UNIFORMLY DISTRIBUTED RANDOM NUMBERS

Table 5.1 Series of random numbers for the linear congruential generator of the form \( I_{n+1} = (5I_n + c) \mod 8 \)

<table>
<thead>
<tr>
<th>c</th>
<th>( I_n )</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,6,7,4,5,2,3,0</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1,7,5,3,1,7,5,3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4,6,0,2,4,6,0,2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1,0,3,2,5,4,7,6</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1,1,1,1,1,1,1,1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2,6,2,6,2,6,2,6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3,3,3,3,3,3,3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4,0,4,0,4,0,4,0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5,5,5,5,5,5,5,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7,7,7,7,7,7,7,7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1,2,7,0,5,6,3,4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1,3,5,7,1,3,5,7</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2,0,6,4,2,0,6,4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3,3,3,3,3,3,3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4,0,4,0,4,0,4,0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5,5,5,5,5,5,5,5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7,7,7,7,7,7,7,7</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1,4,3,6,5,0,7,2</td>
<td>8</td>
</tr>
</tbody>
</table>

This choice has been suggested by Park and Miller [1992a]. The period of the generator is \( 2^{31} - 2 \approx 2 \times 10^{9} \). Another popular random number generator uses

\[ a = 65539; M = 2^{31} - 1; c = 0, \]

and will be used in the following program `trandom2.m`. There we will draw 3000 random numbers using the linear congruential method. In the program we will check the quality of the generator by plotting the 1D, 2D, and 3D distribution of the pseudo–random numbers. The results of the test can be seen in Figs. (5.1), (5.1), (5.1), and (5.1).

**Listing of the program trandom2.m**

**Listing of the function random1**

```matlab
function R = random1(n)
    % function to generate random numbers
    a=65539;
    M=2^31-1;
    R(1) = 12345678;
    for j=1:n-1
        % for i=1:n-1
        R(j+1) = floor(rem(a*R(j),M));
        %end
    end
    R=R/M;
```

The figures clearly reveal that the generator is not perfect. In the exercise we will learn that choosing \( a = 16807 \), the minimal standard generator, significantly improves the performance. The performance of this minimal standard generator can be increased by shuffling the output to remove low–order serial correlations (EXERCISE!!!) (ran1 of Numerical Recipes).

In the book by Press et al. other "Quick and Dirty" linear congruential generators are presented. Furthermore, it is important to remark that serial correlations can be broken up by combining two linear congruential generators.
CHAPTER 5. SAMPLING OF PROBABILITY DISTRIBUTIONS

Figure 5.2: Successive values in a series of 3000 random numbers generated for $a = 65539, c = 0, M = 2^{31} - 1$.

Figure 5.3: Histogram for a series of 3000 random numbers generated for $a = 65539, c = 0, M = 2^{31} - 1$. 
5.1. THE GENERATION OF UNIFORMLY DISTRIBUTED RANDOM NUMBERS

Figure 5.4: Correlation between successive values in a series of 3000 random numbers generated for \(a = 65539, c = 0, M = 2^{31} - 1\).

Figure 5.5: Correlation between successive values \(R(i), R(i+1), R(i+2)\) in a series of 3000 random numbers generated for \(a = 65539, c = 0, M = 2^{31} - 1\).
There are also other algorithms for the generation of random numbers: Shift–register generators (Lit: Kirkpatrick and Stoll), Fibonacci generators (lit: Knuth, James) or quasi–random numbers and we refer the reader to the original literature. Special random number generators for parallel computers has been developed.

5.2 The Transfomation Method: Invertible Distributions

In the previous section we have learned how to generate random numbers with a uniform probability distribution, so that the probability \( p(x)dx \) to generate a random number between \( x \) and \( x+dx \) is given by

\[
p(x)dx = \begin{cases} 
  dx & 0 < x < 1 \\
  0 & \text{otherwise,}
\end{cases}
\]

With the help of the random variable transformation theorem it is easy to transform uniform deviates into random numbers which are distributed according to invertible, one–to–one, distributions. We know from Chap. 2 that if we take a uniform deviate \( x \) and then transform it to a new variable \( y(x) \) the probability distribution of \( y \) is given by

\[
p(y) = \left| \frac{dx}{dy} \right| p(x), \quad (5.1)
\]

We want to derive a transformation which generates random numbers which are distributed according to a given \( p(y) \). Since \( x \) is uniformly distributed the above equation reduces to

\[
\frac{dx}{dy} = p(y), \quad (5.2)
\]

which can be easily integrated

\[
x(y) = P(y) = \int_0^y p(y')dy'.
\]

Hence, the transformation we are looking for is given by the inverse of \( P(y) \). Thus, a random variable \( Y \) with density \( p(y) \) can be generated by uniform deviates through

\[
Y = Y(X) = P^{-1}(X).
\]

In the following we will apply this method to generate exponentially and Gaussian distributed random numbers.

5.2.1 Exponential Distribution

Let \( p(y) = w \exp(-wy) \) for positive \( y \). It follows from Eq. (5.2) that

\[
\frac{dx}{dy} = w \exp(-wy).
\]

Therefore we get immediately

\[
x(y) = \int_0^y dy' w \exp(-w y') = 1 - \exp(-wy),
\]

The above expression is easily inverted

\[
y = \frac{1}{w} \ln(1-x),
\]
and since \( x \) is equally distributed in \([0, 1]\) we can generate exponentially distributed random numbers with the help of the formula
\[
y = -\frac{1}{w} \ln(x).
\]
(5.3)

It is clear that in MATLAB such an exponentially distributed random number \( Y \) can be generated with the help of the following line of code
\[
y = -\log(\text{rand}(1))/w
\]

With the help of the simple program `expdistr` we want to generate 1000 exponentially distributed random numbers and compare them with the prescribed distribution. We check also the mean value and variance and compare them with the analytical expectation values.

**Listing of the program expdistr.m**

```matlab
% expdistr — Program to generate exponentially distributed random numbers with the transformation method
clear; help expdistr;
lambda = input('Enter mean value of the distribution (1) — ');
n = input('Enter number of random numbers (1000) — ');
% generate n exponentially distributed random numbers
x = rand(n,1);
for i = 1:n
    y(i) = lambda * log(x(i));
end
% evaluate mean and variance
mean = sum(y)/n;
var = sum((y.^2 - mean)/((n-1));
xa = [0.5:1:10.5];
[m, xout] = hist(y, xa);
bar(xout, m/n);
title('Histogram of exponentially distributed random numbers');
xlabel('y'); ylabel('p(y)');
hold on;
for i = 1:11
    pexp(i) = exp(-(xout(i))/lambda)/lambda;
end
plot(xout, pexp);
hold off;
fprintf('The estimated mean is %g (exact %f)
', mean, lambda);
fprintf('The estimated variance is %g (exact %f)
', var, lambda^2);
```

In Fig. (5.2.1) we see the histogram of 1000 drawn exponentially distributed random numbers.

### 5.2.2 Gaussian Distributed Random Numbers

Gaussian distributed random numbers can be obtained with the help of the multidimensional random variable transformation theorem. Let us consider the transformation
\[
y_1 = \sqrt{-2\log(x_1)} \cos(2\pi x_2) \\
y_2 = \sqrt{-2\log(x_1)} \sin(2\pi x_2)
\]
(5.4)

(5.5)

where \( X_1 \) and \( X_2 \) are uniformly distributed random numbers on the interval \([0, 1]\). Equivalently we can write
\[
x_1 = \exp\left(-\frac{1}{2} x_1^2 + x_2^2 \right)
\]
\[
x_2 = \frac{1}{2\pi} \arctan \left( \frac{y_1}{y_2} \right).
\]
it is now straightforward to show that the Jacobian determinant reads
\[
\frac{\partial (x_1,x_2)}{\partial (y_1,y_2)} = - \left[ \frac{1}{\sqrt{2\pi}} \exp(-y_1^2/2) \right] \left[ \frac{1}{\sqrt{2\pi}} \exp(-y_2^2/2) \right].
\]

The right hand side of the above equation corresponds to the product of two independent Gaussian distributions. Thus it follows from the multidimensional version of the random variable transformation theorem for invertible distributions that the two random number generated according to Eqs. (5.4) and (5.5) are Gaussian distributed. This algorithm which allows the generation of two Gaussian random numbers from two uniformly distributed ones is called the Box–Muller method.

It is easy to implement the above algorithm in MATLAB. This is done in the program gaussdistr.m which generates gaussian random numbers with mean value \(\mu\) and variance \(\sigma\).

**Listing of the program gaussdistr.m**

```matlab
% gaussdistr - Program to generate Gaussian distributed random numbers
% with the Box-Muller method

clear; help gaussdistr;
mu = input('Enter mean value of the distribution (0) - ');
sigma = input('Enter variance of the distribution (1) - ');
n = input('Enter number of random numbers (1000) - ');
% generate 2n Gaussian distributed random numbers
x1 = rand(n,1);
x2 = rand(n,1)*2*pi;
for i=1:n
    sl = sigma*sqrt(-2*log(x1(i)));
y1(i) = sl*cos(x2(i)) + mu;
y2(i) = sl*sin(x2(i)) + mu;
end
% evaluate mean and variance
mean = sum(y1)/n;
```

**Figure 5.6:** Histogram of 1000 exponentially distributed random numbers with mean 1 generated according to the transformation method. The continuous line represents the expected exponential distribution.
5.3 THE ACCEPTANCE–REJECTION TECHNIQUE

The acceptance–rejection technique is a method of wide applicability. In its original formulation it is due to von–Neumann. The basic idea is to sample a random number from some known and appropriate probability distribution and to perform a test to determine whether or not it is acceptable for use or not. We follow the approach by Rubinstein [1981a] but consider for simplicity only the one–dimensional case.

Let us assume that the stochastic variable $X$ is defined on the interval $a < x < b$ and is distributed according to the probability density $p(x)$. We write this probability distribution as

$$p(x) = Cd(x)q(x),$$

where $Cd(x)$ is a normalization constant.

The corresponding histogram obtained by running the program for $n=1000$, $\mu=0$, $\sigma=2$ can be seen in Fig. (5.2.2).

Let us end this subsection by mentioning that normal distributed random numbers can be generated in MATLAB with the help of the function `randn`.

5.3 The Acceptance–Rejection Technique

Figure 5.7: Histogram of 1000 Gaussian distributed random numbers with mean 0 and variance 2 generated according to the Box–Muller method. The continuous line represents the expected Gaussian density.
where \( C \) is a normalization constant \( C > 1 \), \( q(x) \) is also a probability distribution and \( 0 \leq d(x) \leq 1 \). The probability distribution \( q(x) \) is the importance function, and we are supposed to know how to generate random variates distributed according to it.

The acceptance–rejection method works as follows. We generate two random variates: \( \xi \) is uniformly distributed on the interval \([0, 1]\) and \( Y \) is distributed according to \( q(y) \). Then we test whether or not the equality

\[
\xi \leq d(y)
\]

holds or not. If the condition \( \xi \leq d(y) \) is satisfied, then \( y \) is accepted as a random variate distributed according to \( p(x) \). If the condition is not satisfied, the pair \((\xi, y)\) is rejected, and we have to try again.

It is easy to demonstrate that the above method works. Let us apply Bayes’ formula to the conditional probability \( p(x|\xi \leq d(y)) \):

\[
p_x(\xi \leq d(y)) = \frac{\text{Prob}(\xi \leq d(y)|Y = x)q(x)}{\text{Prob}(\xi \leq d(y))}.
\] (5.6)

It is straightforward to compute

\[
\text{Prob}(\xi \leq d(y)|Y = x) = \text{Prob}(\xi \leq d(x)) = d(x)
\]

\[
\text{Prob}(\xi \leq d(y)) = \int \text{Prob}(\xi \leq d(Y|Y = x))q(x)dx
\]

\[
= \int q(x)d(x)dx = \int dx \frac{p(x)}{C} = \frac{1}{C}.
\]

Inserting into (5.6) we obtain finally

\[
p_x(\xi \leq d(y)) = Cd(x)q(x) = p(x),
\]

which completes the proof.

The above discussion also makes evident the role of the constant \( C \). The efficiency of the method depends on the inequality \( \xi \leq d(y) \), for independent trials the probability of success is given by \( 1/C \). \( C \) represents the average number of passes which must be made with the algorithm in order to select a variate. It is clear, that in order for the method to be efficient it must be easy to generate random numbers according to \( q(x) \) and the efficiency should be large, i.e., \( C \) should be close to one.

In the original formulation by von–Neumann the comparison function was simply chosen to be the uniform distribution. If \( M \) is the maximum of \( p(x) \) then we choose

\[
q(x) = \frac{1}{b - a}
\]

\[
d(x) = \frac{p(x)}{M}
\]

\[
C = M(b - a).
\]

The von–Neumann algorithm then simply reads

1. Generate \( \chi_1 \) and \( \chi_2 \) uniformly distributed in \([0, 1]\).
2. Evaluate \( Y = a + \chi_2(b - a) \).
3. If \( \chi_1 \leq p(y)/M \) then \( Y \) is a variate distributed according to \( p(x) \).
4. Go to 1.

As a simple example we want to generate random numbers on \([0, 1]\) distributed according to

\[
p(x) = 3x^2; \quad 0 \leq x < 1.
\]

We choose \( C = 3 \) and apply the von–Neumann algorithm.

1. Generate \( \chi_1 \) and \( \chi_2 \) uniformly distributed in \([0, 1]\).
2. Test the inequality \( \chi_1 \leq \chi_2^2 \).
3. If the equality holds we accept \( \chi_2 \) as a random number generated according to \( p(x) \).
5.4 Variance Reduction: Importance Sampling

In this section we will see how the Monte–Carlo integration algorithms can considerably be improved. The importance sampling technique will be our first encounter with a so-called variance reduction technique. We already know that the estimation of integrals by the Monte–Carlo method is affected with errors. The basic idea of variance reduction techniques is to use known information about the problem in order to improve the efficiency of the simulation. Obviously, if nothing is known about the problem no variance reduction can be achieved. On the other extreme, if we have full knowledge the variance will be reduced to zero, and there will be no need for a simulation. It is always important to be aware of what is known about the system.

We consider again the problem of estimating the integral

\[ I = \int dx f(x). \tag{5.7} \]

The central idea of importance sampling is to select random variates from regions in proportion to the importance these regions have to the integral we want to evaluate, instead of spreading them evenly. To this end we rewrite the integral (5.7) in the form

\[ I = \int \frac{f(x)}{p(x)} p(x) dx = \left< \frac{f(x)}{p(x)} \right> p(x). \]
where $X$ is a random variable with probability density $p(x)$. $P(x)$ is called the importance sampling distribution. Since the integral is obviously the expectation value of the function $f(x)/p(x)$ it can be estimated using $N$ random numbers $X_i$ distributed according to $p(x)$

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}.$$  

The function $p(x)$ has to be chosen in such a way that the variance of $f(x)/p(x)$

$$\text{Var} = \left\langle \left( \frac{f(x)}{p(x)} - I \right)^2 \right\rangle = \int_a^b \frac{f^2(x)}{p(x)} dx - I^2$$

is minimal. If $f(x) > 0$ it follows from the above equation that $\text{Var} = 0$ if we choose $p(x)$ as $p(x) = f(x)/I$. Unfortunately, this choice implies that we know already the integral we want to solve. In general, the variance can essentially be reduced if $p(x)$ is chosen to resemble $f(x)$.

As an example we consider the integral

$$I = \int_0^1 dx \exp(-x^2). \quad (5.8)$$

In the first two columns of table (5.4) we show the results of two estimates of the above integral with the help of the standard Monte–Carlo integration. In the third column we show the results of the importance sampling integration.

The simulation was performed with the help of the program mciis.m whose listing can be seen below.

Listing of the program mciis.m
5.4. VARIANCE REDUCTION: IMPORTANCE SAMPLING

Table 5.2 Monte–Carlo estimates of the integral (5.8) using the standard method \( p(x) = 1 \) and the importance sampling method \( p(x) = a \exp(-x) \)

<table>
<thead>
<tr>
<th>( p(x) = 1 )</th>
<th>( p(x) = 1 )</th>
<th>( p(x) = a \exp(-x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>1000</td>
<td>16384</td>
</tr>
<tr>
<td>( \hat{I} )</td>
<td>0.736087</td>
<td>0.74504</td>
</tr>
<tr>
<td>( \sigma_{\hat{I}} )</td>
<td>0.00131</td>
<td>0.000317</td>
</tr>
<tr>
<td>CPU time/trial (s)</td>
<td>0.000660</td>
<td>0.003837</td>
</tr>
<tr>
<td>total CPU time (s)</td>
<td>0.66</td>
<td>62.86</td>
</tr>
</tbody>
</table>

% mc_i_is = Monte–Carlo Integration
% Importance sampling
clear; help mci_is
N=input( 'Enter number of trials (1000) - ');

\[
\begin{align*}
t0 = \text{cputime} ; \\
\text{t} = \int \text{fprintf( 'Evaluate normalization constant of importance sampling distribution a = 1 - \exp(-1); ' )} ; \\
\text{a = 1 - \exp(-1); } \\
\text{t} = \int \text{rand(1,1); } \\
\text{r = rand(N,1); } \\
\text{for i = 1:N}
\text{x(i) = -log(1-r(i)*a); } \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{sum_func} = 0 ; \\
\text{for i = 1:N}
\text{p = exp(-x(i))/a; } \\
\text{func = exp(-x(i)*x(i)); } \\
\text{sum_func = sum_func + func / p; } \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{sum_var} = \text{sum_func / p / p; } \\
\text{integral} = \text{sum_func / N; } \\
\text{variance} = \text{sum_var / N - integral}^2 ; \\
\text{variance} = \text{variance / sqrt(N); }
\end{align*}
\]

\[
\begin{align*}
\text{t1 = cputime; } \\
\text{fprintf( 'The estimated integral is %f + - %g \n', integral , variance ); } \\
\text{fprintf( 'CPU time per trial is %f \n', (t1-t0)/N); } \\
\text{fprintf( 'Total CPU time: %f \n', t1-t0); }
\end{align*}
\]

The importance sampling function is chosen to be \( p(x) = a \exp(-x) \), where the constant \( a \) is chosen such that \( p(x) \) is normalized on the unit interval. Accordingly the \( N \) random numbers \( X \) distributed according to \( p(x) \) are generated with the help of the inversion method. Since

\[
P(x) = \int_0^x dx' p(x') = a [1 - \exp(-1)]
\]

the exponentially distributed random numbers on the interval \([0,1]\) are generated according to

\[
X = -\log(1 - \chi/a),
\]

where the \( \chi \) are uniformly distributed random numbers on the interval \([0,1]\). The generation of these random numbers is performed in lines \( x \) to \( y \).

It is important to remark that although the computation time per trial is larger in the importance sampling technique, the total CPU time is smaller compared to the standard Monte Carlo algorithm, because a much smaller number of realizations is required in order to achieve a desired accuracy (variance).
5.5 Sampling of Polymer Configurations

In this section we want to apply the importance sampling technique to one simple problem of polymer physics.

5.5.1 Ideal Chains

One of the simplest idealizations of a flexible polymer chain consists in replacing it by a random walk on a lattice de Gennes [1979]; Doi und Edwards [1986]. The walk is a succession of steps starting from one end and reaching an arbitrary end point. The analogy between the random walk and the polymer stems from the fact that one regards the realizations of the random walk as a possible configuration of the polymer chain. We will start by generalizing the one–dimensional random walk discussed in the previous chapter.

For simplicity, we consider cubic lattices in \( d \) dimensions, i.e. a square lattice for \( d = 2 \) and a cubic lattice for \( d = 3 \). The stepping probability \( p \) from a given site to one of its \( q = 2d \) neighbors is assumed to equal \( p = 1/q \). The number \( q \) is called the coordination number of the lattice. The length of one step will be \( l \). The analogy between random walks and polymer chains is very helpful since it allows to investigate global properties of the polymer.

Let us assume that the random walk begins at \( \mathbf{R}_0 \) and that it ends after \( N \) steps at \( \mathbf{R}_N \). A typical global property of interest in polymer physics is the end–to–end vector, which is given by

\[
\mathbf{R} = \mathbf{R}_N - \mathbf{R}_0.
\]

Of course, the end–to–end vector is the sum of the statistically independent increment vectors

\[
\mathbf{\tau}_i = \mathbf{R}_i - \mathbf{R}_{i-1} \quad \text{for} \quad i = 1, \ldots, N,
\]

i.e.,

\[
\mathbf{R} = \sum_{i=1}^{N} \mathbf{\tau}_i.
\]

It is easy to generalize the results obtained previously for the one–dimensional random walk to \( d \) dimensions. It is clear, that the projections of the \( d \) dimensional \( N \) step random walk onto the Cartesian axes are again one–dimensional random walks of, on average, \( N/d \) steps. These walks are statistically independent, so that the probability density \( p(\mathbf{R}) \) for the displacement of the \( d \)–dimensional random walk of \( N \) steps is simply obtained as the product of \( d \) one–dimensional random walks of \( N/d \) steps each

\[
P(\mathbf{R},N) = P(R_1,N/d)P(R_2,N/d) \cdots P(R_d,N/d),
\]

which inserting the Gaussian density for \( P(R_1,N/d) \) from Chap. 2 Eq. (3.60) is immediately found to be a \( d \)–dimensional Gaussian probability density

\[
P(\mathbf{R},N) = (3/2\pi Nl^2)^{(d/2)} \exp \left( -\frac{3R^2}{2Nl^2} \right).
\]

Obviously, the mean of the end–to–end distance is zero. The mean squared end–to–end distance, i.e. the variance of \( \mathbf{R} \) is (see Chap. 2)

\[
\langle R^2 \rangle = Nl^2,
\]

which is again linear in \( N \).

It is also instructive to look at the fluctuations of \( \langle R^2 \rangle \),

\[
\Delta R = \frac{\sqrt{\langle R^4 \rangle} - \langle R^2 \rangle^2}{\langle R^2 \rangle} = \sqrt{\frac{2}{3}}.
\]

\[
\Rightarrow R^4 \Rightarrow R^2 \Rightarrow R^2
\]
5.5. SAMPLING OF POLYMER CONFIGURATIONS

We immediately recognize that the variance $\Delta R$ does not decrease with increasing system size. Usually, in statistical physics fluctuations decrease with increasing number of degrees of freedom. Accordingly, the difference between expectation values and the true ensemble average scales with $1/\sqrt{N}$. Such a property is called self-averaging. It follows from the above discussion that random walks do not have the self-averaging property.

At this stage it might be appropriate to introduce the concept of universality, which allows the classification of the critical behaviour of a large variety of systems in classes with the same universality. The simple properties of the random walk allow the introduction of the Gaussian universality class. Denoting by $R = \sqrt{\langle R^2 \rangle}$ the size of the polymer, we can write Eq. (5.9) in the form

$$R = N^\nu,$$

where for the simple random walk $\nu = 1/2$ and $R$ is independent of the dimensionality of the lattice. As we will see soon, for other random walks the critical exponent $\nu$ strongly depends on the dimension of the lattice.

Since we want to sample with the help of the following program configurations of ideal polymers it is important to consider the number of possible configurations. To this end we compute the number $Z_N$ of distinct random walks. Since each lattice site has $q$ neighbors, the number of distinct possibilities at each step is $q$ and the total number of configurations is

$$Z_N = q^N.$$  \hspace{1cm} (5.10)

In order to demonstrate the above theoretical results we want to simulate a random walk on a two dimensional lattice. The listing can be seen below.

**Listing of the program rw_2d.m**

```matlab
%%% Symmetric
%%% 2D Random Walk for one particle
%%%
5 hold off;
clg;
clear;
color = ['b','k','y','g','m','c'];
10 realizations = 10;

N=input ('How many steps (1000) ?');
%%p=input ('Probability for a step to the right (0.5) ?');
step=input ('Use fixed stepsize of 1 (otherwise random stepsize in [0,1]) (1) ?');
step_size=ones(N-1,1);
if (step==1)
disp ('Stepsize is 1 !');
step_max = 1;
else
step_max = input ('Maximum stepsize (1) ?');
disp ('Stepsize is random between [0,1] !');
end

t=cputime;

% Start positions and array definition
pos(1:realizations,1:N,1:2)=0;
```

% do the realizations
for j = 1:realizations,
    % draw all random numbers at once
    % between 1 and 4
    random = floor(rand(N-1,1)*3.999)+1;
    if ( step == 0)
        step_size = step_max.*rand(N-1,1);
    end
    % do the steps
    for i = 2:N,
        if ( random(i-1) == 1)
            pos(j,i,1) = pos(j,i-1,1) + step_size(i-1);
            pos(j,i,2) = pos(j,i-1,2);
        elseif ( random(i-1) == 2)
            pos(j,i,1) = pos(j,i-1,1) - step_size(i-1);
            pos(j,i,2) = pos(j,i-1,2);
        elseif ( random(i-1) == 3)
            pos(j,i,2) = pos(j,i-1,2) + step_size(i-1);
            pos(j,i,1) = pos(j,i-1,1);
        elseif ( random(i-1) == 4)
            pos(j,i,2) = pos(j,i-1,2) - step_size(i-1);
            pos(j,i,1) = pos(j,i-1,1);
        end
    end
end
% plot the elapsed CPU time
$t =$ cputime;
disp($\text{sprintf('CPU Time: %.2f seconds', t))$);

% plot the paths of the walker
hold on;
axis square;
maximum = max(max(abs(max(pos)), abs(min(pos))));
axis([-maximum maximum -maximum maximum]);
for j = 1:realizations
    plot(pos(j,:,1), pos(j,:,2), color(1+rem(j,6)));
end
% plot zero axis
plot([-maximum maximum],[0 0],' r ');
plot([0 0],[-maximum maximum],' r ');

A few comments to the listing seem appropriate. The position of the random walker is stored in the matrix $\text{pos}(1:\text{realizations},1:N,1:2)$, where $\text{realizations}$ denotes the number of realizations, $N$ is the number of steps, and the last argument records the position along the x and the y axis. In order to generate a realization (lines 31 to 58) we draw at once $N$ random numbers which are equally distributed between the four discrete values 1,2,3, and 4, which number the four possible neighbors of the lattice point on which the walker is resting. Then within a for loop over all steps we select at random the new position of the walker with the help of the if-elseif-end construction in lines 42 to 57. These procedure is repeated $\text{realizations}$ times. Finally, we plot the paths of the walker. Since we want to plot all paths in the same square plot (in MATLAB this can be done with the help of the command $\text{axis square}$ it is necessary to find out the maximal distance reached by the walker in the 10 realizations. This is done in line 68 and the length of the axis is fixed in line 69 with the command.
5.5. SAMPLING OF POLYMER CONFIGURATIONS

![2d random walk: 10 realizations](image)

Figure 5.9: Ten realizations of a two–dimensional random walk on a square lattice.

\[
\text{axis([-maximum maximum -maximum maximum]);}
\]

With the help of the `hold on hold off` construction we plot ten realizations on the same figure.

We run the program and generate 10 realizations of a random walk of 1000 steps lengths. These realizations can be seen in Fig. (5.5.1). It is now left to the exercises to investigate the end–to–distance of the ideal polymer chain in order to check whether the simulation is in agreement with the Gaussian polymer theory.

5.5.2 Real Chains

In Fig. (5.5.1) we see that the realizations of the random walk, i.e. the polymer chains intersect themselves. This is of course an unphysical behaviour since real polymer chains do not intersect. In order to simulate real chains on a lattice we still can represent the polymer as a random walk, but now the walk can not intersect itself. Such random walks are called self–avoiding random walks (SAW). In other words the SAW has the constraint that the walker cannot return to sites he already visited. As a natural consequence SAWs are more extended than the usual random walks. Of course, also the universal properties of the SAW are different from those of the Gaussian random walk. In particular the exponent \( \nu \) strongly depends on the dimension of the lattice. It is clear that in \( d = 1 \) a SAW is simply a rigid rod and we have \( \nu = 1 \). For increasing dimension of the lattice \( \nu \) decreases as the self–avoiding constraint becomes less important. For large dimensions the difference between random walk and SAW vanishes. In fact for the critical dimension \( d_c = 4 \) (and for higher dimensions) random walk and SAW have the same universal properties de Gennes [1979]; Raposo et al. [1991].

In general it is not possible to evaluate the critical exponent \( \nu \) analytically for the SAW. This is possible only for \( d = 1, d = 2 \) and \( d \geq 4 \). However, a simple mean field theory, the Flory theory, makes the following predictions

\[
\nu = \begin{cases} 
3/(2+d) & \text{for } 1 \leq d \leq 4, \\
1/2 & \text{for } d \geq 4.
\end{cases}
\]

As we will see this theoretical predictions agree well with the stochastic simulations (!!!!!).
The total number of self–avoiding random walks of $N$ steps has (for large $N$) the following asymptotic form

$$Z_N = \text{constant} \times N^{d-1} \frac{N}{q_{\text{eff}}}.$$ 

The second factor in the above scaling law reminds us of the expression (5.10) which holds for ideal random walks. The effective coordination number is smaller then the coordination number $q$, e.g. in 3 dimensions on a cubic lattice we have $q = 6$ and $q_{\text{eff}} = 4.68$ and in two dimensions on a squared lattice we have $q = 4$ and $q_{\text{eff}} = 2.63$ [Kremer und Binder 1988]. The first factor $N^{d-1}$ is called the enhancement factor. The universal exponent $\gamma$ depends on the dimension of the lattice

$$\gamma \approx \begin{cases} 
7/6 & \text{for } d = 3 \\
4/3 & \text{for } d = 2.
\end{cases}$$

Note that for $d = 1$ we have $Z_N = 2$, independently of $N$. Hence, $q_{\text{eff}} = 1$ and $\gamma = 1$.

As an application of the sampling techniques we have learned in this chapter we address the problem of generating configurations of real polymer chains on a lattice, or in other words we want to generate realizations of SAW on the computer.

DISKUSSION VON E(R2) SAW !!!!!!!!!!!!!

**Simple sampling**

The simplest algorithm to generate chains of length $N$ is the following one. We start at the origin. The first step is, of course, taken randomly from the $q$ adjacent sites. The following steps are taken from the remaining $q_0 = q - 1$ possible directions, because a direct backfolding is forbidden. Of course, the SAW condition is also violated whenever the chosen lattice site is already taken. In fact, the self–avoiding walk has an infinite memory. In such cases the attempted walk is terminated and discarded and we have to start with a new chain. In this way we automatically fulfill the condition that all walks of equal length have the same probability. All sampled configurations are statistically independent and hence the standard error analysis is possible. It is instructive to estimate that the success rate $s_N$, i.e., the fraction of walks that will continue successfully for $N$ steps, is of the order

$$s_N \approx \exp(-N\mu),$$

which for large $N$ decreases exponentially as

$$s_N \approx \exp(-N\mu),$$

with $\mu = \ln[q_0/(q - 1)]$. On a two–dimensional square lattice $\mu = 0.128$, on a 3d cubic lattice $\mu = 0.065$ [Kremer und Binder 1988]. Because of this exponential decrease it is very difficult to generate very long chains with the simple sampling algorithm. This problem is called the ”attrition problem”. In Fig. (5.5.2) we have plotted the average number of trials necessary to generate polymers of a given length as a function of the length of the polymers. The figure was generated with the help of the program **RW_2D_sa.m**.

**Listing of the program RW_2D_sa.m**

```matlab

%%%% Self–avoiding Symmetric
%%%% 2D Random Walk for one particle
%%%%
%%%% Attrition problem count
%%%%
hold off; clear;
color = ['b','k','y','g','m','c'];
```
5.5. SAMPLING OF POLYMER CONFIGURATIONS

![2d random walk: 10 realizations](image_url)

Figure 5.10: Attrition problem: The average number of trials necessary to generate polymers of a given length increases exponentially with the length of the polymers.

```matlab
10 realizations = input (' How many realizations (1) ? ');
N = input (' How many walker steps (200) ? ');
len = input (' Length of the polymer (200) ? ');
%%p = input (' Probability for a step to the right (0.5) ? ');
step_size = 1; disp (' Stepsize is 1 !'); step_max = 1;

t = cputime;
% let it walk ...

% Start positions and array definitions
20 xpos(1:realizations,1:N+1)=0; ypos(1:realizations,1:N+1)=0;
trapped(1:N+1)=0;
attrition(1:realizations)=0;

% do the realizations
25 for j=1:realizations,
    % do the steps
    trap(1:4)=1; flag=0;
    while (sum(trap)==4 & flag == 0)
        attrition(j)=attrition(j)+1;

    for i=2:N+1,
        % look for a direction to walk to
        % (self avoiding)
        flag =1;
        trap(1:4)=0;
    while ( flag ==1)
        random=floor ( rand(1)*3.99)+1;
        % right
        if ( random==1)
```
\begin{verbatim}
CHAPTER 5. SAMPLING OF PROBABILITY DISTRIBUTIONS

\texttt{xpos(j,i)=xpos(j,i-1)+step_size;}
\texttt{ypos(j,i)=ypos(j,i-1);}
\texttt{trap(1)=1;}
\texttt{%% left}
\texttt{elseif (random==2)}
\texttt{xpos(j,i)=xpos(j,i-1)-step_size;}
\texttt{ypos(j,i)=ypos(j,i-1);}
\texttt{trap(2)=1;}
\texttt{%% up}
\texttt{elseif (random==3)}
\texttt{xpos(j,i)=xpos(j,i-1);}
\texttt{ypos(j,i)=ypos(j,i-1)+step_size;}
\texttt{trap(3)=1;}
\texttt{%% down}
\texttt{elseif (random==4)}
\texttt{xpos(j,i)=xpos(j,i-1);}
\texttt{ypos(j,i)=ypos(j,i-1)-step_size;}
\texttt{trap(4)=1;}
\texttt{end}
\texttt{%% check self avoiding up to length "len"}
\texttt{flag=0;}
\texttt{for k=i-1:len(1,(i-1)-len),}
\texttt{if (xpos(j,i)==xpos(j,k) & ypos(j,i)==ypos(j,k))}
\texttt{flag=1;}
\texttt{break;}
\texttt{end}
\texttt{%% Check if trapped}
\texttt{if (flag==1 & sum(trap)==4)}
\texttt{\% disp(sprintf(' I am trapped and cant escape after %i steps !', i));}
\texttt{trapped(i)=trapped(i)+1;}
\texttt{flag=0;}
\texttt{end}
\texttt{\% end step loop}
\texttt{end}
\texttt{\% end realizations loop}
\texttt{end}

\texttt{\% plot the elapsed CPU time}
\texttt{t=cputime(t);}
\texttt{disp(sprintf('CPU Time : %f seconds', t));}
\texttt{\% plot the paths of the walker}
\texttt{hold on;}
\texttt{axis square;}
\texttt{maximum=max(abs(max(max(xpos))), abs(min(min(xpos))));}
\texttt{maximum=max(maximum, max(abs(max(max(ypos))), abs(min(min(ypos))))));
\texttt{axis([-maximum maximum -maximum maximum])};
\end{verbatim}
5.5. SAMPLING OF POLYMER CONFIGURATIONS

To overcome the attrition problem it is necessary to exploit biased sampling techniques. One such technique is the "inversely restricted sampling" of Rosenbluth and Rosenbluth [1955]. The idea is to associate each SAW of \( N \) steps with appropriately chosen weighting factors.

We consider a SAW of \( i \) steps on a lattice with the coordination number \( q \). In order to sample the next step we have to check which of the \( q_0 = q - 1 \) neighboring sites is empty. Essentially, we have two possibilities:

1. If \( k (q_0 \geq k > 0) \) sites are empty we choose the possible steps with equal probability \( 1/k \).
2. If there is no empty site \( (k = 0) \) we terminate the walk and start from the beginning.

It is clear that each \( N \) step walk has the probability

\[
P_N(\{P_i\}) = \prod_{i=1}^{N} (k_i)^{-1}
\]

and hence dense configurations are more probable. Thus this obvious procedure of choosing only among those sites that do not violate the SAW condition does not give equal statistical weights to the generated configurations. But since we have seen that these weight can be calculated we can compensate this bias by weighting each chain in the sample by the factor

\[
W_N(\{P_i\}) = \prod_{i=1}^{N} \frac{k_i}{q_0}
\]

Note, that we have \( W_N P_N = q_0^{-N} \). The importance sampling procedure samples exactly the same configuration space as simple sampling.

The correct way to estimate \( \langle R^2 \rangle \) is

\[
\langle R^2 \rangle = \frac{\sum_{i=1}^{M} W_{N,i} R_{N,i}^2}{\sum_{i=1}^{M} W_{N,i}^2}
\]
where the sum is over all realizations.

In order to compare the two sampling techniques we have written a program RW2Dsa2.m (VORSCHLAG: SAW2 !!!!!) which by specifying an input parameter draws configurations of real chains on a two-dimensional squared lattice with the simple sampling technique or with the biased (importance) sampling technique. The flow diagram of the program can be seen in Fig. (5.5.2).

Listing of the program rw2d_sa2.m

```plaintext
%%% Self-avoiding Symmetric
%%% 2D Random Walk for one particle
%%% calculating end to end distance
```
5.5. SAMPLING OF POLYMER CONFIGURATIONS

hold off; clf; clear; set(0,'DefaultAxesFontSize',14);

% maximum number of displayed text during a simulation
10 disp_step = 5;
% maximum number of plots for one N
plot_step = 5;
% colors for the random walks plots
color = [ 'b' , 'k' , 'y' , 'g' , 'm' , 'c' ];
15
% weighted RW ??
weight = input( ' Weighted random walk ( 1 = ja ) ? ' );

% input parameters
20 realizations = input( ' How many realizations ( 10 ) ? ' );
N_min = input( ' polymer length to start with ( 5 ) ? ' );
N_max = input( ' polymer length to stop with ( 5 5 ) ? ' );
N_step = input( ' Step size for N ( 1 0 ) ? ' );
25 step_size = 1; disp( ' Stepsize is 1 ! ' ); step_max = 1;
R_square( N_min : N_step : N_max ) = 0;

% Loop over length of polymer
30 for len = N_min : N_step : N_max,
len = len;
7 t(len) = cputime;
% let it walk ... 
35
% Start positions and array definition
xpos(1 : realizations , 1 : N) = 0;
ypos(1 : realizations , 1 : N) = 0;
40 % do the realizations
for j = 1 : realizations ,
% set starting weight to 1
W_2d(j) = 1;
45
trap(1 : 4) = 1;
% do the realization until no deadlock occurs
while ( sum(trap) == 4 ),
% do the steps
for i = 2 : N,

% look for a direction to walk to
%%% ( self avoiding )
flag = 1;
trap(1 : 4) = 0;
while ( flag == 1 )
random = floor ( rand(1) * 3.99 ) + 1;
%%% right
if ( random == 1 )
xpos(j,i) = xpos(j,i-1) + step_size;
ypos(j,i) = ypos(j,i-1);
trap(1) = 1;
%%% left
elseif ( random == 2 )

...
xpos(j,i)=xpos(j,i-1)-step_size;
ypos(j,i)=ypos(j,i-1);
trap(2)=1;

% up
elseif (random==3)
ypos(j,i)=ypos(j,i-1)+step_size;
xpos(j,i)=xpos(j,i-1);
trap(3)=1;

% down
elseif (random==4)
ypos(j,i)=ypos(j,i-1)-step_size;
xpos(j,i)=xpos(j,i-1);
trap(4)=1;
end

% check self avoiding up to length "len"
flag=0;
for k=i-1:i:max(1,(i-1)-len),
  if (xpos(j,i)==xpos(j,k) & ypos(j,i)==ypos(j,k))
    flag=1;
    break;
end

end

%% Check if trapped
if (flag==1 & sum(trap)==4)
  disp('I am trapped and cant escape!');
  break;
end

%% Check if trapped
if (flag==1 & sum(trap)==4) break; end

%% Change weight appropriately
if (weight==1 & i>2)
anz=0;
xy(1,1)=xpos(j,i-1)-1; xy(1,2)=ypos(j,i-1)-0;
xy(2,1)=xpos(j,i-1)-0; xy(2,2)=ypos(j,i-1)-1;
xy(3,1)=xpos(j,i-1)+1; xy(3,2)=ypos(j,i-1)+0;
xy(4,1)=xpos(j,i-1)+0; xy(4,2)=ypos(j,i-1)+1;
for k=i-1:i:1,
  for direct=1:4,
    if (xpos(j,k)==xy(direct,1) & ypos(j,k)==xy(direct,2))
      anz=anz+1;
      break;
    end
  end
% not possible to have anz=0!
if (anz==0)
  disp('!!! Anzahl 0!!!');
else
  W_2d(j)=W_2d(j)*(anz/3);
end
end

% display only "disp_step" times in a simulation
if (rem(j,floor(realizations/disps_step))==0)
disp(sprintf('I did %i steps in realization # %i !',i,j));
end
end_point(j)=i;

% end while loop
end

% end realizations loop
end

% plot the elapsed CPU time
t(len)=cputime-t(len);
disp(sprintf('CPU Time: %f seconds',t(len)));

% plot the paths of the walker
hold on;
axis(square);
maximum=max(abs(max(abs(xpos))),abs(min(min(xpos))));
maximum=max(abs(max(abs(ypos))),abs(min(min(ypos))));
axis([-maximum maximum -maximum maximum maximum]);
% plot zero axis
plot([-maximum maximum],[0 0],'r');
plot([0 0],[-maximum maximum],'r');
% plot the trajectories (but maximum plot_step)
for j=1:realizations
  if (rem(j,floor(realizations/plot_step))==0)
    plot(xpos(j,:),ypos(j,:),color(1+rem(j,6)));
    % mark the endpoint
    plot(xpos(j,end_point(j)),ypos(j,end_point(j)),'rs');
  end
end

% Calculate the end to end distance
R_square(len)=sum((W_2d(:).*((xpos(:,N).^2+ypos(:,N).^2))/sum(W_2d)));
R_square_error(len)=std((W_2d(:).*((xpos(:,N).^2+ypos(:,N).^2))/sum(W_2d)));

% end loop over length of polymer
end

% plot the scaling behaviour!
% blue line
figure;
loglog(N_min:N_step:N_max,R_square(N_min:N_step:N_max));
hold on;
% compare with theoretical scaling behaviour
% dotted red line
factor=0.8;
loglog(N_min:N_step:N_max,factor*((N_min:N_step:N_max).^(4/3)),r:');
title('end to end distance versus N in 2 dimensions');
xlabel('length N');
ylabel('<R>');
% do a polynomial fit in the loglog plot to get scaling
% fit a line using least square method
% black line
coeff = polyfit( log(N_min:N_step:N_max), log(R_square(N_min:N_step:N_max)), 1);
disp(sprintf('estimated scaling exponent s = %f (theory: 1.33333) !', coeff(1)));
disp(sprintf('prefactor a of a*N^s is: %f, exp(coeff(2))'),
loglog(N_min:N_step:N_max, exp(coeff(2))*N_min:N_step:N_max.^coeff(1), 'k');
axis tight;

% elapsed CPU time plot in new figure
figure;
plot(N_min:N_step:N_max, t(N_min:N_step:N_max));
title('CPU time for the 2D self avoiding random walk');
xlabel('length N');
ylabel('CPU time in seconds');

Let us comment on the program. BLA BLA BLA !!!!!!!! We run the program for the following set of parameters. We variate the length of the SAW from N=5 to N=55 in steps of 10. For each SAW length we generate 10 realizations first with the simple sampling technique and second with the importance sampling technique.

First we run the program for the simple sampling algorithm. In Fig. (5.5.2) we see 5 realizations of the self–avoiding random walk for N=55 generated by the importance sampling technique. In Fig. (5.5.2) we show the mean square end–to–end distance as a function of N in a double logarithmic plot. The continuous line indicates the theoretically expected behaviour whereas the dotted line is the result of the least square fit to a straight line from which we derived the scaling exponent. In Fig. (5.5.2) we plot the CPU time necessary to generate 10 realizations of a polymer of a specific length as a function of N. One easily recognizes that the CPU time increases exponentially. The critical exponent determined by the least square fit $R^2 \approx aN^s$ where $a=0.81$ and $s=1.36$. Thus the estimate of the critical exponent is $\nu = 0.68$ (the expected result is 0.75).

Let us now discuss the importance sampling approach. In Fig. (5.5.2) we see 5 realizations of the self–avoiding random walk for N=55 generated by the importance sampling technique. In Fig. (5.5.2) we show the mean square end–to–end distance as a function of N in a double logarithmic plot. The continuous line indicates the theoretically expected behaviour whereas the dotted line is the result of the least square fit to
5.5. SAMPLING OF POLYMER CONFIGURATIONS

**Figure 5.13:** The mean square end–to–end distance of a self–avoiding random walk generated by the simple sampling technique as a function of the polymer length.

**Figure 5.14:** The CPU time for generating 10 realizations of a self–avoiding random walk by the simple sampling technique as a function of the polymer length.
Self Avoiding Random Walk in 2 Dimensions

Figure 5.15: Five realizations of a two-dimensional self-avoiding random walk on a square lattice generated by the importance sampling technique.

Table 5.3 Values of $\langle R^2 \rangle$ as functions of $N$, for two-dimensional random walks generated by the simple sampling (ss) and by the importance sampling (is) technique.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\langle R^2 \rangle$ (ss)</th>
<th>$\langle R^2 \rangle$ (is)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7.0</td>
<td>4.3</td>
</tr>
<tr>
<td>15</td>
<td>41.6</td>
<td>14.02</td>
</tr>
<tr>
<td>25</td>
<td>51.2</td>
<td>44.20</td>
</tr>
<tr>
<td>35</td>
<td>86.6</td>
<td>100.13</td>
</tr>
<tr>
<td>55</td>
<td>193.8</td>
<td>72.51</td>
</tr>
<tr>
<td>65</td>
<td>179.2</td>
<td>103.79</td>
</tr>
</tbody>
</table>

a straight line from which we derived the scaling exponent. In Fig. (5.5.2) we plot the CPU time necessary to generate 10 realizations of a polymer of a specific length as a function of $N$. One easily recognizes that the CPU time increases exponentially. The critical exponent determined by the least square fit $R^2 \approx aN^s$, where $a=0.424$ and $s=1.404$. Thus the estimate of the critical exponent is $\nu = 0.702$ (the expected result is 0.75).

In table (5.5.2) we sum up the results of the simulations.

In the next table (5.5.2) we show the result of another run of the program `rw2dsa2` which was run for 500 realizations.

The same data are depicted in Fig. (5.5.2). The scaling behaviour estimated from the above data is found to be $R^2 = aN^s$, where $a=0.38614$ and $s=1.41248$. Thus the critical exponent is found to $\nu = 0.706$.

Fig. (5.5.2) shows the corresponding CPU time consumption.

IRGENDEIN VERGLEICH ZWISCHEN SIMPLE UND IMPORTANCE SAMPLING !!!!!
FIGUREN: FEHLER!!!!

5.6 Exercises

**Exercise 5.1 Random-Number Generator [Press et al., 1992b, Chapter 7]**

Now that we have already used random numbers for many simulations, we would like to know, how to
Figure 5.16: The mean square end-to-end distance of a self-avoiding random walk generated by the importance sampling technique as a function of the polymer length.

Figure 5.17: The CPU time for generating 10 realizations of a self-avoiding random walk by the importance sampling technique as a function of the polymer length.
Table 5.4 Mean square end–to–end distance estimated by importance sampling from a sample of 500 realizations.

<table>
<thead>
<tr>
<th>N</th>
<th>(&lt;R^2&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13.709</td>
</tr>
<tr>
<td>30</td>
<td>45.651</td>
</tr>
<tr>
<td>50</td>
<td>53.559</td>
</tr>
<tr>
<td>70</td>
<td>153.023</td>
</tr>
<tr>
<td>90</td>
<td>234.747</td>
</tr>
<tr>
<td>110</td>
<td>220.095</td>
</tr>
<tr>
<td>130</td>
<td>350.079</td>
</tr>
<tr>
<td>150</td>
<td>566.384</td>
</tr>
<tr>
<td>170</td>
<td>711.81</td>
</tr>
<tr>
<td>190</td>
<td>744.514</td>
</tr>
</tbody>
</table>

Figure 5.18: The mean square end–to–end distance of a self–avoiding random walk estimated from a sample of 500 realizations by the importance sampling technique as a function of the polymer length.
generate random numbers on a computer. Therefore write a program, which produces random numbers using the linear congruential method. That means, successive random numbers are generated by

\[ I_{n+1} = (aI_n + c) \mod M, \]

where \( I_1 \) is the initial seed.

Use the following parameters:

1. \( a = 16807; c = 0; M = 2^{31} - 1 \) (the Matlab parameters for \texttt{rand}, the ran0 routine from Numerical Recipes book)

2. \( a = 65539; c = 0; M = 2^{31} - 1 \)

Compare the sequences of the two generators by looking at the produced distribution, calculating the moments (use Assignment 2), looking at vectors of random numbers with length 2 and 3 using the plotting facilities of Matlab.

Maybe you have another idea to check the random number generators?? (Dont forget to tell us ... )

**EXERCISE 5.2 Poisson Distribution [Rubinstein, 1981b, Chapter 3.7.2]**

Write a program to produce Poisson distributed random numbers.

\[
P_\lambda(n) = \frac{\lambda^n e^{-\lambda}}{n!}, \quad n \in \mathbb{N}, \lambda > 0 \quad \text{Poisson Distribution}
\]

The first way would be to use the rejection method as described in [Press et al., 1992b, Chapter 7.3]. We want to use a different method, based on the generation of the exponential distribution (and therefore the transformation method).

If the time intervals between some events are from an exponential distribution, then the number of events occurring in an unit interval of time is from the Poisson distribution \( P_\lambda(n) \). Which means (use \( T_i \equiv -\frac{1}{\lambda} \ln U_i, U_i \) in (0,1))

\[
\sum_{i=1}^{x} T_i \leq 1 \leq \sum_{i=1}^{x+1} T_i \Leftrightarrow \prod_{i=1}^{x} U_i \geq e^{-\lambda} \geq \prod_{i=1}^{x+1} U_i
\]
where \( T_i (i=0,1,\ldots,x+1) \) are drawn from \( \exp(1/\lambda)T_i \). Then the corresponding algorithm reads like: (\( A \) is a real variable, \( k \) is an integer variable)

1. Set \( A = 1 \) and \( k = 0 \)
2. Generate a random number \( U_k \) from \( U(0,1) \) (uniform distribution, but not \( U[0,1] \) !)
3. Set \( A \) to \( A \times U_k \)
4. if \( A < e^{-T_i} \) then use \( k \) as random number and finish!
5. set \( k \) to \( k + 1 \) (increment \( k \)) and return to step 2

Again check the generated sequences of random numbers like in the previous exercises.

**Exercise 5.3** Acceptance-Rejection-Method [Rubinstein, 1981b, Chapter 3.4.2]
Calculate the volume of a \( n \)-dimensional hypersphere, or in other words: generate a random vector uniformly distributed inside an \( n \)-dimensional unit sphere using the Acceptance-Rejection-Method.

The idea is to produce a vector inside the \( n \)-dimensional hypercube and accept all random vectors lying inside the hypersphere. Remember that you have to produce numbers in \( [-1,1] \), so you have to transform the numbers generated by \( \text{rand}() \) to be in the desired interval.

What is the exact result? Compare it with the estimated values and plot the error versus the dimension \( n \).

What has to be changed, if you want to calculate the surface of the unit-sphere?

Is the algorithm efficient for large \( n \) and why/why not?

**Exercise 5.4** Importance Sampling [Rubinstein, 1981b, page 122]
First calculate the integral (Maxwell-Boltzmann-distribution for the modulus of the velocity)

\[
\int_0^\infty v e^{-\frac{v^2}{2}} dv = \left[-e^{-\frac{v^2}{2}}\right]_0^\infty = \frac{1}{2},
\]

using the standard Monte-Carlo-Integration with simple sampling. Use a reasonable cut-off value \( c \) to get rid of the infinite integration interval. What is the systematic error involved using a cut-off of \( c \)? e.g. use a \( c \) to reduce the systematic error to be lower than \( 10^{-10} \)? (\( -\frac{1}{2}e^{-\frac{c^2}{2}} < \varepsilon = 10^{-10} \Rightarrow c > 4.73 \), so \( c = 10 \) would be a very good choice.)

Now write a program for calculating the same integral, but using the importance sampling method. Therefore we choose the importance function (distribution) to be

\[
p(v) = e^{-\frac{v^2}{2}}.
\]

So the integral to be calculated for this \( p(v) \) is

\[
I_c = \int_0^c \frac{v e^{-\frac{v^2}{2}}}{e^{-\frac{c^2}{2}}} \times e^{-\frac{v^2}{2}} = \int_0^c v \times p(v) dv.
\]

You have to generate random numbers distributed with \( p(v) \) (normally distributed random numbers can be generated in Matlab using the \( \text{randn}() \) function.) and evaluate the integral \( \int_0^c v dp(v) \). If you use \( \text{randn}() \), you have to correct for the normalization factor of the normal distribution.

The formula for the standard MCI:

\[
I_c = \frac{1}{N} \sum_{i=0}^N c \xi_i,
\]

where \( \xi_i \) is a normally distributed random number.

Compare both methods using a plot of accuracy versus CPU-time.
Exercise 5.5 Symmetric 2D Random Walk - First Passage Times

Write a program for a 2D random walk (not self-avoiding) on a square lattice with constant step size.

Use a sample of walkers to calculate an estimate of the time (number of steps) it needs, to cross a given circle around the origin with Radius $R$. This is called the first passage time.

A square lattice in 2D with a symmetric random walker (probability for each direction is $p$) starting at the origin and crossing the circle with radius $R$ after three equidistant steps with size $a$.

Exercise 5.6 Scaling Behavior of Random Walk in 2D and 3D

Write a program for a 2D random walk on a square lattice (symmetric and asymmetric). Use this program to calculate the scaling behavior of the end-to-end distance $<R^2>$ for different lengths $N$ of the walk.

After viewing some nice walks, plot the end-to-end distance versus the length $N$ of the random walk. If you have enough data, do a least square fit using the `polyfit` function of Matlab using a fit function:

$$<R^2> \approx aN^b,$$

to get $a$ and $b$.

Then extend the program to a 3D random walk on a cubic lattice and again analyze the scaling behavior of the end-to-end distance.

Now compare your results (b) with the experiments done in the lecture about the self-avoiding random walk!

Remark: A least square fit is a method, where you fit a analytically known function with (observed) data. The function is given using certain parameters (here the function is $y = R^2 = aN^b$) and the parameters have to be calculated. This is done by minimizing the euclidean distance ("mean square") between the function and the data.

Exercise 5.7 Percolation in 2D and Cluster Algorithms [MacKown, 1997, Chapter 4.2]

Assume we have a 2D plane divided into cells of equal size. We start by sweeping the whole plane from the upper left corner to the lower right corner. With probability $p$ we set each cell to one (with probability $1-p$ to zero). Then we are left with an array of ones and zeros.

Now we analyze this pattern by looking for clusters of ones in the whole plane. A cluster is an area of ones where the ones are connected to each other in the nearest neighbor sense (no diagonal bonds possible). If we have a cluster connecting the left side with the right side or the upper and the lower side, we call it a spanning cluster. Otherwise it is just a finite cluster of a certain size.

Now, one is interested in the critical probability $p_c$, where such spanning clusters occur or do not occur. Of course using a simulation with finite size, we get a broad interval (instead of a sharp one) in which the transition of the two behaviors takes place. We will neglect such finite size effects and concentrate on the transition probability.

Now you can watch the configurations and decide if there is a spanning cluster or not. That is very tedious. Therefore we give a very easy algorithm for clustering the given configuration (The Hoshen-Kopelman algorithm): We
need two arrays, one stores the indices (say N(m), a 1D array) and the other one stores the labels assigned to each site (say L(i,j), a N x N-array, where N is the number of cells in one direction).

You scan all the occupied sites starting with the upper left corner assigning a label to each site using the following rules (unoccupied sites are labeled with zero):

- if neither of the site above and to the left of the cell are occupied, you assign a new integer value m to the site and set N(m) = m.
- if the left and/or the upper site are occupied and have the same label assigned to them, you give the site the same label.
- if the labels of the left and upper site are different, you assign the smaller (say k) of the two labels to the new site. And you set the index of the larger one also equal to this value (N(larger) = k).

After you have swept over all sites, you have to remove redundant labels from the array N. An index is redundant, if the index h is not equivalent to the label stored in N(h), and change the corresponding labels in the 2D array L(i,j).

Now you are done: Just look for the labels on opposite faces, if there are same labels on them, you have a spanning cluster.

Write a program for a percolation in 2D and try to estimate the critical probability. Increase the size of the grid and compare. To that end plot the probability distribution of having a spanning cluster versus the probability p used for creating the configuration. For 2D pc can be calculated analytically: pc = 0.5.

**Exercise 5.8 The Einstein Solid and the Boltzmann Distribution [Rhula, 1992, Chapter 5]**

The probability for an oscillator in a solid to be in a state with energy ε is given by the Boltzmann distribution (canonical distribution)

\[
P(\varepsilon_i) = \frac{1}{Z} \exp\left(-\frac{\varepsilon_i}{kT}\right),
\]

where Z is the partition function (Z = \(\sum_i \exp(-\varepsilon_i/kT)\)). This distribution plays a very important role in physics and especially in statistical mechanics. To get a first impression of the distribution and to have a very simple model producing a Boltzmann distribution, we study a very simple model for a solid, named after Einstein.

We study a 2D plane and divide the plane into cells of equal sizes, like quantum mechanics through the Heisenberg uncertainty principle prescribes. Each cell represents an atom and each atom can vibrate around its equilibrium position in the middle of the cell. The strength of the vibration is quantized and given in multiples of a fundamental mode with energy \(\hbar \omega\). Because we have an isolated system, no energy gets dissipated or absorbed. The only way of changing the configuration is by hopping of a quanta from one cell to another (which is just a transfer of an energy quantum \(\hbar \omega\)).

The algorithm (sometimes called temperature game) reads as follows:

- Initialize all cells with 1 quanta
- Choose a source cell and a destination cell at random (drawn from a uniform distribution)
- If there is NO quanta in the source cell, draw new random numbers.
- If there is a quanta in the source cell to jump and the destination cell is not equivalent to the source cell, do a jump of one quanta from source cell to destination cell.
- Having done enough jumps, we reach the equilibrium situation. Now we can count the number of cells \(\nu\) having zero, one, two, etc. quanta in it (call these \(m(\nu) = m(\varepsilon_i)\)). By plotting the corresponding histogram, we get the Boltzmann distribution.

The energy of a cell at the end is just the number of quanta \(\nu\) times the energy per quanta plus 1/2\(\hbar \omega\) (assuming harmonic oscillations). From now on, we neglect the zero temperatur energy of the harmonic oscillator to make life easier (it is just a shift of energy anyway.).

The Boltzmann distribution for the histogram is

\[
m(\varepsilon_\nu) = m(0) \exp\left(-\frac{\nu \hbar \omega}{kT}\right).
\]
So to find the “temperature” of the solid, we have to use a fit of the Boltzmann distribution to the data and extract the value $\nu_{1/2}$, where the distribution has come down to half of the initial value $m(0)$. Then we can use

$$\frac{\hbar \omega}{kT} = \frac{\ln 2}{\nu_{1/2}},$$

and can calculate the temperature for the given system. The temperature depends only on the size of the simulation and therefore the total number of quantas present.

Write a program to implement the above algorithm in Matlab. Watch the configuration evolving in time (especially at the first steps). Compare the calculated distribution with a Boltzmann distribution using a least square fit in a semilogarithmic (y-axes) plot (use \texttt{polyfit}). Why don’t we get a uniform distribution as an equilibrium distribution?
Bibliography


Part III

Stochastic Processes
Chapter 6

Markov Processes and Master Equations

This chapter is devoted to the introduction of some mathematical concepts, which allow the correct treatment of time–dependent probabilistic phenomena. Such processes occur in many branches of physics. A typical example, is for instance, the dynamics of the velocity field in a turbulent fluid. We will introduce stochastic processes as time dependent stochastic variables and we will learn how the dynamics of a particular class of stochastic processes, the so–called Markov processes is described with the help of differential Chapman–Kolmogorov equations. These concepts will be applied in the next chapters to typical examples from statistical physics.

6.1 Stochastic Processes

We have already learned in Chap. 2 that once a stochastic variable $X$ has been defined it is possible to define other stochastic variables, say $Y$, as functions of $X$ by some mapping $f$. In particular, the quantity $Y$ may be a function of an additional time variable $t$, i.e.,

$$Y(t) = f(X,t),$$

Sloppy speaking, such a quantity $Y(t)$ is called a stochastic processes. If we insert for $X$ one of its possible values $x$ we obtain an ordinary function

$$y(t) = f(x,t),$$

which is a realization of the stochastic process van Kampen [1992]. It is customary in statistical physics to regard the stochastic process as an ensemble of such realizations.

It follows immediately from the random variable transformation theorem that the probability density for $Y(t)$ to take the value $y$ at time $t$ is given by

$$P_1(y,t) = \int dx \delta(y - f(x,t))P(x)$$

and, accordingly, the joint probability density that $Y$ has the value $y_1$ at time $t_1$, the value $y_2$ at time $t_2$, ..., and the value $y_n$ at time $t_n$ is given by

$$P_n(y_1,t_1; y_2,t_2; \ldots; y_n,t_n)$$

$$= \int dx_1 \delta(y_1 - f(x_1,t_1)) \delta(y_2 - f(x_2,t_2)) \cdots \delta(y_n - f(x_n,t_n))P(x).$$
In such a way an infinite hierarchy of joint probability densities \( P_n \) \( (n = 1, 2, \ldots) \) is defined, which allows the evaluation of expectation values like

\[
< Y(t_1)Y(t_2) \cdots Y(t_n) > \nonumber \\
= \int dy_1 \int dy_2 \cdots \int dy_n y_1 y_2 \cdots y_n P_n(y_1, t_1; y_2, t_2; \ldots; y_n, t_n).
\]

It has been shown by Kolmogorov (van Kampen [1992]) that the hierarchy of joint probability densities introduced above completely specifies a stochastic process if the following four consistency conditions are satisfied

(i) \( P_n \geq 0; \)

(ii) \( P_n \) is a symmetric function of the pairs \( (y_1, t_1), \ldots, (y_n, t_n); \)

(iii) \( \int dy_n P_n(y_1, t_1; \ldots; y_n, t_n) = P_{n-1}(y_1, t_1; \ldots, y_{n-1}, t_{n-1}); \)

(iv) \( \int dy_1 P(y_1, t_1) = 1. \)

Thus, the hierarchy of joint probability densities constitutes an alternative way to define stochastic processes. With increasing \( n \) the description of the stochastic process gets more precise. It is important to make the following remarks. The condition (iii) implies that each density \( P_n \) includes the knowledge of all previous densities \( P_k \) with \( k < n \). Furthermore, the density \( P_n \) does have the following property if two time arguments are identical

\[
P_n(x, t; y_1, t_1; y_2, t_2; \ldots; y_{n-1}, t_{n-1}) = P_{n-1}(x, t_1; y_2, t_2; \ldots; y_{n-1}, t_{n-1}) \delta(x - y_1).
\]

The hierarchy of probability densities is also the starting point for the classification of stochastic processes. A stochastic process is said to be purely random if events at different times are not correlated. In this case the joint probability density factorizes, i.e. we have

\[
P_2(y_1, t_1; y_2, t_2) = P_1(y_1, t_1)P_1(y_2, t_2)
\]

and so on.

This means that the value of \( Y \) at time \( t \) is completely independent of its values in the past and in the future. An even more special case occurs when the \( P_1(y_i, t_i) \) are independent of \( t_i \). In this case the same probability law governs the process for all times. Such processes are called Bernoulli trials (Gardiner [1990]). In the next section we will introduce the next most simple class, the Markov processes, in which the knowledge of only the present determines the future.

### 6.2 Markov Processes

In order to define the class of stochastic processes, which will be of central importance in the forthcoming theoretical discussions and in the examples of the next chapters, we will formulate the Markov assumption. This assumption is formulated in terms of conditional probability densities which we will denote by \( T_n(x, t; y_1, t_1; y_2, t_2; \ldots; y_n, t_n) \). This quantity gives the probability that the stochastic process takes the value \( x \) at time \( t \) given that it had the value \( y_1 \) at time \( t_1 \), \( y_2 \) at time \( t_2 \), \ldots, \( y_n \) at time \( t_n \), where we assume that \( t_1 < \cdots < t_n < t \). The conditional probability density has the following properties

(i) \( T_n \geq 0; \)

(ii) \( \int dx T_n = 1; \)

(iii) \( T_n(x, t; y_1, t_1; y_2, t_2; \ldots; y_n, t_n) = \delta(x - y_1). \)

As we already know the joint probability density \( P_n \) can be expressed with the help of Bayes’ theorem through the conditional probability density \( T_{n-1} \) as

\[
P_n(x, t; y_1, t_1; y_2, t_2; \ldots; y_{n-1}, t_{n-1})
\]

As we already know the joint probability density \( P_n \) can be expressed with the help of Bayes’ theorem through the conditional probability density \( T_{n-1} \) as

\[
P_n(x, t; y_1, t_1; y_2, t_2; \ldots; y_{n-1}, t_{n-1})
\]

Now we are in the position to define the class of Markov processes. Let \( t_1 < \cdots < t_n < t_{n+1} \) be an ordered sequence of times. A Markov process is defined through the following condition for the conditional
probability density of the stochastic process

\[ T_n(y_{n+1}, t_{n+1} | y_1, t_1; y_2, t_2; \ldots; y_n, t_n) = T_1(y_{n+1}, t_{n+1} | y_n, t_n). \]

In other words, the conditional probability density at \( t_{n+1} \) given the value of \( y_n \) at time \( t_n \) is uniquely determined and is not affected by any value of \( y \) at earlier times. Thus, the conditional probability density is determined completely by the knowledge of the most recent condition. The above definition implies that for a Markov process all \( T_n \) with \( n \geq 1 \) can be determined from the conditional probability density \( T_1 \), which will also be called the one step transition probability density. As an immediate consequence a Markov processes is completely characterized by the knowledge of the one step transition probability and by the probability density \( P_1 \). With the help of these two functions we can reconstruct the whole hierarchy of probability densities. For example, we have

\[ P_2(y_1, t_1; y_2, t_2; y_3, t_3) = P_2(y_1, t_1; y_2, t_2) T_2(y_3, t_3 | y_1, t_1; y_2, t_2) = T_1(y_3, t_3 | y_2, t_2) T_1(y_2, t_2 | y_1, t_1) P_1(y_1, t_1). \]  

(6.1)

Integrating the above equation (6.1) over \( y_2 \) we obtain

\[ P_2(y_1, t_1; y_3, t_3) = P_1(y_1, t_1) \int dy_2 T_1(y_3, t_3 | y_2, t_2) T_1(y_2, t_2 | y_1, t_1). \]  

(6.2)

Dividing both sides by \( P_1(y_1, t_1) \) we obtain an identity which must be obeyed by the transition probability of any Markov process

\[ T_1(y_3, t_3 | y_1, t_1) = \int dy_2 T_1(y_3, t_3 | y_2, t_2) T_1(y_2, t_2 | y_1, t_1). \]  

(6.3)

The above identity is called the Chapman–Komogorov equation. It has a simple interpretation. The transition probability between two states \( y_1 \) and \( y_3 \) with \( t_1 < t_3 \) corresponds to the product of the transition probability between the initial state and some intermediate state and the transition between this intermediate state and the final state integrated over all intermediate states.

As we already noted the functions \( P_1 \) and \( T_1 \) uniquely define a Markov process. However, these two functions are not arbitrary. They must satisfy the Chapman–Kolmogorov equation and the obvious consistency condition

\[ P_1(y_2, t_2) = \int dy_1 T_1(y_2, t_2 | y_1, t_1) P_1(y_1, t_1). \]  

(6.4)

For the sake of a compact notation we will write now \( P = P_1 \) and \( T = T_1 \).

### 6.3 The Differential Chapman–Kolmogorov Equation

We now derive a differential form of the Chapman–Kolmogorov equation which is more practical for physical applications. We will proceed in two steps. First, we introduce the concept of generator of a stochastic process. Second, we will construct with the help of the generator an equation of motion for the transition probability density.

#### 6.3.1 The Generator of a Markov Process

We consider the time evolution of the expectation value of a function \( f(y) \). Thus,

\[ \frac{\partial}{\partial t} \langle f(y) \rangle = \frac{\partial}{\partial t} \left\{ \int dy f(y) P(y, t) \right\} \]

\[ = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left\{ \int dy f(y) [P(y, t + \Delta t) - P(y, t)] \right\}. \]

### 6.3.2 Time evolution of the conditional probability density

We consider the time evolution of the conditional probability density \( T(y, t | y_0, t_0) \). Thus,

\[ \frac{\partial}{\partial t} T(y, t | y_0, t_0) = \frac{\partial}{\partial t} \left\{ \int dy f(y) T(y, t | y_0, t_0) \right\} \]

\[ = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left\{ \int dy f(y) [P(y, t + \Delta t | y_0, t_0) - P(y, t | y_0, t_0)] \right\}. \]
Making use of the consistency condition (6.4) in the first term on the right–hand side of the above equation we obtain
\[
\frac{\partial}{\partial t} \langle f(y) \rangle = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left\{ \int dy \int dy' f(y') T(y, t + \Delta t | y', t) P(y', t) - \int dy f(y) P(y, t) \right\}.
\]

We rename the integration variables in the positive term of the right–hand side of the above equation (\(y \to y', y' \to y\)) to obtain
\[
\frac{\partial}{\partial t} \langle f(y) \rangle = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left\{ \int dy' f(y') T(y', t + \Delta t | y, t) \delta(y - y') f(y) P(y, t) \right\},
\]
which we can also write as
\[
\frac{\partial}{\partial t} \langle f(y) \rangle = \int dy P(y, t) \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left\{ \int dy' f(y') T(y', t + \Delta t | y, t) \delta(y - y') f(y') \right\}.
\]

At this point it is convenient to introduce the infinitesimal generator of a Markov process \(A\) as
\[
A(t)f(x) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left[ \int dy f(y) T(y, t + \Delta t | x, t) - f(x) \right]. \tag{6.5}
\]

\(f(y)\) is some measurable function for which the above limit exists. Evidently \(A\) is a linear operator, which can be determined from the transition probability density. When the operator \(A\) operates on \(f\) it describes the change of the expectation value of \(f\) in an infinitesimal time step. As a consequence of the Chapman–Kolmogorov equation each time step \(t - t_1\) can be decomposed into a sequence of smaller time steps. So it is plausible to characterize the Markov process by regarding infinitesimal time steps. The importance of the generator \(A\) lies in the fact that together with some initial condition \(P(x, t = 0)\) it specifies uniquely the Markov process. The time evolution equation for the expectation value can be written in the compact and suggestive form
\[
\frac{\partial}{\partial t} \langle f \rangle = \langle Af \rangle. \tag{6.6}
\]

### 6.3.2 The Differential Chapman–Kolmogorov Equation

With the help of the generator we derive an equation of motion for the transition probability \(T\). Multiplying equation (6.5) with \(T(x, t | x', t')\) (\(t' < t\)) and integrating over \(x\) we obtain
\[
\int dx [A(t)f(x)] T(x, t | x', t') = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left[ \int dy f(y) T(y, t + \Delta t | x', t') - \int dx f(x) T(x, t | x', t') \right],
\]
where we made use of the Chapman–Kolmogorov equation. We now rename the variable \(y\) on the right–hand side of the above equation and call it \(x\) and perform the limit \(\Delta t \to 0\)
\[
\int dx [A(t)f(x)] T(x, t | x', t') = \int dx f(x) \frac{\partial}{\partial t} T(x, t | x', t'). \tag{6.7}
\]

It is convenient to introduce the adjoint operator \(A^\dagger\) to the generator \(A\) according to the following definition
\[
\int dx [A(t)f(x)] T(x, t | x', t') \equiv \int dx f(x) \left[ A^\dagger(t) T(x, t | x', t') \right]. \tag{6.8}
\]

We will see in the next sections how the adjoint operator is explicitly constructed. Inserting Eq. (6.8) into Eq. (6.7) and considering that (6.8) holds for any function \(f(x)\) we conclude that the equation of motion for the transition probability is given by
\[
\frac{\partial}{\partial t} T(x, t | x', t') = A^\dagger(t) T(x, t | x', t'). \tag{6.9}
\]
We will call the above equation the *differential Chapman–Kolmogorov equation*. The differential Chapman–Kolmogorov equation is the central equation of this chapter. Together with some initial probability distribution it defines completely a Markov process. With its help it is possible to compute time–dependent expectation values and multi–time correlation functions. Because of its importance, it is sometimes named the *master equation* in the physical literature.

Let us end this subsection with an overview of this chapter. In Fig. (6.3.2) we have schematically summerized the theory of stochastic processes. Particular emphasis is given to Markov processes which will be at the center of the present and of the next chapters. In the next section we will construct the differential Chapman–Kolmogorov equation for deterministic Markov processes, for jump processes and for diffusion processes.

### 6.4 The Liouville Equation

Let us consider a physical system whose dynamics is described by a system of ordinary differential equations of first order

\[
\frac{d}{dt} x(t) = g(x(t)),
\]

(6.10)

where \( g \) is a function \( \mathbb{R}^d \rightarrow \mathbb{R}^d \). It is clear that Hamiltonian systems belong to this class (Arnold [1978]). The initial condition is

\[ x(0) = x \in \mathbb{R}^d. \]

We denote the unique solution of this equation by \( \phi(t,x) \), where the \( x \) stresses the dependence on the initial condition.

If \( f : \mathbb{R}^d \rightarrow \mathbb{R}^d \) is a continuous differentiable function then it follows from Eq. (6.10)

\[
\frac{d}{dt} f(x(t)) = \sum_i \frac{\partial f}{\partial x_i}(x(t))g^i(x(t)),
\]

(6.11)

where \( g^i \) denotes the \( i \)-th component of \( g \).

With the help of these formal preliminaries it is easy to construct the generator of a deterministic Markov process. Obviously we have for the expectation value

\[ E f(x(t)) = f(\phi(x,t)), \]

where the symbol \( E \) denotes the expectation value.

Inserting the above expectation value into the definition of a generator (6.5) we immediately obtain

\[
A_L(t)f = \lim_{\Delta t \to 0} \frac{1}{\Delta t} [E f(x(t)) - f(x)] = \lim_{\Delta t \to 0} \frac{1}{\Delta t} [f(\phi(x,t)) - f(\phi(0,x))] = \frac{d}{dt} f(x),
\]

and finally, using Eq. (6.11),

\[
A_L(t)f = \sum_i \frac{\partial f}{\partial x_i}(x)g^i(x),
\]

Having determined the generator it is now straightforward to evaluate the corresponding differential Chapman–Kolmogorov equation. To this end we only have to determine the operator which is adjoint to
Figure 6.1: Overview of the theory of stochastic processes.
6.4. THE LIOUVILLE EQUATION

$A_L$ by partial integration. It is evident that we have

$$
\int dx [A_L f(x)] h(x) = \sum_i \int dx \left[ \sum_j \frac{\partial f}{\partial x_j} \right] h(x) \\
= - \sum_i f \frac{\partial}{\partial x_i} g^i h(x).
$$

(6.12)

Since Eq. (6.12) holds for any function $h(x)$

$$
A_L^ih(x) = - \sum_i \frac{\partial}{\partial x_i} (g_i h(x)).
$$

(6.13)

Inserting (6.13) into the differential Chapman–Kolmogorov equation (6.9) leads to the master equation for a deterministic Markov process

$$
\frac{\partial}{\partial t} T(x,t|x',t') = - \sum_i \frac{\partial}{\partial x_i} (g_i(x) T(x,t|x',t')).
$$

(6.14)

In statistical physics the above equation is called the Liouville equation.

The Liouville equation is the starting point for the microscopic description of matter for classical as well as for quantum mechanical systems. It is one of the fundamental equations of statistical physics.

6.4.1 Example: Classical Statistical Mechanics

In order to give an example of the occurrence of the Liouville equation we consider a closed classical system with $N$ degrees of freedom, e.g., $N$ particles in a three–dimensional box. We know from classical mechanics, that the state of such a system is completely specified by the set of $6N$ independent variables $p^N = (p_1, \ldots, p_N)$ and $q^N = (q_1, \ldots, q_N)$, where $p_i$ and $q_i$ denote the momentum and the position of the $i$–th particle.

If the system is Hamiltonian (Arnold [1978]), i.e., if we can define a Hamiltonian $H(p^N,q^N)$, then the time evolution of the momentum and of the position of the particles is given by Hamilton’s equations of motion

$$
\frac{d}{dt} p_i = \frac{\partial H}{\partial q_i},
$$

$$
\frac{d}{dt} q_i = \frac{\partial H}{\partial p_i}.
$$

In a real physical system it is not possible to specify exactly the state of the system. There is always some uncertainty in the initial conditions. Therefore, we regard $(p^N,q^N)$ as a stochastic variable which is initially distributed according to the joint probability density $P^N(p^N,q^N,0)$. The dynamics of this probability distribution is described by the following Liouville equation

$$
\frac{\partial}{\partial t} P^N = A_L^i P^N,
$$

where

$$
A_L^i = - \sum_{i=1}^N \left( \frac{\partial H}{\partial p_i} \cdot \frac{\partial}{\partial q_i} - \frac{\partial H}{\partial q_i} \cdot \frac{\partial}{\partial p_i} \right).
$$

The Liouville equation is often written in the following form

$$
i \frac{\partial}{\partial t} P^N(p^N,q^N,t) = LP^N(p^N,q^N,t),
$$
where

\[ L = iA_{\uparrow}^\dagger. \]

The operator \( L \) is called the Liouville operator. If the probability distribution at time \( t = 0 \) is known the above Liouville equation may be integrated formally to find the probability density at later times \( t \)

\[ P^N(x^N,q^N,t) = \exp(-iLt)P^N(x^N,q^N,0), \]

The Liouville equation is the starting point for the evaluation of probability distributions in statistical mechanics. Extensive use of the Liouville equation is done in kinetic theory. From the Liouville equation it is possible to derive a hierarchy of equations for probability densities, the so-called BBGKY–hierarchy from which kinetic equations may be derived (Reichl [1980]).

6.5 The Master Equation

Let us now introduce jump processes (Davies [1993]; Feller [1950]). We consider a system in a given state \( x \). In order to characterize a jump process, i.e., a process in which the system undergoes sudden discontinuous changes of its state, we have to specify the probability for the system to remain in \( x \) during the time interval \( dt \)

\[ (1 - \lambda(x)dt) \]

and the probability that the system jumps from state \( x \) to state \( x' \) during the time interval \( dt \)

\[ \lambda(x)Q(x',x)dt, \]

where

\[ \int dx'Q(x',x) = 1. \quad (6.15) \]

Then,

\[ Ef(x(dt + t)) = (1 - \lambda(x)dt)f(x) + \lambda(x)dt \int dx'f(x')Q(x',x). \]

From the definition of the generator we obtain immediately the generator of the jump process

\[ A_M(f(x)) = \lambda(x) \int dx' \left( f(x') - f(x) \right) Q(x',x), \quad (6.16) \]

where we made use of Eq. (6.15). Again, in order to derive the Kolmogorov forward equation we have to construct the adjoint operator to \( A_M \). We start from Eq. (6.7)

\[ \int dx[A_M(t)f(x)]T(x,t|x',t') = \int dx f(x) \frac{\partial}{\partial t} T(x,t|x',t'). \quad (6.17) \]

and insert the generator (6.16) into the left–hand side of the above equation

\[
\begin{align*}
\int dx[A_M(t)f(x)]T(x,t|x',t') &= \int dx \left[ \lambda(x) \int dx'' \left( f(x'') - f(x) \right) Q(x'',x) \right] T(x,t|x',t') \\
&= \int dx \int dx'' \lambda(x) f(x'') Q(x'',x)T(x,t|x',t') \\
&= \int dx \int dx'' \lambda(x) f(x) Q(x'',x)T(x,t|x',t') - \int dx \int dx'' \lambda(x) f(x) Q(x'',x)T(x,t|x',t').
\end{align*}
\]
By renaming $x \rightarrow x''$ and $x' \rightarrow x$ in the first line of the above equation we get

$$
\int dx f(x) \int dx'' \lambda(x') Q(x,x') T(x',t' | x',t')
- \int dx f(x) \int dx'' \lambda(x) Q(x',x) T(x,t | x',t')
\equiv \int dx f(x) \left[ A_{ij}^c(x) T(x,t | x',t') \right]
$$

(6.18)

From Eq. (6.17) and from Eq. (6.18) we conclude that the differential Chapman–Kolmogorov equation of a jump process reads

$$
\frac{\partial}{\partial t} T(x,t | x',t') =
\int dx'' \lambda(x'') Q(x,x'') T(x'',t'' | x',t'') - \int dx'' \lambda(x) Q(x',x) T(x,t | x',t'),
$$

(6.19)

Because of Eq. (6.15) we can write the above equation also in the form

$$
\frac{\partial}{\partial t} T(x,t | x',t') =
\int dx'' \lambda(x'') Q(x,x'') T(x'',t'' | x',t'') - \lambda(x) T(x,t | x',t'),
$$

(6.20)

Usually the differential Chapman–Kolmogorov equation for a jump process is written in a more suggestive form. To this end we introduce the total transition rate per time unit for a transition from state $x'$ into state $x$ to occur

$$
w(x,x') = \lambda(x') Q(x,x')
$$

and write the differential Chapman–Kolmogorov equation for a jump process in its final form

$$
\frac{\partial}{\partial t} T(x,t | x',t') = \int dx'' \left( w(x,x'') T(x'',t'' | x',t'') - w(x'',x) T(x,t | x',t') \right).
$$

(6.21)

The above equation is called the master equation. The name master equation appears for the first time in a paper by Nordsieck, Lamb and Uhlenbeck (Nordsieck et al. [1940]). It was chosen to denote an equation from which all relevant equations and results can be derived.

In the physical literature Eq. (6.21) is written in the simplified form

$$
\frac{\partial}{\partial t} P(x,t) = \int dx'' \left( w(x,x'') P(x'',t) - w(x'',x) P(x,t) \right).
$$

(6.22)

This equation has the following meaning ([van Kampen [1992]]). Take a time $t'$ and a state $x'$ and consider the solution of Eq. (6.22) for $t \geq t'$ with the initial condition $P(x,t') = \delta(x - x')$. This solution is the conditional transition probability $T(x,t | x',t')$ of the Markov process for each choice of $x'$ and $t'$. It is important to keep in mind that Eq. (6.22) is always to be interpreted as an equation for $T$ and not for $P$.

In the above form the physical meaning of the master equation is also evident. The master equation is a balance equation for the probability to find the system in some state. The first term in the master equation describes the gain of state due to transitions from the other states. The second term is the loss due to the transitions from the given state into the others. Evidently, the term with $x = x'$ does not contribute to the integral.

If the state space of the stochastic process is discrete, i.e. all integer numbers $n$, the master equation for the time evolution of $P(n,t)$ will assume the following discrete form

$$
\frac{\partial}{\partial t} P(n,t) = \sum_{n'} \left( w(n,n') P(n,t') - w(n',n) P(n,t) \right).
$$

(6.23)

We will discuss in the next section how the stochastic processes defined in terms of a master equation can be simulated numerically.
6.6 Stochastic Simulation

In principle there are two ways to treat numerically stochastic processes which are defined in terms of master equations. In the first approach one solves numerically the master equation as a differential equation for the probability density. Although this deterministic method is direct we will not consider it here for the following reason. It turns out that the direct numerical solution of the master equation is not particularly efficient from a computational point of view. The second approach, which we will introduce in this section relies upon the simulation of the underlying stochastic process. In other words one considers a particle and lets it jump from one state to another with certain given transition rates. Such a procedure is called the generation of a realization of the stochastic process. If a sufficiently great number of realizations has been generated the interesting statistical quantities can be evaluated as ensemble averages.

In fact the numerical performance of the direct approach gets worse compared to the stochastic simulation with increasing dimension of the system considered. We already met a similar situation, when we considered numerical algorithms for the computation of integrals. There the direct integration of multidimensional integrals turned out to be less efficient than the Monte–Carlo integration.

In order to formulate a stochastic simulation algorithm we consider for simplicity the master equation of a one–step process. Sometimes such processes are also called birth–and–death processes. The range of such processes consists of all integers \( n \) and the matrix of the transition probabilities per unit time allows only jumps between adjacent sites

\[
w(n', n) \neq 0 \quad \text{for} \quad n' = n \pm 1
\]

and

\[
w(n', n) = 0 \quad \text{otherwise}.
\]

Exploiting the above structure we can write the master equation for the discrete one–dimensional process (6.23)

\[
\dot{P}(n, t) = w(n, n+1)P(n+1, t) + w(n, n-1)P(n-1, t) - w(n+1, n)P(n, t) - w(n, n-1)P(n, t).
\]

Now it is convenient to introduce the following notation

\[
\begin{align*}
    r(n+1) &= w(n, n+1) \\
    g(n-1) &= w(n, n-1),
\end{align*}
\]

so that the master equation for a general one–step process may be written in the following suggestive form

\[
\dot{P}(n, t) = r(n+1)P(n+1, t) + g(n-1)P(n-1, t) - [r(n) + g(n)]P(n, t).
\] (6.24)

Let us now assume that at time \( t \) the particle is in state \( n \). The total transition rate to leave this state, i.e. to jump out of this state either to state \( n + 1 \) or to state \( n - 1 \), is given by

\[
\lambda(n) = r(n) + g(n).
\] (6.25)

As we already know the quantity \( \lambda(n)dt \) is the probability that the next jump occurs within the infinitesimal time step \( dt \). Accordingly the probability that the next jump occurs after \( (N + 1) \) time steps \( dt \) is given by

\[
q = [1 - \lambda(n)dt]^N \lambda(n)dt,
\]

where \([1 - \lambda(n)dt]^N\) denotes the probability that no jump occurs during the first \( N \) steps. We now write \((N + 1)dt = \tau\), so that we have

\[
q = \left(1 - \frac{\lambda(n)\tau}{N+1}\right)^N \lambda(n)\tau.
\]
We now perform the limit $dt \to 0$ for fixed $t$. This limit implies, of course, $N \to \infty$ and hence the probability that a jump occurs after time $t$ is given by

$$ q = \lambda \exp(-\lambda t) dt = f(t) dt, $$

The waiting time distribution for the time of the next jump is an exponential distribution. It is now clear what we have to do in order to determine the time of the next jump in the stochastic simulation. We simply have to draw random times $\tau$, which are distributed according to the density $f(t)$. Since $f(t)$ is an exponential distribution, these random times can easily be drawn with the help of the inversion method

$$ \tau = -\frac{1}{\kappa(n)} \log(\xi), $$

where $\xi$ is a uniformly distributed random number on the interval $[0, 1)$.

Having determined the stochastic time of the next jump we still have to decide which transition actually takes place. We have only two possibilities: In case of a jump the particle can reach the state $(n-1)$ with probability

$$ y_1 = \frac{r(n)}{\kappa(n)} $$

or the state $(n+1)$ with probability

$$ y_2 = 1 - y_1 = \frac{g(n)}{\kappa(n)}. $$

Thus, the algorithm for the simulation of a one–step–process reads:

1. Draw a uniformly distributed random number $\xi_1$ on $[0, 1)$ and compute the random jump time $\tau$ according to (6.26).
2. Draw a uniformly distributed random number $\xi_2$ on $[0, 1)$. If the condition ($\xi_2 < y_1$) is satisfied we set $\nu = +1$. Otherwise we set $\nu = -1$.
3. Advance the process

$$ t \longrightarrow t + \tau $$

$$ n \longrightarrow n + \nu. $$

The flow chart of the algorithm can be seen in Fig. (6.6). The algorithm has been implemented in the program `onestep.m` whose listing can be seen below. The program `onestep.m` makes use of a function called `decayymaster.m` in which the transitions rates $g(n)$ and $r(n)$ are specified. In the following subsection we will discuss three typical one–step processes. The specific form of the function `decaymaster` will be given there.

**Listing of the program onestep.m**

```matlab
% onestep - Program to simulate a one-step process
clear; help onestep; % Clear memory and print header
set (0,'DefaultAxesFontSize',16);

nstart = input('Enter initial value of N (500) - ');  
tend = input('Enter final time in s (30) - ');  
nreal = input('Enter number of realizations (10) - ');  
T_start = cputime;

nstart =0;  
nmes=zeros(1,tend+1);
```
Initialize:
\[ t \leftarrow t_0 \]
\[ n \leftarrow n_0 \]

while \( t < t_{\text{end}} \)

Pick random jump time:
\[ \tau = -\frac{\log(\xi_1)}{\lambda} \]

Pick increment:
\[ \text{if } (\xi_2 < y_1) \quad \nu = -1 \]
\[ \text{else } \nu = +1 \]

Advance the process:
\[ t \leftarrow t + \tau \]
\[ n \leftarrow n + \nu \]

record as required for sampling and plotting

final time reached?

all realizations done?

Plot Results

Figure 6.2: Flow chart of a stochastic simulation of a one–step process. The symbols used are explained in the text.
6.6. STOCHASTIC SIMULATION

nmes(1)= nstart;
nmes2=zeros(1, tend +1);
nmes4=zeros(1, tend +1);
15 nmes2(1)=0;
nmes4(1)=0;
times=[0: tend +1]:
% realizations loop
for j = 1: nreal
20 t=tstart;
n=nstart;
imes=2;
while (t<=tend)
% determine one-step jump probabilities per unit time
25 % transition rates for radioactive decay
% [g, r] = decaymaster(n);
% transition rates for the Poisson process
% [g, r] = poissonmaster(n);
30 % transition rates for the continuous time random walk
% [g, r] = walkmaster(n);
35 g=0.4*n;
r=0.5*n;
lambda=g+r;
y1=r/lambda;
% draw exponentially distributed random number
40 tau=-log(rand(1,1))/lambda;
45 t=t+tau;
while t>=times(imes)
46 nmes(imes)=nmes(imes)+n;
nmes2(imes)=nmes2(imes)+n*n;
nmes4(imes)=nmes4(imes)+n*n*n*n;
imes=imes+1;
end
50 if times>=tend+2
break
end
55 end
% end of time integration
end
% end of realization loop
60 end
% normalize mean values and variance
nmes=nmes/nreal;
nmes(1)=nmes(1)*nreal;
nmes2=nmes2/nreal;
nmes4=nmes4/nreal;
sdev(1)=0;
sdev2(1)=0;
for imes =2: tend+1
55 sdev(imes)= sqrt(((nmes2(imes)-nmes(imes)*nmes(imes))/nreal));
Thus we have found the following dynamical equation for the average of the stochastic variable
\[ \frac{d}{dt} \langle N(t) \rangle = -\gamma \langle N(t) \rangle. \]  

\section*{6.6.1 Radioactive Decay}

As a first example we consider the master equation description of the radioactive decay. To this end let \( P(n,t) \) be the probability density to find \( n \) radioactive nuclei at time \( t \). The probability for a nucleus to decay in unit time will be denoted by \( \gamma \). The radioactive decay is a typical example of a one–step process. It is defined through the transition rates \( g(n) \equiv 0 \) and \( r(n) \equiv \gamma n \), where \( \gamma \) is the decay constant. Substitution of the above \( g(n) \) and \( r(n) \) into the master equation for the one–step process (6.24) yields the master equation of the radioactive decay
\[ \frac{\partial}{\partial t} P(n,t) = \gamma (n+1)P(n+1,t) - \gamma n P(n,t). \]  

Before we apply the stochastic simulation algorithm to the generation of trajectories of the stochastic process we consider for one moment the above master equation.

The above equation has to be solved for the initial condition \( P(n,0) = \delta(n,n_0) \). It is interesting to establish the relation between the master equation and the macroscopic description in terms of differential equations we already met in the introduction. To this end we consider
\[
\sum_{n=0}^{\infty} n P(n,t) = \gamma \sum_{n=0}^{\infty} (n+1)P(n+1,t) - \gamma \sum_{n=0}^{\infty} n^2 P(n)
\]
\[
= \gamma \sum_{n=0}^{\infty} (n-1)P(n) - \gamma \sum_{n=0}^{\infty} n^2 P(n)
\]
\[
= -\gamma \sum_{n=0}^{\infty} n P(n).
\]

Thus we have found the following dynamical equation for the average of the stochastic variable \( N(t) \)
\[ \frac{d}{dt} \langle N(t) \rangle = -\gamma \langle N(t) \rangle. \]
6.6. STOCHASTIC SIMULATION

![Radioactive decay graph]

Figure 6.3: Stochastic simulation of radioactive decay. The initial number of decaying nuclei is \( n_0 = 100 \). \( t_{\text{end}} \) is 30 and the ensemble average was taken over 10 realizations. The decay rate is \( \gamma = 0.1 \).

Note that the mean value of the stochastic process obeys the differential equation for the concentration. It is clear that the above equation has the following solution for the initial value \( \langle N(0) \rangle = n_0 \)

\[ \langle N(t) \rangle = n_0 \exp(-\gamma t). \]

Let us now turn to the stochastic simulation. In order to use the program onestep.m we still have to specify the function decaymaster.m. The listing of this function can be seen below.

Listing of the function decaymaster.m

```matlab
function [g, r] = decaymaster(n)
% g = 1;
% r = 0;
gamma = 0.1;
g = 0;
r = gamma*n;
```

Now we are in the position to simulate the stochastic process of radioactive decay. We run the program for the following parameters \( n_0 = 500 \), \( t_{\text{end}} = 30 \), and \( n_{\text{real}} = 10 \). The decay rate is \( \gamma = 0.1 \). The result of the simulation can be seen in Fig. (6.6.1).

6.6.2 The Poisson Process

A further important one–step process is the Poisson process, which is defined by

\[ r(n) = 0; \quad g(n) = q = \text{const}. \]  \hspace{1cm} (6.30)

Inserting the above transition rates into the master equation (6.24) we get the master equation defining the Poisson process

\[ \frac{\partial}{\partial t} P(n, t|n', t') = q[P(n-1, t|n', t') - P(n, t|n', t')]. \]  \hspace{1cm} (6.31)
The Poisson process describes a random walk over the integers \(0, 1, 2, \ldots\). The steps of the walk are all of length \(l\) and are only to the right. They are performed at random times with probability per unit time equal to \(q\).

It is instructive to consider the analytical solution of the above master equation (6.31). The analytical solution will be constructive with the help of a very useful technique which is based upon the characteristic function

\[
G(s,t) = \langle \exp(is) \rangle = \sum_n P(n,t|n',t') \exp(is).
\]

The equation of motion for the characteristic function is easily obtained with the help of (6.31). We find

\[
\frac{\partial}{\partial t} G(s,t) = q[\exp(is) - 1] G(s,t),
\]

Assuming that at time \(t'\) the one-sided random walk starts at \(n' = 0\), the initial condition of the above differential equation reads

\[
G(s,0) = 1.
\]

In this case the solution of the above differential equation of motion for the characteristic function is easily found. It reads

\[
G(s,t) = \exp\{tq[\exp(is) - 1]\}.
\]

To read out the analytical solution for the transition probability \(P(n,t|0,0)\) it is convenient to write the above solution in the form

\[
G(s,t) = \exp(-tq) \exp[tq \exp(is)] = \sum_{n=0}^{\infty} \exp(-tq) \frac{(tq \exp(is))^n}{n!}.
\]

For \(s = 0\) and from the definition of the characteristic function it follows immediately that

\[
P(n,t|0,0) = \exp(-tq) \frac{(tq)^n}{n!},
\]

which evidently is a Poisson distribution. As we know the characteristic function allows the determination of all moments of the stochastic process through the relation

\[
\langle n^m(t) \rangle = \left(-i \frac{\partial}{\partial s}\right) G(s,t) \bigg|_{s=0}.
\]

Applying the above formula we find

\[
\langle n(t) \rangle = qt
\]

and

\[
\langle n^2(t) \rangle = qt + (qt)^2.
\]

Accordingly, the variance of the Poisson process is given by

\[
\text{Var}(n) = \langle n^2(t) \rangle - \langle n(t) \rangle^2 = qt.
\]

The simulation of the Poisson process is straightforward. We make use of the program \texttt{onestep.m} and simply write a new function \texttt{poissonmaster.m} according to the master equation (6.31) whose listing can be seen below.
Figure 6.4: Stochastic simulation of the Poisson process. The one-sided random walk starts at \( n_{\text{start}} = 0 \), \( t_{\text{end}} \) is 30 and \( n_{\text{real}} = 1 \). The jump rate is \( q = 1 \).

**Listing of the function poissonmaster.m**

```
function [g, r] = poissonmaster(n)

% q = 1.0;
q = 10.0;
g = q;
r = 0;
```

To begin we generate one realization of the Poisson process. To this end we run the program with the following parameters: \( n_{\text{start}} = 0 \), \( t_{\text{end}} = 30 \), \( n_{\text{real}} = 1 \). In the function `poissonmaster` we have chosen the transition rate to be \( q = 1 \). One realization of the Poisson process can be seen in Fig. (6.6.2). In Fig. (6.6.2) we show an ensemble average over 20 realizations of the Poisson process. The other parameters are unchanged. It is interesting to consider the limit of continuous space for the Poisson process. Let us denote the distance travelled by

\[ x = n l. \]

As a function of the new stochastic variable \( x \) the characteristic function \( \tilde{G} \) is

\[ \tilde{G}(s, t) = \langle \exp(i sx) \rangle = \exp \{ t q[\exp(is) - 1] \}. \]

We perform the limit \( l \to 0 \) keeping

\[ q l = v = \text{const} \]

and obtain keeping only the terms linear in \( l \) in the Taylor expansion of \( \exp(is) \)

\[ \lim_{l \to 0} \tilde{G}(s, t) = \exp(itv). \]

So, in the continuum limit the transition probability is

\[ P(x, t|0, 0) = \delta(x - vt). \]
Thus, the Poisson process has a deterministic limit. This can also be seen by considering, that in the same limit the master equation for the Poisson process turns into the Liouville equation

\[
\frac{\partial}{\partial t} P(x,t|0,0) = -q \frac{\partial}{\partial x} P(x,t|0,0),
\]

whose solution is the deterministic process we have just derived. This behaviour can also be seen in the simulation. To this end we run the program \texttt{onestep} with the function \texttt{poissonmaster} choosing \(q=10\). The result of the simulation can be seen in Fig. (6.6.2). As we can see for a larger value of \(q\) the variance of the process gets smaller. Thus the dynamics of the process is nearly deterministic.

### 6.6.3 The Continuous Time Random Walk

In this subsection we want to consider the master equation for the one–dimensional random walk. The steps of the walker are of length \(l\). The positions of the walker are \(nl\) and are labelled by the integer \(n\). We already considered the random walk problem in Chapter 2. There the walker was allowed to take steps to the left and to the right at some discrete times \(N\tau\), where the time step \(\tau\) was fixed. Now, we consider a random walk which is continuous in time. The walker is allowed to take steps to the left or to the right with the probability per unit time \(q\). This process is again described by a master equation for a one–step process (6.24) by choosing

\[
r(n) = g(n) = q.
\]

Thus the master equation for the continuous time random walk reads

\[
\frac{\partial}{\partial t} P(n,t|n',t') = q \{ P(n+1,t|n',t') + P(n-1,t|n',t') - 2P(n,t|n',t') \}, \tag{6.32}
\]

Again the above master equation is easily solved with the help of the characteristic function \(G(s,t)\). It is easy to check that the characteristic function satisfies the equation

\[
\frac{\partial}{\partial t} G(s,t) = q[\exp(is) + \exp(-is) - 2]G(s,t).
\]
6.6. STOCHASTIC SIMULATION

![Figure 6.6: Stochastic simulation of the Poisson process. The one-sided random walk starts at \( n_{\text{start}} = 0 \). \( t_{\text{end}} \) is 30 and \( n_{\text{real}} = 20 \). The jump rate is \( q = 10 \).](image)

Assuming that the walker starts at time \( t' = 0 \) in \( n' = 0 \) we find \( G(s,0) = 1 \) and the solution of the above equation reads

\[
G(s,t) = \exp\{[\exp(is) + \exp(-is) - 2]tq\}.
\]

With the help of the above expression for the characteristic function the moments are easily evaluated. We find

\[
\langle n(t) \rangle = 0,
\]

\[
\langle n^2(t) \rangle = 2tq.
\]

Again we find the typical behaviour of a diffusive process.

The continuous time random walk is also easily simulated with the help of the program \texttt{onestep}. To this end we have to write a new function \texttt{walkmaster.m} which implements the appropriate transition rates.

**Listing of the function \texttt{walkmaster.m}**

```matlab
function [ g, r] = walkmaster(n)
q = 1.0;
g=q;
r=q;
```

We run the program with the following parameters: \( n_{\text{start}} = 0, n_{\text{real}} = 10, t_{\text{end}} = 30, \) and \( q = 1 \). The result of the simulation can be seen in Fig. (6.6.3).

It is of particular interest to look at the continuous space limit of the continuous time random walk. Again we write for the distance travelled by the walker \( x = nl \). The characteristic function as a function of \( x \) reads

\[
\tilde{G}(s,t) = G(sl,t) = \langle \exp(is) \rangle
= \exp\{[\exp(is) + \exp(-is) - 2]tq\}.
\]
The limit of infinitesimally small steps \( l \rightarrow 0 (ql = \text{const.}) \) leads to
\[
\tilde{G}(s,t) = \exp(-s^2tD),
\]
where
\[
D = \lim_{l \rightarrow 0} (l^2q).
\]

The quantity \( D \) can be interpreted as the mean square distance traveled per unit time. The characteristic function (6.33) is the characteristic function of a Gaussian process of the form
\[
P(x,t|0,0) = \frac{1}{(4\pi Dt)^{1/2}} \exp\left(-\frac{x^2}{4Dt}\right).
\]
This is the transition probability of a so-called Wiener process, which can be regarded as a continuous time random walk in the limit of infinitesimally small step size. We will consider the Wiener process in more detail in the next section.

The following remark may serve as an introduction to the next section. Of course, we can also expand directly the master equation (6.32) as a function of \( x \) up to second order terms in \( l \) and get
\[
\frac{\partial}{\partial t}P(x,t|0,0) = (l^2q) \frac{\partial^2}{\partial x^2}P(x,t|0,0),
\]
which is a special type of Fokker–Planck equation, as we will see in the next section.

6.7 The Fokker–Planck Equation

In this section we want to derive the Fokker–Planck equation (Risken [1989]), which is a special type of master equation (van Kampen [1992]) in the limit of small jumps. We begin by expressing the transition probability \( w \) as a function of the size \( r \) of the jump and of the starting point
\[
w(y,y') = w(y';r); \quad r = y - y'.
\]
6.7. THE FOKKER–PLANCK EQUATION

The master equation (6.22) then reads

\[ \frac{\partial}{\partial t} P(y, t) = \int d\mathbf{r} (y - \mathbf{r}; r) P(y - \mathbf{r}, t) - \int d\mathbf{r} (y; -\mathbf{r}) P(y, t). \]  

(6.34)

In order to consider the limit of small jumps essentially two assumptions will be needed. The first is that the function \( w(y; r) \) will be a sharply peaked function of \( r \) and will vary slowly with \( y \). To be more precise we assume that a \( \delta > 0 \) exists such that

\[
w(y', r) \approx 0 \quad \text{for} \quad |r| > \delta
\]

\[
w(y' + \Delta y; r) \approx w(y'; r) \quad \text{for} \quad |\Delta y| < \delta.
\]

The second assumption is that the solution \( P(y, t) \) of the master equation in this limit will be a slowly varying function of \( y \).

If these assumptions hold it is safe to expand the shift from \( y \) to \( y - \mathbf{r} \) in the first integral in Eq. (6.34) in a Taylor series

\[
\frac{\partial}{\partial t} P(y, t) = \sum_{v=0}^{\infty} \frac{(-1)^v}{v!} \left( \frac{\partial}{\partial y} \right)^v \left\{ a_v(y) P(y, t) \right\} - P(y, t) \int_{-\infty}^{\infty} d\mathbf{r} w(y; -\mathbf{r}),
\]

(6.35)

where we have defined

\[
a_v(y) = \int_{-\infty}^{\infty} d\mathbf{r} r^v w(y; r).
\]

Since the zeroth term in the sum and the second term of Eq. (6.35) cancel the small jumps expansion of the master equation reads

\[
\frac{\partial}{\partial t} P(y, t) = \sum_{v=1}^{\infty} \frac{(-1)^v}{v!} \left( \frac{\partial}{\partial y} \right)^v \left\{ a_v(y) P(y, t) \right\}.
\]

(6.36)

The above expansion is called the Kramers–Moyal expansion. Formally, we can write

\[
\frac{\partial}{\partial t} P(y, t) = A_{KM}^+ (y) P(y, t),
\]

(6.37)

where we introduced the adjoint generator

\[
A_{KM}^+(y) = \sum_{v=1}^{\infty} \frac{(-1)^v}{v!} \left( \frac{\partial}{\partial y} \right)^v a_v(y).
\]

It is easy to show by partial integration that the corresponding generator reads

\[
A_{KM}(y) = \sum_{v=1}^{\infty} \frac{1}{v!} a_v(y) \left( \frac{\partial}{\partial y} \right)^v.
\]

It is clear that dealing with the Kramers–Moyal expansion will not be easier than dealing with the original master equation.

A particularly interesting and useful approximation to a jump process is obtained by keeping only terms up to the second order in \( v \)

\[
\frac{\partial}{\partial t} P(y, t) = -\frac{\partial}{\partial y} \left\{ a_1(y) P(y, t) \right\} + \frac{1}{2} \frac{\partial^2}{\partial y^2} \left\{ a_2(y) P(y, t) \right\}.
\]

(6.38)

The above equation is called the Fokker–Planck equation. The first term on the right–hand side is usually called the drift term since it is essentially Liouvillian. The second term on the right–hand side is the diffusion term. In a later section we will learn how to deal with this equation.
6.7.1 The Wiener Process

Let us consider the important special case of vanishing drift, i.e. \(a_1 = 0\), and diffusion coefficient equal to one, i.e. \(a_2 = 1\). The generator takes the form

\[
A_W = A_W^t = \frac{1}{2} \frac{\partial^2}{\partial y^2},
\]

and it defines a Wiener process. We already met the corresponding Fokker–Planck equation, when we considered the continuous space limit of the the continuous time random walk. Now we want to write down this Fokker–Planck equation in the form

\[
\frac{\partial}{\partial t} T(w,t|w_0,t_0) = \frac{1}{2} \frac{\partial^2}{\partial x^2} T(w,t|w_0,t_0).
\]

The above equation can easily be solved with the help of the characteristic function

\[
G(s,t) = \int dw T(w,t|w_0,t_0) \exp(isw),
\]

where we have assumed that the initial condition on the transition probability is

\[
T(w,t_0|w_0,t_0) = \delta(w - w_0).
\]

The characteristic function satisfies the equation

\[
\frac{\partial}{\partial t} G(s,t) = -\frac{1}{2} s^2 G(s,t),
\]

whose solution is, given the initial condition \(G(s,t_0) = \exp(isw_0)\),

\[
G(s,t) = \exp \left[ isw_0 - s^2 (t - t_0)/2 \right].
\]

Performing the inverse Fourier transformation of the above expression we find the transition probability

\[
T(w,t|w_0,t_0) = \frac{1}{\sqrt{2\pi(t - t_0)}} \exp \left\{ -\frac{(w - w_0)^2}{2(t - t_0)} \right\}. \tag{6.40}
\]

Thus, the transition probability density is a Gaussian with

\[
\langle W(t) \rangle = w_0, \quad \langle (W(t) - w_0)^2 \rangle = t - t_0.
\]

An initially sharp peaked distribution spreads in time. It is important to make some remarks on the Wiener process \(W(t)\).

The mean square of the Wiener process diverges linearly with time. As we already know this behaviour is typical for diffusion processes and the trajectories of the Wiener process are very variable. We will look at the trajectories of the Wiener process soon.

Although the paths of the Wiener process are continuous they are not differentiable since the derivative at any point is almost certainly infinite (Gardiner [1990]).

The Wiener process plays a central role in the description of diffusion processes by means of stochastic differential equations. The reason is that the increments of the Wiener process

\[
\Delta W_i \equiv W(t_i) - W(t_{i-1}) \equiv W_i - W_{i-1}
\]

are statistically independent. This can be seen in the following way. Let us consider the joint probability density

\[
P(w_n,t_n;w_{n-1},t_{n-1};\ldots;w_0,t_0) = \prod_{i=2}^{n-1} T(w_{i+1},t_{i+1}|w_{i+1},t_i) P(w_i,t_i). \tag{6.39}
\]
Exploiting the explicit form of the transition probabilities (6.40) the above joint probability density can be cast in the following form

\[
P(w_{n-1}, t_{n-1}; w_n, t_n; \ldots; w_0, t_0) = \prod_{i=0}^{n-1} \left\{ \frac{1}{\sqrt{2\pi(t_{i+1} - t_i)}} \exp\left[ -\frac{(w_{i+1} - w_i)^2}{2(t_{i+1} - t_i)} \right] \right\} P(w_0, t_0).
\]

Expressed in terms of the increments \(\Delta w_i\) the above equation reads

\[
P(\Delta w_n, \Delta t_n; \Delta w_{n-1}, \Delta t_{n-1}; \ldots; w_0, t_0) = \prod_{i=1}^{n} \left\{ \frac{1}{\sqrt{2\pi\Delta t_i}}} \exp\left[ -\frac{\Delta w_i^2}{2\Delta t_i} \right] \right\} P(w_0, t_0),
\]

where we have introduced the variables

\[
\Delta t_i = t_i - t_{i-1}.
\]

Thus, the increments \(\Delta W_i\) are evidently statistically independent and are distributed according to

\[
P(\Delta W, \Delta t) = \frac{1}{\sqrt{2\pi\Delta t}}} \exp\left[ -\frac{\Delta W^2}{2\Delta t} \right].
\]

At this point it is convenient to introduce a short hand notation for Gaussian distributed random numbers. The Gaussian random variable \(X\) with mean \(m\) and variance \(\sigma^2\) will be denoted by

\[
X \equiv N(m, \sigma^2).
\]

In particular we will name the random variable \(\xi\)

\[
\xi \equiv N(0, 1)
\]

the unit normal random variable. In this notation we can write for the Wiener \(W(t)\) process

\[
W(t) = N(W_0, (t - t_0))
\]

and for the increment \(\Delta W\)

\[
\Delta W(t) = N(0, \Delta t).
\]

For later convenience we give two rules concerning the transformation of Gaussian random variables. First, let \(a\) and \(b\) be two numbers, then we have

\[
a + bN(m, \sigma^2) = N(a + bm, b^2 \sigma^2).
\]

In particular we have for a unit normal random variable

\[
a + b\xi = N(a, b^2).
\]

Second, if \(N(m_1, \sigma_1^2)\) and \(N(m_2, \sigma_2^2)\) are statistically independent, then

\[
N(m_1, \sigma_1^2) + N(m_2, \sigma_2^2) = N(m_1 + m_2, \sigma_1^2 + \sigma_2^2).
\]

The above rule expresses the fact that, as we know, that Gaussian random variables remain Gaussian distributed under addition. The rules just given can be demonstrated with the help of the random variable transformation theorem.

With the help of the first above rule it is clear that the increment \(\Delta W\) can be expressed as

\[
\Delta W = \sqrt{\Delta t} \xi,
\]
where \( \xi \) is the unit normal random variable.

Let us now generate numerically some realizations of the Wiener process. To this end we need a way of calculating for a given value of the Wiener process \( W(t) \) at time \( t \) the value of the process at time \( t + \Delta t \). We will present an algorithm which is exact for any positive value of \( \Delta t \) (Gillespie [1992]).

The algorithm exploits the fact that we know analytically the conditional transition probability density. \( T(w(t)|w(t'),t') \) is a Gaussian with mean \( w' \) and variance \( \sigma^2 = 2(t - t') \). Accordingly, given \( W(t) \) the increment \( \Delta W(t) \)

\[
W(t + \Delta t) = W(t) + \Delta W(t)
\]

(6.41)
is distributed according to a Gaussian with zero mean and variance \( \sigma^2 = 2\Delta t \). The above formula is the update formula for the algorithm for the generation of realizations of the Wiener process. The algorithm essentially consists of the following steps:

(i) Let \( W(t) \) be given.
(ii) Draw a Gaussian distributed random number \( \Delta W \) with zero mean and variance \( \sigma^2 = 2\Delta t \).
(iii) Advance the stochastic process according to the formula (6.41).
(iv) Goto (i) until the desired final time is reached.

A flow diagram of the program \texttt{wiener.m} can be seen in Fig. (6.7.1).

**Listing of the function \texttt{wiener.m}**

```matlab
% wiener - Program to generate realizations of the Wiener process
clear; help wiener; % Clear memory and print header
set(0,'DefaultAxesFontSize',16);

xstart = input('Enter initial value of x (0) - ');
tend = input('Enter final time in s (5) - ');
deltat = input('Enter deltat in s (0.01) - ');
nreal = input('Enter number of realizations (1-1000) - ');
tstart = 0;
nstep = tend / deltat;
sigma = sqrt(deltat);
xmes = zeros(1, nstep + 1);
xmes2 = zeros(1, nstep + 1);
xmes4 = zeros(1, nstep + 1);

for j = 1:nreal
    t = tstart;
    x = xstart;
    % generate vector of gaussian distributed random numbers
    dw = randn(1, nstep);
    for i = 1:nstep
        x = x + sigma * dw(i);
        xmes(i+1) = xmes(i+1) + x;
        xmes2(i+1) = xmes2(i+1) + x**x;
        xmes4(i+1) = xmes4(i+1) + x**4;
    end
end
% normalize mean values and variance
xmes = xmes / nreal;
xmes(1) = xmes(1) * nreal;
xmes2 = xmes2 / nreal;
xmes4 = xmes4 / nreal;
sdev(1) = 0;
sdev2(1) = 0;
```

A flow diagram of the program \texttt{wiener.m} can be seen in Fig. (6.7.1).
6.7. THE FOKKER–PLANCK EQUATION

Initialize:
\[ t \leftarrow t_0 \]
\[ W \leftarrow W(t_0) \]

For \( t = t_{stat} : \Delta t : t_{end} \)

Draw Wiener increment according to
\[ (2\pi \Delta t)^{-1/2} \exp \left\{ -\frac{(\Delta W)^2}{2 \Delta t} \right\} \]

Advance the process:
\[ W \leftarrow W + \Delta W \]

record as required for sampling and plotting

final time reached?

NO

all realizations done

NO

YES

Plot Results

Figure 6.8: Flow diagram of the program wiener.m
Let us run the program two times with the following parameters \texttt{xstart}=0, \texttt{tend}=50, \texttt{deltat}=0.01, and \texttt{nreal}=1 in order to generate two realizations of the Wiener process. The two independent realizations can be seen on Figs. (6.7.1) and (6.7.1). The great variability of the realizations is evident. In Fig. (6.7.1) we show an ensemble average of the Wiener process over 1000 realizations.
Figure 6.10: One realization of the Wiener process. The parameters used in the simulation are $x_{start}=0$, $t_{end}=50$, $\delta t=0.01$, and $n_{real}=1$.

Figure 6.11: One realization of the Wiener process. The parameters used in the simulation are $x_{start}=0$, $t_{end}=5$, $\delta t=0.01$, and $n_{real}=1000$. 
6.7.2 The Ornstein–Uhlenbeck Process

Up to now we have considered only Fokker–Planck equations without drift. As a simple example of a Fokker–Planck equation with an additional linear drift we consider

$$\frac{\partial}{\partial t} T(x,t|x',t') = \frac{\partial}{\partial x} [q x T(x,t|x',t')] + \frac{1}{2} D \frac{\partial^2}{\partial x^2} T(x,t|x',t').$$  (6.42)

The above Fokker–Planck equation defines the Ornstein–Uhlenbeck process.

Again, we can look at the equation of motion for the characteristic function of the Ornstein–Uhlenbeck process

$$G(s,t) = \int_{-\infty}^{\infty} dx \exp(isx) T(x,t|x',t').$$

The equation reads

$$\frac{\partial}{\partial t} G(s,t) = -qs \frac{\partial}{\partial s} G(s,t) - \frac{1}{2} D s^2 G(s,t).$$

The above partial differential equation may be solved by the method of characteristics (Gardiner [1990]). Its solution for \( T(x,t_0|x',t_0) = \delta(x - x_0) \) requires the initial condition

$$G(s,0) = \exp(isx_0)$$

and reads

$$G(s,t) = \exp \left[ - \frac{Ds^2}{4q} \left[ 1 - \exp(-2q(t-t_0)) \right] + isx_0 \exp(-q(t-t_0)) \right].$$

Hence, the transition probability \( T(x,t|x',0) \) is a Gaussian with mean

$$\langle X(t) \rangle = x_0 \exp(-q(t-t_0))$$  (6.43)

and variance

$$\text{Var}[X(t)] = \frac{D}{2q} \left[ 1 - \exp(-2q(t-t_0)) \right].$$  (6.44)

Since the Ornstein–Uhlenbeck process is a Gaussian process we can write

$$X(t) = N \left( x_0 \exp(-2q(t-t_0))\frac{D}{2q} \left[ 1 - \exp(-2q(t-t_0)) \right] \right).$$

In contrast to the Wiener process the Ornstein–Uhlenbeck process has a stationary distribution in the limit \( t \to \infty \) which is a Gaussian with zero mean and variance \( D/2q \).

We now turn to the numerical simulation of realizations of the Ornstein–Uhlenbeck process. The problem is to find a way of determining from the value of the process \( X \) at time \( t \) its value at a later time \( t + \Delta t \). As for the generation of trajectories of the Wiener process it is possible to construct an update formula for the Ornstein–Uhlenbeck process, which is exact for any positive value of the time increment \( \Delta t \) (Gillespie [1996]). In order to derive an update formula we replace in Eqs. (6.43) and (6.44) for the mean and variance of the Ornstein–Uhlenbeck process \( t \) by \( t + \Delta t \) and \( t_0 \) by \( t \) and accordingly \( x_0 \) by \( X(t) \).

Since we know that the Ornstein–Uhlenbeck process is Gaussian distributed it is now clear that the update formula reads

$$X(t + \Delta t) = X(t) \exp(-qt) + \left[ \frac{D}{2q} \left[ 1 - \exp(-2q\Delta t) \right] \right]^{1/2} \xi,$$  (6.45)
where $\xi$ is a Gaussian distributed random number with zero mean and unit variance. Since we know how to generate Gaussian distributed random numbers the algorithm for the generation of realizations of the Ornstein–Uhlenbeck process with diffusion constant $D$ and inverse relaxation time $q$ reads

(i) Specify the values of $D$, $q$, $x_0$, $\Delta t$.
(ii) Compute the constant coefficients

$$\mu \equiv \exp(-q\Delta t)$$

and

$$\sigma \equiv \left[ \frac{D}{2q} \left[ 1 - \exp(-2q\Delta t) \right] \right]^{1/2}.$$ 

(iii) Initialize setting $X = x_0$ and $t = 0$.
(iv) Replace $t$ by $t + \Delta t$. Terminate the simulation if $t$ exceed $t_{end}$.
(v) Generate a Gaussian distributed random number $\xi$ and update the process replacing $X$ by $\mu X + \sigma \xi$.
(vi) Goto (iv).

In figure 6.7.2 we show the flow diagram of the program `ornstein.m` which will be used to generate the realizations. The listing of the program can be seen below.

**Figure 6.12:** The flow diagram of the simulation of the Ornstein-Uhlenbeck process.

### Listing of the program `ornstein.m`

```matlab
% ornstein - Program to generate realizations of the Ornstein-Uhlenbeck process with the help of an exact algorithm
clear; help ornstein; % Clear memory and print header

xstart = input('Enter initial value of x (5) - '); q = input('Enter value of drift q (1) - '); 
D = input('Enter value of diffusion constant D (1) - '); tend = input('Enter final time in s (5) - '); deltat = input('Enter delta t in s (0.01) - '); nreal = input('Enter number of realizations (1-1000) - ');
tstart = 0; nstep = tend / deltat;
sigma2=D*(1 - exp(-2*q*deltat ));
sigma2=sigma2/(2*q);
sigma=sqrt(sigma2);
muconst=exp(-q*deltat);
xmes=zeros(1, nstep +1);
xmes(1)=xstart;
xmes2=zeros(1, nstep +1);
xmes4=zeros(1, nstep +1);

% realizations loop
for j = 1: nreal
    t=tstart;
x=xstart;
    % generate vector of gaussian distributed random numbers
dw=randn(1, nstep);
    for i=1:nstep
```

---

6.7. THE FOKKER–PLANCK EQUATION

217
30 \[ x = x \cdot \mu_{\text{const}} + \sigma \cdot dw(i); \]
\[ x_{\text{mes}}(i+1) = x_{\text{mes}}(i+1) + x; \]
\[ x_{\text{mes}2}(i+1) = x_{\text{mes}2}(i+1) + x^2; \]
\[ x_{\text{mes}4}(i+1) = x_{\text{mes}4}(i+1) + x^4; \]
end
35 \% end of realization loop
end
% normalize mean values and variance
x_{\text{mes}} = x_{\text{mes}} / n_{\text{real}};
\[ x_{\text{mes}}(1) = x_{\text{mes}}(1) \cdot n_{\text{real}}; \]
40 x_{\text{mes}2} = x_{\text{mes}2} / n_{\text{real}};
\[ x_{\text{mes}4} = x_{\text{mes}4} / n_{\text{real}}; \]
\[ s\text{dev}(1) = 0; \]
\[ s\text{dev}2(1) = 0; \]
for \[ i_{\text{mes}} = 2: n_{\text{step}} + 1 \]
45 \[ s\text{dev}(i_{\text{mes}}) = \sqrt{((x_{\text{mes}2}(i_{\text{mes}}) - x_{\text{mes}}(i_{\text{mes}}) \cdot x_{\text{mes}}(i_{\text{mes}})) / n_{\text{real}});} \]
\[ s\text{dev}2(i_{\text{mes}}) = \sqrt{((x_{\text{mes}4}(i_{\text{mes}}) - x_{\text{mes}2}(i_{\text{mes}}) \cdot x_{\text{mes}2}(i_{\text{mes}})) / n_{\text{real}});} \]
end
% plot results
time = [0 : n_{\text{step}}] \cdot \delta_{\text{tat}};
50 if n_{\text{real}} == 1
\% plot one realization of the process
plot(time, x_{\text{mes}});
\% title (Ornstein–Uhlenbeck process');
\% xlabel ('t');
55 ylabel ('x');
else
\% plot result of simulation with errorbars
subplot(2,1,1)
errorbar(time, x_{\text{mes}}, s\text{dev});
60 title (Ornstein–Uhlenbeck process');
xlabel ('time');
ylabel ('<x>');
subplot(2,1,2)
errorbar(time, x_{\text{mes}2}, s\text{dev}2);
65 xlabel ('time');
ylabel ('<x^2>');
end

One realization of the Ornstein–Uhlenbeck process generated with help of the algorithm we have just constructed can be seen in Fig. (6.7.2)
The average over 10 realizations of the Ornstein–Uhlenbeck process can be seen in Fig. (6.7.2).

### 6.8 Lévy or Stable Distributions

In this section we address again the problem of the random walk. As we already know the canonical application of the random walk is the theory of Brownian motion. The chaotic motion of the Brownian particles over the length scale \( \Delta \) and time scale \( \tau \) is modelled by a random walk on a lattice of spacing \( \Delta \), while the steps take place at equal time intervals \( \tau \). It is the aim of this chapter to go "beyond Brownian motion" Klafter et al. [1996] and to look at fractal generalizations of Brownian motion which have proven to be a rich field in probability theory, statistical physics, chaotic dynamics, and, last not least, in economics.

We know already that from a mathematical point of view the problem of the random walk is the problem of the addition of independent (usually identically distributed) random variables. For example, if the individual steps in a one-dimensional walk have displacement \( \mu \) and variance \( \sigma^2 \), then a simple application of the Central Limit Theorem tells us that the asymptotic probability density function \( P_n \) for the position...
Figure 6.13: One realization of the Ornstein–Uhlenbeck process. The parameters used in the simulation are \(x_{\text{start}}=5, t_{\text{end}}=10, \Delta t=0.01, n_{\text{real}}=1, q=1, \) and \(D=1.\)

Figure 6.14: The average over 10 realizations of the Ornstein–Uhlenbeck process. The parameters used in the simulation are \(x_{\text{start}}=5, t_{\text{end}}=50, \Delta t=0.01, n_{\text{real}}=10, q=1, \) and \(D=1.\)
variable). As Lévy (recall that we have demonstrated in Chap. ??, that the sum of two Cauchy variables is again a Cauchy has the same probability density function as the variable $X$. This is the basic idea of the random walk. To put it differently, the Central Limit Theorem states, that if $X_1, \ldots, X_N$ are Gaussian distributed random variables with zero mean and variance $\sigma^2$, the new stochastic variable

$$X = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} X_i$$

has the same probability density function as the $X_i$. Now let us consider a slight generalization of the random walk. We consider an $N$–step random walk in one dimension, with each step of length $x$ governed by the same probability density $p(x)$, with zero mean. Note, that we expressively do not make any assumption about the variance! The French mathematician Paul Lévy posed the following question: When does the probability $p_N(x)$ for the sum of the steps

$$X = X_1 + X_2 + \cdots + X_N$$

in general have the same distribution $p(x)$ (up to a scale factor) as the individual steps? As remarked in Klafter et al. [1996] this is the fundamental question of fractals: When does the whole look like its parts?

?? Abschnitt unverständlich !!! Of course, we already know two answers to the above question. As we just recalled in the introductory remarks $p(x)$ may be a Gaussian density. And we know already that $p(x)$ may be a Cauchy density

$$\frac{1}{\pi} \frac{c}{x^2 + (x-x)^2}$$

(recall that we have demonstrated in Chap. ??, that the sum of two Cauchy variables is again a Cauchy variable). As Lévy has proved there exist other solutions. The remarkable aspect of these other solutions is that, as it is the case for the Cauchy density, all involve random variables with infinite variance!

The solutions that Lévy found are called stable (or Lévy) distributions. They play a constantly increasing role as a generalization of the normal distribution. In order to describe stable distributions and look at some of their properties we follow Feller [1966] and introduce the convenient notation

$$U^d = V$$

to indicate that the random variables $U$ and $V$ have the same distribution. Throughout this section $X$, $X_1$, $X_2$, \ldots, $X_N$ denote mutually independent random variables with a common distribution, say $R$ and $S_N = X_1 + \cdots + X_N$.

A distribution $R$ is called stable if for each $N$ there exist constants $c_N > 0$ and $\gamma_N$ such that

$$S_N^d = c_N X + \gamma_N$$

The distribution $R$ is called stable in the strict sense if (6.46) holds with $\gamma_N = 0$. The above definition can also be formulated equivalently in the following form: $R$ is stable if to arbitrary constants $c_1$, $c_2$ there exist constants $c$ and $\gamma$ such that

$$c_1 X_1 + c_2 X_2^d \equiv cX + \gamma.$$

Let us now look at some basic properties of stable distributions. For a proof of the statements that follow we refer the reader to Feller [1966]. The most important one is that the norming constants $c_N$ in Eq. (6.46) are of the form

$$c_N = N^{1/\alpha},$$

---

1Paul Lévy (186–1971)
with $0 < \alpha \leq 2$. The constant $\alpha$ is called the characteristic exponent of the stable distribution $R$. Sometimes it is also named the stability index.

An important property is that in practice the centering constants $\gamma_N$ may be disregarded. The reason for this fact is that we are free to center the distribution $R$ in an arbitrary manner, that is, we can replace $R(x)$ by $R(x+b)$. In fact, if $R$ is stable with an exponent $\alpha \neq 1$ the centering constant $b$ may be chosen so that $R(x+b)$ is strictly stable. To see why this is so we consider the random variable $S_{MN}$, which is a sum of $M$ independent variables each distributed according to $c_NX + \gamma_N$, i.e.,

$$S_{MN} = \sum_{i=1}^{N} c_NX_i + \gamma_N. \tag{6.47}$$

So we have

$$S_{MN} \overset{d}{=} c_NS_M + \gamma_{MN} \overset{d}{=} c_NC_MX + c_N\gamma_M + \gamma_{MN}. \tag{6.48}$$

Since $M$ and $N$ play the same role it follows from (6.47) and (6.48) that

$$(c_N - N)\gamma_M = (c_M - M)\gamma_N. \tag{6.49}$$

For $\alpha = 1$ the above equation does not have a solution, but when $\alpha \neq 1$ it implies that

$$\gamma_N = b(c_n - N) \tag{6.50}$$

for all $N$. It follows now from Eq. (6.46) that the sum $S'_N$ of $N$ variables distributed as $X' - b$ satisfies the condition

$$S'_N \overset{d}{=} X', \tag{6.51}$$

which completes our proof.

Let us consider $S_{M+N}$, i.e., the sum of the independent variables $S_M$ and $S_{M+N} - S_N$, which are distributed, respectively, as $c_MX$ and $c_NX$, i.e., we assume that $\gamma_n = 0$. For such a symmetric stable distribution we have the important relation

$$c_{M+N}X \overset{d}{=} c_MX_1 + c_NX_2. \tag{6.52}$$

An important relation follows from Eq. (6.49), namely

$$s^{1/\alpha}X_1 + t^{1/\alpha}X_2 \overset{d}{=} (s + t)^{1/\alpha}X, \tag{6.53}$$

whenever the ratio $s/t$ is rational. Since every stable distribution $R$ is easily shown to be continuous (see Feller [1966]) Eq. (6.50) holds for all $s > 0$ and $t > 0$. The meaning of the above equation is clear. For the normal distribution it simply restates the addition rule for the variances. In general, however, it implies that all linear combinations $a_1X_1 + a_2X_2$ belong to the same type.

The importance of the normal distribution stems from the Central Limit Theorem, which states that the normal distribution is the only stable distribution with variance! Remarkably similar limits may be formulated for distributions without variance. Only stable distributions occur as such limits. Consider for example a stable distribution with $\alpha < 1$. The average $(X_1 + \cdots + X_N)/N$ has the same distribution as $X_1N^{-1+1/\alpha}$, and the last factor tends to infinity. In other words the average of $N$ variables is likely to be larger than any of the components $X_i$. This is, of course, only possible if the maximal term $\max[X_1, \ldots, X_N]$ grows exceedingly large and receives a dominating influence on the sum $S_N$.

Thus, we are now able to answer the question raised by Lévy, which we mentioned at the beginning of the section. If $X_i, i = 1, \ldots, N$ are $N$ independent identically distributed Lévy random variables, with the same stability index $\alpha$, then the renormalized sum

$$S_N = \frac{1}{N^{1/\alpha}} \sum_{i=1}^{N} X_i$$
has the same probability density function as the $X_i$. It is important to remark that the sum scales as $N^{1/\alpha}$ and not as $\sqrt{N}$, as it is the case for a diffusive random walk.

Let us conclude by mentioning that we already met two examples of stable distributions, the normal distribution which corresponds to the case $\alpha = 2$ and the Cauchy distribution, which corresponds to the case $\alpha = 1$. For the case $\alpha = 1/2$ there is also a stable distribution with density

$$p(x) = \begin{cases} 0 & \text{for } x < 0 \\ \exp \left( -\frac{x^2}{\alpha x^2} \right) \left\{ \frac{x^2}{4\alpha x^2} \right\}^{1/2} & \text{for } x > 0 \end{cases}$$

and with norming constants $c_N = N^2$. This probability density is called the Smirnov density.

In the next section we are going to construct the most general form of all stable distributions, which is due to Lévy.

### 6.8.1 The Cauchy Process

**Bachelier’s Chain Equation**

Let us now consider a Markov process $X(t)$ and let its space of states be $R$. As we know the conditional transition probability $T(x|t, t')$ satisfies the Chapman–Kolmogorov equation. In order to derive a differential form of the Chapman–Kolmogorov equation to describe Brownian motion we had to assume that the moments

$$a_n = a_n(z, \Delta t) = \int (x - z)^n T(x, t + \Delta t | z, t) \, dx$$

exist and are convergent.

Let us now look at the so-called Cauchy process. A Cauchy process is defined through the propagator

$$T_C(x, t + \Delta t | z, t) = \frac{\Delta t}{\pi (x - z)^2 + \Delta t^2}.$$

It is easy to check that $T_C$ satisfies the necessary conditions of a propagator, namely

$$\int dx T_C(x, t + \Delta t | z, t) = 1$$

and

$$\lim_{\Delta t \to 0} T_C(x, t + \Delta t | z, t) = \delta(x - z).$$

The Cauchy process satisfies the Chapman–Kolmogorov equation and is therefore a Markov process. However, the moments $a_n$ do not exist for the Cauchy process and hence it is not possible to derive a differential form of the Chapman–Kolmogorov equation, i.e., there is no master equation describing the dynamics of the Cauchy process.

**The Realizations of the Cauchy Process**

In order to generate trajectories of the Cauchy process we first have to be able to generate Cauchy distributed random numbers. We have already seen in Chapter 2 that Cauchy distributed random numbers can be generated as the ratio of two gaussian distributed random numbers. Here we prefer to use a more efficient method which is based upon the inversion generating method (see the exercises for a comparison of the numerical performance of the two methods).

We recall that the inversion generating method is based upon the following idea. Let $X$ be a real random variable with density function $P(x)$ and distribution function $F(x)$. Then, if $r$ is a unit uniform random number, the random number $x$ obtained by solving the equation

$$F(x) = r,$$
i.e., the number
\[ x = F^{-1}(r) \]
is a sample value of \( X \). The Cauchy random variable \( C(m,a) \), defined by the density function
\[ P(x) = \frac{a}{\pi(x-m)^2 + a^2}, \]
where \( a, m \) are real numbers satisfying \( 0 < a < \infty \) and \( -\infty < m < \infty \). The variable \( X \) which is Cauchy distributed around \( m \) with half–width \( a \) has the distribution function
\[
F(x) = \int_{-\infty}^{x} \frac{a/\pi}{(x'-m)^2 + a^2} dx' = \frac{1}{\pi} \left[ \arctan \left( \frac{x-m}{a} \right) + \frac{\pi}{2} \right].
\]
Setting this equal to a unit interval uniform random number \( r \) and solving for \( x \) we obtain the generating formula
\[ x = m + a \tan \left( (r - \frac{1}{2})\pi \right). \]

We are now in the position to generate numerically some realizations of the Cauchy process. As it was the case for the Wiener and the Ornstein–Uhlenbeck process we will construct an algorithm, which is exact. Again the method is based on the fact that we have an analytical expression for the conditional transition probability \( T_{C}(x,t+\Delta t|z,t) \), which is itself a Cauchy density \( C(z,\Delta t) \). Since the Cauchy density satisfies the Chapman–Kolmogorov equation, paralling the construction of an exact algorithm for the Wiener process, we may use the fact that the increments of the Cauchy process
\[ \Delta C_{i} = C(i) - C(i-1) \]
are statistically independent and distributed according to a Cauchy density
\[ P(\Delta C, \Delta t) = \frac{\Delta t}{\pi} \frac{1}{(\Delta x)^2 + \Delta t^2}. \]
Obviously, the exact algorithm is straightforward and reads:
(i) Let \( C(t) \) be given.
(ii) Draw a Cauchy distributed random number \( \Delta C \) around 0 with half–width \( \Delta t \).
(iii) Advance the stochastic process according to
\[ C(t+\Delta t) = C(t) + \Delta C(t). \]
(iv) Goto (i) until the desired final time is reached.

The above algorithm has been implemented in the listing \texttt{CauchyProcess.java}, which can be seen below.

```java
import java.awt.Applet;
import java.awt.*;
import java.awt.event.*;
import ptplot.*;

public class CauchyProcess extends PlotApplet {
    private static int width = 500, height = 400;
```
private double dt = 0.01;                // time increment for saving
private double t_end = 100.0;            // private int steps = 100;
public double[ ] C_simu;                // the array for saving the realization

/** The main routine for running the program as an application */
public static void main(String[ ] args) {
    Applet applet = new CauchyProcess();
    Frame frame = new Frame("Cauchy_Process");
    frame.addWindowListener(new WindowAdapter() { // Handle window close requests
        public void windowClosing(WindowEvent e) {
            System.exit(0); // exit ?!
        }
    });
    frame.setSize(width, height); // set size of window
    frame.add("Center", applet); // add applet to the window
    frame.show(); // display window on screen
    applet.init(); // start applet
}

/** The actual main program, started by a browser or by the main method
Calculate a radioactive decay and plot the resulting points using
the PTPlot classes. Compare with the exact result. */
public void init() {
    int steps = (int) (t_end / dt) + 1;
    C_simu = new double[steps];
    C_simu[0] = 0.0;

    // Advance : time steps
    for (int t = 0; t < t_end; t++) {
        C_simu[t + 1] = C_simu[t] + dt * Math.tan((Math.random() - 0.5) * Math.PI);
        // Generate the increment according to the Cauchy distribution
        // The Cauchy random numbers are generated by the inversion method
    }

    // start anew plot and plot the points
    super.newPlot();
    super.init();
    int t_max = (int) t_end;
    plot().setTitle("Realization_of_the_Cauchy_Process"); // Title of plot
    plot().setMarksStyle("none"); // dots, points or various
    plot().setXLabel("time t"); // set the labels of the axes
    plot().setYLabel("x");
    plot().setXRange(0, t_max); // set the x range
    plot().setGrid(true); // Grid or not ?
    plot().setYLog(false); // logarithms plot ?
    plot().setBars(false); // should I use bars ?
    /* // Create the ticks for the axis
    for (int i = 0; i <= t_max; i++) {
        plot().addXTick(Integer.toString(i), i);
    }
    for (int i = 0; i <= N*0; i += 100) {
        plot().addYTick(Integer.toString(i), i);
    } */
    // plot the points and connect them
    boolean connect = false;
    for (int t = 0; t < t_end; t++) {
        plot().addPoint(0.1, C_simu[t], connect);
        if (connect == false) connect = true;
    }
}
In lines xx–yy we have coded the generation of the Cauchy distributed random variables according to the algorithm based on the inversion method. In line xx we advance the stochastic process. The realization is then plotted in lines xx to yy, with the help of the Pplot routines we already know.

Figure 6.15: Two possible realizations of the Cauchy process.

The illustration of the sample paths of the Cauchy process seen in Fig. (6.15) clearly indicates that the sample paths are discontinuous. Formally, this can be demonstrated by checking that the limit

$$\lim_{\Delta t \to 0} \frac{1}{\Delta t} \int_{|s| > \varepsilon} dx T(x, t + \Delta t | z, t) \neq 0$$

for any $\varepsilon > 0$. For continuous Markov processes the above limit can be shown to be zero with probability 1.

### 6.8.2 Lévy Processes

We have just seen that there exist Markov processes which cannot be described by a master equation. Such processes are generally called Lévy processes. We now want to characterize them in general Montroll and West [1979].

To this end we consider a Markov process $X(t)$ in the phase space $R$ and assume that it is homogeneous in time and space. In other words, we assume that the propagator satisfies the following condition

$$T(x, t | x', t') = P(x - x', t - t').$$

Hence the Chapman–Kolmogorov equation can be written as

$$P(x_2 - x_1, t) = \int dy P(x_2 - y, t_1) P(y - x_1, t - t_1),$$

(6.51)
It turns out that the characterization of Lévy process is best accomplished in Fourier space. To this end we now look at the characteristic function

\[ G(k,t) = \int_{-\infty}^{\infty} dx \exp(ikx)P(x,t). \]

From the Chapman–Kolmogorov equation (6.51) we can now derive a functional equation for the characteristic function

\[ G(k,t) = G(k,t - t_1)G(k,t_1), \]  

(6.52)

The characterization of Markov processes which are homogeneous in space and time is based upon the characterization of the solutions of the above functional equation. To this end we must find those solutions of (6.52) whose Fourier transformation is non-negative and normalized.

We already know two characteristic function satisfying Eq. (6.52), namely

\[ G(k,t) = \exp(-Dk^2t), \] 

which is the characteristic function of the Wiener process, and

\[ G(k,t) = \exp(-a|k|t), \] 

which is the characteristic function of the Cauchy process. The general solutions of Eq. (6.52) were investigated by P. Lévy. He found the Fourier transform of all strictly stable distributions which now bear his name. It is evident that the characteristic functions

\[ G(k,t) = \exp(-b|k|^\alpha t) \] 

for \(0 < \alpha \leq 2\) and \(b > 0\) are solutions of Eq. (6.52). However we have to check that the corresponding \(P(x,t)\) are probability densities. Formally we have

\[ P(x,t) = \frac{1}{2\pi} \int dk \exp(-b|k|^\alpha t - i k x). \]  

(6.53)

\(P(x,t)\) is normalized because

\[ \int dx P(x,t) = \frac{1}{2\pi} \int dk \int dx \exp(-b|k|^\alpha t - i k x)) 
= \int dk \delta(k) \exp(-b|k|^\alpha t) 
= 1. \]

Furthermore, it follows immediately from the formal expression (6.53) that

\[ P(x,0) = \delta(x). \]

In fact it can also be shown that the characteristic function defined in Eq. (6.52) leads to a positive density \(P(x,t)\) for \(0 < \alpha \leq 2\). This has been demonstrated by Bochner. The proof is trivial for \(0 < \alpha < 1\) but rather involved for the case \(1 < \alpha < 2\). So that we refer the interested reader to the original literature for the latter case.

For the case \(0 < \alpha < 1\) the proof is as follows:

\[ P(x,t) = \frac{1}{2\pi} \int dk \exp(-b|k|^\alpha t - i k x)) 
= \frac{1}{\pi} \int_0^\infty dk \cos(kx) \exp(-b|k|^\alpha t) 
= \frac{t b \alpha}{\pi x} \int_0^\infty dk k^{\alpha-1} \exp(-bk^\alpha t) \sin(kx). \]
By defining
\[ g(k) \equiv k^{\alpha-1} \exp(-bk^\alpha t) \]
we can write
\[
P(x,t) = \frac{tba}{\pi x} \int_0^\infty dk g(k) \sin(kx)
= \frac{tba}{\pi x} \sum_{n=0}^\infty \int_{2n\pi/x}^{2(n+1)\pi/x} g(k) \sin(kx)
= \frac{tba}{\pi x^2} \sum_{n=0}^\infty \int_0^{2\pi} du \left( \frac{u+2n\pi}{x} \right) \sin(u).
\]

For \(0 < \alpha < 1\) the function \(g\) is monotonically decreasing. For each value of \(\sin(u)\) in the \(u\)-interval \((0, \pi)\) there is a corresponding negative value of \(\sin(u)\) in the \(u\)-interval \((\pi, 2\pi)\). Since \(g\) is monotonically decreasing the positive contributions dominate over the negative ones. So we can conclude that \(P(x,t) > 0\) for \(t > 0\) and \(x \neq 0\). The case \(t = 0\) and the case \(x = 0\) are trivially satisfied.

The explicit calculation of the probability density \(P(x,t)\) is only possible for the special cases \(\alpha = 1\) (the Cauchy process), \(\alpha = 2\) (the Wiener process) and for the case \(\alpha = 1/2\) (the Smirnov density). These are exactly the densities we have considered as examples of stable distributions!

Let us conclude this section by remarking that the characteristic function \(G(k)\) may also be generalized by adding an imaginary part \(bc\)
\[ G(k,t) \equiv \exp[-tb|k|^{\alpha}(1 + icsign(k))]. \]
The factor \(sign(k)\) is introduced, in order to satisfy the necessary condition for characteristic functions
\[ G^s(k,t) = G(-k,t), \]
which guarantees that the probability density is real.

The requirement that the Fourier transform \(G(k,t)\) be a non–negative density function puts certain limit on \(c\), which have been studied by Khintchine and Lévy. Their results are: In order that a normalized, non–negative distribution function \(P(x,y,t)\) satisfy the Chapman-Kolmogorov equation it is necessary and sufficient that its characteristic function be represented by the formula (for \(t \leq 0\))
\[ \log(G(k,t)) = -(vk - bt|k|^{\alpha}\{1 + ico(k,\alpha)\text{sign} + i\mu k\}), \]
where \(\alpha, \ c, \ v, \ b\) are constants. \(v\) is any real number, \(-1 \leq c \leq 1, \ 0 \leq \alpha \leq 2, \ b \geq 0\) and
\[ \omega(k,\alpha) = \begin{cases} \tan(\pi\alpha/2) & \text{if } \alpha \neq 1, \\ (2/\pi)\log|k| & \text{if } \alpha = 1. \end{cases} \]
Lévy distributions with vanishing skewness parameter are called symmetric distributions. In the following we shall consider only symmetric distributions and we will denote them by \(S(x,\sigma,\mu)\), where \(\sigma\) denotes the scale parameter and \(\mu\) the shift.

Let us conclude this section with two remarks. Lévy distributions have in addition to their stability under convolutions two other interesting properties. Except for the gaussian (\(\alpha = 2\)) all \(\alpha\)–stable probability density functions have power–law tails with exponents \(1 + \alpha\). In other words, for large arguments \((x \gg 1)\) the asymptotic approximations of a Lévy stable distribution of index \(\alpha\) is given by
\[ P_\alpha(x) \approx C_\alpha(x^{\alpha})^{(1+\alpha)/\alpha}. \]
which evidently leads to an infinite variance and heavy tails. Thus stable distributions are characterized by a power–law behavior on the far wings of the distribution. The index \( \alpha \) does not only control the wings of the distribution, it also affects the value of the distribution at the origin

\[
P_\alpha(0) \approx \frac{\Gamma(1/\alpha)}{\pi \alpha} \quad (\gamma = 1!!)
\]

The stability under convolution gives rise also to another interesting property of Lévy distributions: the scale invariance of the process. If appropriately rescaled, the increment at scale \( N \tau \) will have the same distribution as the increment at scale \( \tau \)

\[
P_{\nu \tau}(x) = \frac{1}{\lambda} P_{\tau}(x/\lambda); \quad \lambda = N^{1/\alpha}.
\]

In other words the process \( x(t) \) is self–similar with a self–similarity exponent which is the inverse of the stability index \( \alpha \). This self–similarity structure of the Lévy distribution is at the basis of many interesting applications.

Several other interesting properties of the Lévy distributions can be found in the literature Montroll und Bendler [1984]; Bouchaud und Georges [1990].

### 6.8.3 The Numerical Generation of Levy Distributed Random Variables

In general the generation of stable random variables is quite an involved business. We will start our discussion by presenting an algorithm which allows the generation of random numbers distributed according to \( S_\alpha(1,0) \). The algorithm requires one random number, say \( \gamma \), uniformly distributed on the interval \((-\pi/2, \pi/2)\) and of an exponentially distributed random number \( W \) with mean 1. \( \gamma \) and \( W \) are assumed to be independent. The random number

\[
x = \frac{\sin(\alpha \gamma)}{(\cos(\gamma))^{1/\alpha}} \left( \frac{\cos((1-\alpha)\gamma)}{W} \right)^{(1-\alpha)/2}
\]

(6.54)

is distributed according to \( S_\alpha(1,0) \). It is easy to check that for the special case \( \alpha = 1 \) Eq. (6.54) reduces to

\[
x = \tan(\gamma),
\]

whose distribution is Cauchy. In the case \( \alpha = 2 \) Eq. (6.54) reduces to

\[
W^{1/2} \sin(2\gamma) / \cos(\gamma) = 2W^{1/2} \sin(\gamma),
\]

which corresponds to the Box-Muller method for the generation of \( N(0,2) \) random variables. The proof that Eq. (6.54) indeed generates \( S_\alpha(1,0) \) random variables can be found in [Samorodnitsky und Taqqu, 1994].

Knowing how to generate \( S_\alpha(1,0) \) distributed symmetric random variables it is clear that

\[
\alpha x + \mu \approx S_\alpha(\alpha x, \mu).
\]

[Mantegna, 1994]

### 6.9 Fractal Space Processes


#### 6.9.1 Levy Flights

As a first example of a random walk which does not belong to the class of Brownian motion, we consider a random walk process for which the variance of the jump length is infinite. The absence of a finite variance implies the absence of a characteristic length scale for the process. This makes such Lévy random walks, which are also called Lévy flights, scale invariant fractals. A particular illustrative and pedagogical example is the one–dimensional Weiertstrass random walk.
The Weierstrass Random Walk

The Weierstrass random walk is a discrete space–dimensional Lèvy flight. It represents a very simple model for a random process which generates self–similar clusters.

The Weierstrass random walk is generated by the transition probability density function \( p(r) \) for jumps of length \( r \)

\[
p(r) = \frac{a - 1}{2a} \sum_{m=0}^{\infty} a^{-m} \left[ \delta(r - \Delta b^m) + \delta(r + \Delta b^m) \right],
\]

with \( a > 1 \), \( b \) is an integer \((b > 1)\). The random walk takes place on a lattice with spacing \( \Delta \). In the following we will set for convenience this length scale equal to one. The above transition probability density allows jumps of length \( 1, b, b^2, b^3, \ldots \). However, when the length of the jump increases by an order of magnitude in base \( b \) the probability for the occurrence of such a jump decreases by a factor \( a \). Typically, we get a cluster of jumps of length \( 1 \) before a jump of length \( b \) occurs. About \( a \) such clusters separated by lengths of order \( b \) are found before one sees a jump of order \( b^2 \), and so on. In this scheme a step of length \( b^m \) is \( a \) times more likely then a step of length \( b^{m+1} \). In other words, we expect to see \( a \) clusters of size \( b^m \) per cluster of size \( b^{m+1} \).

Let us check that the variance of this random walk is indeed infinite. To this end we calculate the mean–square displacement per step. This quantity is given by

\[
\langle x^2 \rangle = \sum_{r=-\infty}^{\infty} r^2 p(r) = \frac{a - 1}{a} \sum_{m=0}^{\infty} \left( \frac{a}{a^2} \right)^m.
\]

Thus, the mean–square displacement is infinite if \( b^2 / a > 1 \). In the following we will assume that this condition is satisfied (For \( b^2 / a < 1 \) the variance \( \langle x^2 \rangle \) is finite and the random walk is described by a Gaussian diffusion process).

In order to investigate the qualitative behaviour of the Weierstrass random walk we look at the characteristic function of the process

\[
G(k) = \sum_{r=-\infty}^{\infty} \exp(irk) p(r),
\]

and we find

\[
G(k) = \frac{a - 1}{a} \sum_{m=0}^{\infty} a^{-m} \cos(b^m k).
\]

We recognize that \( G(k) \) is given by the famous Weierstrass function, which is everywhere continuous, but nowhere differentiable, when \( b > a \). In figure 6.16 we plot \( \sum_{m=0}^{M} a^{-m} \cos(b^m k) \) for \( M = 0 \) (bottom), 1, 2, 3, 4 (top) with \( a = 2 \) and \( b = 3 \). As is seen the adding higher order terms, the sum of the series fluctuates more wildly on smaller length scales.

It may be of interest prior to simulating the Weierstrass random walk to investigate the continuum limit of the Weierstrass random walk. to this end we must look at the small \( k \)–behaviour of the characteristic function. The characteristic function satisfies the following functional equation

\[
G(K) = a^{-1} G(bk) + \frac{a - 1}{a} \cos(k),
\]

which can be obtained by separating off the \( m = 0 \) term and reindexing the terms in the remaining series. it is easy to verify that if \( b^{2m} / a \neq 1 \) the solution of the above functional equation for any positive integer \( m \) reads

\[
G_b(k) = \frac{a - 1}{a} \sum_{m=0}^{\infty} \frac{(-1)^m}{(2m)!} \frac{k^{2m}}{[1 - b^{2m} / a]},
\]
The Weierstrass Function

Figure 6.16: The Weierstrass function $G(k)$ for $M=0,1,2,3,4,5$ in different colors. The parameters are $a=b=3$. 
$G_{k}(k)$ is holomorphic in the neighbourhood of $k = 0$. The general solution of the functional equation (6.56) reads Hughes [1995]

$$G(k) = G_{h}(k) + G_{s}(k),$$

where $G_{s}(k)$ satisfies the homogeneous equation

$$G_{s}(k) = a^{-1}G_{s}(bk).$$

Since the moments of the transition probability function are not all finite, the function $G(k)$ must be singular at $k = 0$. This singularity must reside in $G_{s}(k)$, the singular part of the characteristic function. To this end we focus on the homogeneous part of the functional equation. The solutions to this special equation have the form

$$G_{s}(k) = \text{const}|k|^{\mu},$$

where the exponent is given by

$$\mu = \frac{\ln a}{\ln b}.$$  

When $\mu < 2$ the small $k$–behaviour (large $r$) of $g(k)$ involves the exponent $\mu$, while for $\mu > 2$ the moments $< r^{2} >$ is finite and a Taylor expansion of $G(k)$ does exist. Summarizing it can be shown that

$$G_{h}(k) = \left\{ \begin{array}{ll}
1 - \text{const}|k|^{\mu} + O(k^{2}) \approx \exp(-|k|^{\mu}), & \text{for } \mu < 2 \\
1 - \frac{1}{2} < r^{2} > k^{2} \approx \exp(-|k|^{2}), & \text{for } \mu > 2.
\end{array} \right. \quad (6.57)$$

The characteristic function may now be used to calculate $P_{n}(s)$, the probability that the walker arrives at $s$ after the $n$–th step.

The probability $P_{n}(s)$ satisfies the following obvious recurrence formula, which is characteristic for all discrete-time random walks,

$$P_{n+1}(s) = \sum_{s'} p(s-s')P_{n}(s'). \quad (6.58)$$

It is easy to check that the necessary condition

$$\sum_{s} P_{n}(s) = 1$$

is satisfied. In the above sum the summation extends over all lattice points. It follows from Eq. (6.58) in the limit of lattice spacing $\Delta$ (which is now no longer assumed to be unity) and the time step $\tau$ going to zero

$$\lim_{\Delta,\tau \to 0} \frac{1}{\tau}[P_{n+1}(s) - P_{n}(s)] = \frac{\partial}{\partial t}P(x,t)$$

and

$$\lim_{\Delta,\tau \to 0} \sum_{s'} \left[ p(s-s') - \delta_{s,s'} \right] P_{n}(s') = \int_{-\infty}^{\infty} dx' P(x',t) \lim_{\Delta,\tau \to 0} \left[ p(x-x') - \delta(x-x') \right].$$

Introducing

$$P(k,t) = \int_{-\infty}^{\infty} dy P(y,t) \exp(iky),$$

the equation of motion reads

$$\frac{\partial}{\partial t}P(k,t) = \lim_{\Delta,\tau \to 0} \left[ \frac{G(k) - 1}{\tau} \right] P(k,t).$$
Hughes [1995] have demonstrated that for $\beta < 2$ the joint limits
\begin{align*}
a &= 1 + \alpha \Delta + O(\Delta) \\
b &= 1 + \beta \Delta + O(\Delta)
\end{align*}
and
\[
\lim_{\Delta \to 0} \frac{\Delta^\mu}{\tau} = D = \text{const} \quad 0 < \alpha < 2\beta
\]
yield
\[
\frac{\partial}{\partial t} P(k,t) = -D|k|^{\alpha/\beta} P(k,t), \tag{6.59}
\]
The solution of the above equation is clearly a Lévy characteristic function
\[
P(k,t) = \exp(-D|k|^{\mu}), \quad 0 < \mu < 2.
\]
Let us finally remark that the inverse Fourier transform of Eq. (6.59) yields the following integrodifferential equation
\[
\frac{\partial}{\partial t} P(x,t) = \frac{D}{\pi} \sin(\pi \mu / 2) \Gamma(\mu + 1) \int_{-\infty}^{\infty} dy \frac{P(y,t)}{|x-y|^{\mu+1}}
\]
as the evolution equation for the probability density of a Lévy process. Remarkably, the Lévy process appears to be nonlocal in the state space of the system and therefore no finite number of derivatives can be used to represent the kernel in the above equation. In fact the same equation can be associated to so-called fractional derivatives Zaslavski [1994]. Their consideration however is beyond the scope of the present book.

Lévy flights are realised in several physical systems. Th applications arise in different contexts ranging from the diffusion in micelles, to laser cooling. Last not least Lévy statistics finds application in finance. For a recent survey of the applications of Lévy statistics to physical systems see Ref. Shlesinger et al. [1995]. This reference contains also an article by Mandelbrot on Lévy.

### 6.10 The Continuous Time Random Walk

????????? Hier fehlt ein ganzes Stueck!!!!!!!!

#### 6.10.1 Lévy Walks

Because of the infinite moments Lévy flights have been ignored in the physical literature. It has been shown however that rather than focusing on this characteristic feature of Lévy flights one should concentrate on their scaling properties. The divergence of the moments can be tamed by associating a velocity to each Lévy flight trajectory segment. The reasonable question one has to pose is then: How far has a Lévy walker wandered from its starting point in time $t$? The answer to this question is a well–behaved time dependent moment of the corresponding probability density. In fact, as we will see shortly, a Lévy random walker moving with a velocity $v$, but with an infinite mean displacement per jump can have a mean square displacement from the origin that varies as $v^2 t^2$. To see this we make use of the continuous time random walk formalism.

Let $\Psi(r,t)$ be the probability density to make a jump of displacement $r$ in a time $t$. We write
\[
\Psi(r,t) = \psi(r|t)p(r) = p(r|t)\psi(t)
\]
where $p(r)$ is the probability density of a single jump and $\psi(t)$ has the same meaning as before and $\psi(r|t)$ and $p(r|t)$ are conditional probabilities for a jump taking a time $t$ given it is of distance $r$ and respectively for a jump being of distance $r$ given it took at time $t$. For simplicity we assume

$$\psi(r|t) = \delta(t - \frac{r}{v(r)}),$$

which ensures that $r = vt$. It is important to remark, that random walks with explicit velocities visit all points of the jump on the path between 0 and $r$. Such random walks are called Lévy walks in order to distinguish them from the Lévy flights, which visit only the end points of the jump. Note that the velocity need not be constant, it may as well be a function of $r$.

In 1926 Richardson formulated the law of turbulent diffusion which bears his name. The mean square separation $r$ between two particles in a turbulent flow grows like $t^3$. This is of course in contrast with the canonical result $<r^2(t)> = Dt$ which we know from the theory of Brownian motion. In the study of turbulent diffusion Kolmogorov assumed a scaling behaviour implying

$$v(r) \approx r^{1/3}.$$

If we furthermore assume

$$p(r) \approx |r|^{1+\beta},$$

which for small enough $\beta$ produces a Lévy flight with $<r^2> = \infty$, we find

$$<r^2(t)> = \begin{cases} 
  t^3, & \text{for } \beta \leq 1/3 \\
  t^{2+3(1-\beta)/2}, & \text{for } 1/3 \leq \beta \leq 5/3 \\
  t^2, & \text{for } \beta \geq 5/3.
\end{cases}$$

Thus we see that for $\beta \leq 1/3$ Richardson’s law may of turbulent diffusion may be reproduced. It corresponds to Lévy walk with Kolmogorov scaling for $v(r)$ combined with such a $\beta$ that the mean time spent by a segment of the trajectory is infinite.

The Lévy walk approach to turbulent diffusion provides a method for simulating trajectories of turbulent particles.

### 6.11 Exercises

**Exercise 6.1 Linear one-step process - quantized harmonic oscillator in a radiation field** [Kampen, 1992, page 143]

Let $n = 0, 1, 2, \ldots$, numerate the state of a quantized harmonic oscillator with energy $hv(n + 1/2)$. Transitions between the states are induced by the interaction of the oscillator with the radiation field. The transition probability is given by the dipole moments. The only allowed transitions according to the dipole moments are from $n$ to $n + 1$ and from $n$ to $n - 1$ (see quantum mechanics lecture).

Therefore the transition rates (probabilities per unit time) are:

- $g(n-1) = \beta n$ for the transition $(n-1) \rightarrow n$ and
- $r(n) = \alpha n$ for the transition $n \rightarrow (n-1)$.

$\alpha$ and $\beta$ are two constants, which depend only on the radiation density at the frequency $\nu$ and not on $n$.

Finally the Master equation for this special one-step process reads

$$\frac{\partial}{\partial t} P(n,t) = \alpha n P(n+1,t) + \beta (n+1) P(n+1,t) - (\alpha n + \beta (n+1)) P(n,t).$$

Write a program to simulate the given Master equation for the one-step process using the numerical scheme, learned in the lecture. Choose the parameters $\alpha$ and $\beta$. The result should be a plot of $P(n,t)$ for different $n$. Plot also $P(n)$ at large $t$, which gives the distribution of the harmonic oscillators with $n$ (and therefore the energy) in the steady state.

The exact stationary solution of the Master equation is $P_\infty(n) = \text{const} \cdot \left(\frac{n}{\alpha}\right)^\beta$.
**Exercise 6.2 Non-linear one-step process - growth of a competitive population [Kampen, 1992, page 163]**

The number of individuals of some species is called \( n \). The death rate for this population is \( \alpha \) and the birth rate (e.g. by fission) is \( \beta \). \( \alpha \) and \( \beta \) are fixed and independent of the age, otherwise it would not be a Markov process. Both rates are per unit time.

This would be still a linear one-step process. So consider an additional rate: the competition rate between the individuals of the population. This rate is an additional death rate and is \( \gamma(n-1) \).

Therefore the transition rates (probabilities per unit time) are:

- \( g(n) = \beta n \) for the transition \( n \rightarrow n+1 \)
- \( r(n) = \alpha n + \gamma n(n-1) \) for the transition \( n \rightarrow n-1 \).

\( \alpha, \beta \) and \( \gamma \) are constants, which do not depend on \( n \).

Finally the Master equation for this one-step process reads

\[
\frac{\partial}{\partial t} P(n,t) = (\alpha n + \gamma n(n-1))P(n+1,t) + \beta n P(n-1,t) - (\alpha n + \beta n + \gamma n(n-1))P(n,t).
\]

Write again a program to simulate the Master equation and again choose suitable parameters \( \alpha, \beta, \gamma \). View the time dependence of the population \( n \) and try to find out the behaviour of the stationary solutions.

The equation for the first moment (so called macroscopic equation) is called the Malthus-Verhulst equation and reads

\[
\langle n \rangle = (\beta - \alpha) n > - \gamma < n > ^2.
\]

If you neglect the nonlinear term on the right hand side, you get Malthus law. Then the solution is just an exponential growth of the population.

The solutions to the nonlinear equation can be calculated and you get

\[
\langle n \rangle = \frac{\beta - \alpha}{\gamma} \text{ and } \langle n \rangle \equiv 0,
\]

where the first one is the stable solution (so called attractor) and the second one is unstable.

**Example parameters:** \( \alpha = 0.5, \beta = 1, \gamma = 0.05 \).


The Random Telegraph process is the most simple Markov-process possible. It is a discrete process, which has only two possible states, called \( n = 0 \) and \( n = 1 \). The master-equation reads

\[
\frac{\partial}{\partial t} P(0,t \mid n') = b P(1,t \mid n') - a P(0,t \mid n') \quad (6.60)
\]
\[
\frac{\partial}{\partial t} P(1,t \mid n') = a P(0,t \mid n') - b P(1,t \mid n'). \quad (6.61)
\]
\[
\frac{\partial}{\partial t} P(0,t \mid n) = a P(0,t \mid n) - b P(1,t \mid n). \quad (6.62)
\]

\( a \) and \( b \) are the transition rates from state \( 0 \rightarrow 1 \) and \( 1 \rightarrow 0 \). Examples of this equation are processes, which jump from one state to the other and back (e.g. spin flipping).

We can rewrite the two above equations into one equation, resulting in the familiar master-equation - DO IT (you have to ?????) Then write a program to simulate the master-equation. Use \( n = 1 \) as the initial condition. For the long time behavior - the stationary solution - the initial condition is not significant. Try different settings for the parameters \( a \) and \( b \).

Compare the results of the simulation with the exact analytical results. For \( t \rightarrow \infty \) the stationary solution for the first moment is:

\[
P(0) = \frac{b}{a+b}, \quad P(1) = \frac{a}{a+b},
\]

and

\[
\langle n \rangle = \sum_{n=0}^{\infty} n P(n) = P(1) = \frac{a}{a+b}.
\]
And for the stationary covariance (for the second moment set \( t = t' \)) we get (\( t \geq t' \))

\[
<n(t)n(t')> = \left( \frac{a}{a + b} \right)^2 + \frac{ab}{(a + b)^2} e^{-(a+b)(t-t')}.
\]

Comment: This is an example of an ergodic process (a process with identical ensemble mean and time mean), where you can explicitly prove the ergodicity. Because if the correlation time is finite, the system is ergodic. And in this case the correlation time is

\[
t_c := \frac{1}{\text{var}(n(0))} \int_0^\infty dt |\text{var}(n(t))| = \frac{1}{(a+b)}
\]

and therefore finite.

**Exercise 6.4 Monomolecular Chemical Reaction** \( A \rightleftharpoons X \) [Schnakenberg, 1995, page 183]

A further example of a discrete one-step process is a chemical reaction, where an atom can be either bound to a molecule (call it state \( X \)) or be by itself (call it state \( A \)). We assume that we have an \( A \)-reservoir, so that there are always enough atoms to become absorbed by a molecule. The number of molecules (state \( X \)) is called \( N \), the number of atoms \( A \). Another example of this situation would be an atom in the ground state at a given temperature. The atom jumps to a higher state and back, depending on the temperature; assuming low temperatures (\( A \)-reservoir).

For chemical reactions the transition rates are given by the rate-constant \( k \), depending on the temperature. The derivation is based on the Stochahlsatz. So the transition of atoms to molecules is proportional to the number of atoms in the reservoir \( A \) and the transition of molecules releasing an atom is proportional to the number of molecules \( N \).

\[
W_{N+1,N} = A \quad \text{and} \quad W_{N-1,N} = kN^*.
\]

Then the master-equation is

\[
\frac{\partial}{\partial t} P(N,t) = AP(N-1,t) + k(N+1)P(N+1,t) - (A+kN)P(N,t).
\]

Again write a program to simulate the master-equation. Use \( k = 1, A = 100 \) for the parameters and \( N(0) = A \) as initial value. Then compare the simulation results with the analytical results:

\[
<N(t)> = A + \left( <N(0)> - A \right) e^{-t} \longrightarrow A \quad \text{for} \quad t \rightarrow \infty.
\]

\[
P_{\text{stat}}(N) = \frac{A^N}{N!} e^{-A} \quad \text{Poisson-distribution}.
\]

Also try different parameters and initial conditions.

Comment: In this example we assumed that there is enough time for the reactants to diffuse in the volume. That means the diffusion of atoms and molecules is very fast compared to the time a reaction takes place. If the two time scales are almost the same, we also have to simulate the diffusion. Such systems are known as reaction-diffusion systems - we will discuss them later on.

**Exercise 6.5**


Chapter 7

Stochastic Differential Equations

In the previous section we have derived an exact simulation algorithm for the generation of trajectories of the Ornstein–Uhlenbeck process. The “exact” update formula was

\[
X(t + \Delta t) = X(t) \exp(-q\Delta t) + \left[ \frac{D}{2q} (1 - \exp(-2q\Delta t)) \right]^{1/2} \xi(t),
\]

where we have now written \( \xi(t) \) to stress the fact that at each time step \( t \) we have to draw another Gaussian distributed random number. The update formula is exact in the sense that it holds for arbitrary values of \( \Delta t \).

However, it will turn out to be convenient to have an update formula which works for small values of \( \Delta t \). To this end we expand the exact update formula to first order in \( \Delta t \) and obtain

\[
\begin{align*}
X(t + \Delta t) &= X(t)(1 - q\Delta t) + \left[ \frac{D}{2q} (2q\Delta t) \right]^{1/2} \xi(t) \\
&= X(t) - qX(t)\Delta t + \sqrt{D\Delta t} \xi(t),
\end{align*}
\]

(7.1)

In the limit \( \Delta t \to 0 \) this approximate update formula turns exact. We recognize immediately that the stochastic increment in this discretized version of the Ornstein–Uhlenbeck process scales with the square root of the time increment \( \Delta t \).

Note that in deriving the above discretized update formula we have intentionally omitted the terms linear in \( \Delta t \) stemming from the expansion of the factor in front of the stochastic term. In doing so we have achieved that the update formula has an important property. Namely, it is selfconsistent in the following sense (Gillespie [1992]). Let us apply the above formula twice, starting from,

\[
X(t + 2\Delta t) = X(t + \Delta t) - qX(t + \Delta t)\Delta t + \sqrt{D\Delta t} \xi(t + \Delta t),
\]

Inserting (7.1) we immediately obtain keeping terms up to first order in \( \Delta t \)

\[
X(t + 2\Delta t) = X(t) - qX(t)2\Delta t + \sqrt{D\Delta t} [\xi(t) + \xi(t + \Delta t)],
\]

Since \( \xi(t) \) and \( \xi(t + \Delta t) \) are statistically independent Gaussian stochastic processes we have

\[
\xi(t) + \xi(t + \Delta t) = N(0,1) + N(0,1) = N(0,2) = \sqrt{2}N(0,1),
\]

so that we finally have

\[
X(t + 2\Delta t) = X(t) - qX(t)2\Delta t + \sqrt{D\Delta t} \xi(t),
\]

This selfconsistency of the discretized stochastic differential equation expresses essentially the fundamental properties of the propagator of a Markov process as they are defined in the Chapman–Kolmogorov equation.

Due to the presence of a stochastic term, the Gaussian stochastic process \( \xi(t) \), the above equation is a discretized version of a so–called stochastic differential equation (SDE). It is the aim of this section to
introduce into some peculiarities of stochastic differential equations. In particular we will also show the
equivocation of stochastic processes defined in terms of stochastic differential equations and in terms of
Fokker–Planck equations.

The above expression is a special case of the standard form of a stochastic differential equation (some
times stochastic differential equations are also called Langevin equations):

$$X(t + dt) = X(t) + A(X(t), t) dt + \sqrt{D(X(t), t)} \xi(t) dt^{1/2},$$

(7.2)

where we have replaced \( \Delta t \) by \( dt \) to stress the infinitesimal character of the above equation. The term
proportional to \( dt \) is called the drift term, whereas the term proportional to \( \sqrt{dt} \) is called the diffusion term.

The above definition of the stochastic process \( X(t) \) in terms of a stochastic differential equation clearly
shows that the stochastic process \( X(t) \) is continuous, but, in general, not differentiable. This can be seen
by writing Eq. (7.2) as

$$\frac{X(t + dt) - X(t)}{dt} = A(X(t), t) + \sqrt{D(X(t), t)} \xi(t),$$

Obviously, the limit \( dt \to 0 \) of the above equation does not exist, unless \( D \equiv 0 \). Thus, a purely stochastic
Markov process is everywhere continuous but nowhere differentiable. Nevertheless it is customary in the
physical literature to “pretend” (Gillespie 1992]) that \( dx/dt \) exists even for non vanishing \( D \). In fact we
know that we can write (see section 6.7.1)

$$\frac{\xi(t)}{\sqrt{dt}} = \frac{1}{\sqrt{dt}} N(0, 1) = N(0, 1/dt).$$

So, we may define a Gaussian white noise process as

$$\eta(t) \equiv \lim_{dt \to 0} N(0, 1/dt).$$

With the help of the above definition, we can now formally write (compare with 7)

$$\frac{d}{dt} X(t) = A(X(t), t) + \sqrt{D(X(t), t)} \eta(t).$$

This equation is called the white noise form of the Langevin equation. The white noise process introduced
above does have the following averaged properties:

$$\langle \eta(t) \rangle = 0$$

$$\langle \eta(t) \eta(t') \rangle = \delta(t - t'),$$

which satisfy the requirement of no correlation at different times. Note, that the white noise process \( \eta \) has
infinite variance. Accordingly, the spectral density, i.e., the Fourier transform of the correlation function of \( \eta \) is constant. This is the reason for calling \( \eta \) a white noise process.

It is important to establish precisely the relationship between the white noise process and the Wiener
process (Gillespie 1992]). We know already that the special Wiener process \( dW(dt) \) is a normal random
variable with mean zero and variance \( dt \)

$$dW(dt) = N(0, dt).$$

It follows from the theorems of Gaussian probability densities that

$$N(0, dt) = dtN(0, 1/dt).$$

Because of the definition of the Gaussian white noise process we can conclude that

$$dt \eta(t) = dW(dt)$$
and hence we have formally in the limit $dt \to 0$

$$\frac{dW}{dt} = \eta(t).$$

This equation asserts that the derivative of the Wiener process is the white noise process. However, we know already that the Wiener process is not differentiable so that the white noise process must be ill-defined. We will see shortly how these formal difficulties may easily be circumvented in the proper definition of stochastic differential equations. Before doing so we will consider for a moment the most classical Langevin equation of statistical physics, namely the one describing Brownian motion.

### 7.1 The Langevin Equation and Brownian Motion

In 1908 Langevin considered the problem of the dynamical description of Brownian motion (van Kampen [1992]). He suggested that the equation of motion of a Brownian particle with mass $m = 1$ be described by the following differential equation for the velocity $V$

$$\frac{d}{dt}V = -\gamma V + L(t), \quad (7.3)$$

where the terms on the right hand–side of the above equation model the forces which the surrounding molecules excerpt on the Brownian particle. Since these forces are unknown in detail the following assumptions were postulated. The Brownian particle moving in the fluid of surrounding particles feels a dissipative drag force which is proportional to its velocity, $\gamma$ being the friction coefficient. Furthermore, the Brownian particle hits the surrounding particles. These collisions cause irregular changes in the velocity of the Brownian particle. Thus, the external force $L(t)$ is modeled as a zero mean, temporally uncorrelated randomly fluctuating force. The first two moments of the stochastic process $L(t)$ are assumed to have the following properties

$$\langle L(t) \rangle = 0$$
$$\langle L(t)L(t') \rangle = \Gamma \delta(t-t').$$

The Langevin equation is the prototype of a stochastic differential equation, i.e. of a differential equation whose coefficients are random functions of the time with some given statistical properties. It is clear that choosing $L(t) = \sqrt{\Gamma} \eta(t)$, where $\eta(t)$ is a Gaussian white noise process, the Langevin equation of Brownian motion describes an Ornstein–Uhlenbeck process. The stochastic process $V(t)$ is completely defined once an initial condition $V(0) = V_0$ is specified. Its formal solution reads

$$V(t) = V_0 \exp(-\gamma t) + \exp(-\gamma t) \int_0^t dt' \exp(\gamma t') L(t'),$$

Taking the average over an ensemble of Brownian particles all having the same initial condition we find for the mean value of the velocity

$$\langle V(t) \rangle = V_0 \exp(-\gamma t),$$

where we made use of the statistical properties of the Langevin force $L(t)$. Accordingly, the second moment of the velocity field is found to be

$$\langle V^2(t) \rangle = V_0^2 \exp(-2\gamma t) + \exp(-2\gamma t) \int_0^t dt'' \int_0^t dt' \exp(\gamma (t' + t'')) \langle L(t')L(t'') \rangle$$

$$= V_0^2 \exp(-2\gamma t) + \frac{\Gamma}{2\gamma} [1 - \exp(-2\gamma t)].$$
Up to now the constant $\Gamma$ was left unspecified. From equilibrium statistical physics (theorem of equipartition of energy) we expect that for long times
\[ \langle V^2(t \to \infty) \rangle = kT. \]
Hence we have
\[ \Gamma = 2\gamma kT \] (7.4)
and we have established a relation between the attrition coefficient $\gamma$ and the random fluctuations. Eq. (7.4) is a simple version of the so-called fluctuation–dissipation theorem.

### 7.2 Stochastic Integration

It is the aim of this section to show how the formal problems arising in the formulation of Langevin equations can be avoided. Let us begin by formulating the Langevin equation in a discrete way,
\[ dX(t) = a(X(t),t)dt + b(X(t),t)\eta(t)dt, \]
where $a(X(t),t)$ is a deterministic drift and $b(X(t),t)$ is the diffusion term, $\eta(t)$ being a Gaussian white noise process. We proceed by integrating the above equation from $t_0$ to $t$ and obtain for each sample path
\[ X(t) = X(t_0) + \int_{t_0}^{t} da(X(s),s) + \int_{t_0}^{t} b(x(s),s)\eta(s)ds. \]
Since the Wiener process $W(t)$ can be represented as the integral over a white noise process, i.e.,
\[ W(t) = \int_{t_0}^{t} ds \eta(s) \] (7.5)
the integral form of the Langevin equation can be written as
\[ X(t) = X(t_0) + \int_{t_0}^{t} da(X(s),s) + \int_{t_0}^{t} b(x(s),s)dW(s). \] (7.6)
The above expression is expected to make sense because the Wiener process is continuous. We will see shortly that the above equation does have a precise meaning. In fact from here on a solution of a stochastic differential equation will be interpreted as a solution of the corresponding integral equation, which will be written in the short-hand notation
\[ dX(t) = a(X(t),t)dt + b(x(t),t)dW(t). \]

Of course, the second integral in Eq. (7.6) is not an ordinary integral. It is an integral with respect to the Wiener process $W(t)$. Such integrals are called stochastic integrals and we will define and discuss them now. For a precise mathematical definition of stochastic integrals see Gard [1988]; Potter [1990]; Kloeden und Platen [1992]; Öttinger [1996]. We will follow here the more physical line of reasoning of Gardiner [1990].

#### 7.2.1 Definition of the Stochastic Ito Integral

The starting point for the definition of the Ito integral is the following reasoning. For $b(X(t),t) = b = \text{const}$, the stochastic integral
\[ I = \int_{t_0}^{t} b dW(s) \]
is expected to be defined and to be equal to

\[ I = b\{W(t) - W(t_0)\}. \]

In general it seems to be safe to treat the stochastic integral

\[ I(f) = \int_{t_0}^{t} f(X(s), s) dW(s) \]

as a kind of Riemann–Stieltjes integral, i.e., as a limit of partial sums. To do so we divide the interval \([t_0, t]\) into \(n\) subintervals

\[ t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_{i-1} \leq t_i \equiv t \]

and define intermediate points \(\tau_i\)

\[ t_{i-1} \leq \tau_i \leq t_i. \]

The stochastic integral \(I(f)\) is then defined as the limit of the partial sums

\[ S_n = \sum_{i=1}^{n} f(\tau_i) [W(t_i) - W(t_{i-1})]. \]

In general it turns out that the definition of the stochastic integral depends on the particular choice of the intermediate point \(\tau_i\). In the definition of the Ito stochastic integral the intermediate points are chosen to be at the beginning of the corresponding time interval, i.e.,

\[ \tau_i = t_{i-1}. \]

Accordingly the Ito stochastic integral is defined as the limit of the partial sums

\[ S_n = \sum_{i=1}^{n} f(t_{i-1}) [W(t_i) - W(t_{i-1})]. \]

The limit of the sequence of partial sums is to be understood in the following sense. The random variable \(S_n\) is said to converge to \(S\) in the mean square limit if

\[ \lim_{n \to \infty} \langle (S_n - S)^2 \rangle = 0, \]

The above limit is usually written as

\[ \text{ms- lim}_{n \to \infty} S_n = S. \]

In this sense the Ito stochastic integral of the function \(f(t)\) is defined as

\[ \int_{t_0}^{t} f(x(t'), s') dW(t') = \text{ms- lim}_{n \to \infty} \left\{ \sum_{i=1}^{n} f(t_{i-1}) [W(t_i) - W(t_{i-1})] \right\}. \]

### 7.2.2 The Stratonovich Stochastic Integral

An alternative definition of a stochastic integral has been given by Stratonovich. He suggested the following definition

\[ S \int_{t_0}^{t} f(x(t'), s') dW(t') = \text{ms- lim}_{n \to \infty} \left\{ \sum_{i=1}^{n} f(\frac{x(t_i) + x(t_{i-1})}{2}, t_{i-1}) [W(t_i) - W(t_{i-1})] \right\}. \]

The \(S\) in front of the integral denotes a Stratonovich integral in contrast to the Ito integral. Note, that in this definition the integrand is evaluated in an averaged way.
7.2.3 Ito Calculus

We now want to derive some very useful formulas. In order to do so we have to introduce a special class of functions. A function \( g(t) \) is called a nonanticipating function of \( t \) if for all \( s \) and \( t \) such that \( t < s \), \( g(t) \) is statistically independent of \( W(s) - W(t) \). In other words \( g(t) \) is independent of the behaviour of the Wiener process in the future of \( t \). Within the context of the stochastic differential equations such functions are quite reasonable since they express the fact that the future does not affect the present. This guarantees, evidently, causality.

We are now in the position to give the proof of the fundamental equation of Ito calculus, namely, that

\[
dW(t)^2 = dt
\]

and that

\[
dW(t)^2 + N = 0
\]

for \( N \geq 1 \). These formulas will allow for a comfortable handling of stochastic differentials.

We begin by proving that

\[
\int_0^t \left[ dW(t') \right]^2 g(t') = \int_0^t dt' g(t')
\]

(7.8)

for a nonanticipating function \( g(t) \). By definition of the stochastic Ito integral we have

\[
\int_0^t \left[ dW(t') \right]^2 g(t') = \lim_{n \to \infty} \sum_i g_{i-1} \Delta W_i^2
\]

\[
= \lim_{n \to \infty} \left[ \sum_i g_{i-1} \Delta W_i^2 \right] >.
\]

Eq. (7.8) is of course to be understood in the mean square sense, so we consider the following expression

\[
I = \lim_{n \to \infty} \left[ \sum_i \Delta W_i^2 - \Delta t_i \right] >
\]

\[
= \lim_{n \to \infty} \left[ \sum_i (g_{i-1})^2 (\Delta W_i^2 - \Delta t_i) + \sum_{i \neq j} 2 g_{i-1} g_{j-1} (\Delta W_i^2 - \Delta t_i) (\Delta W_j^2 - \Delta t_j) >.
\]

We can now exploit the fact that in the first sum in the above expression the \((g_{i-1})^2\) and \((\Delta W_i^2 - \Delta t_i)^2\) and accordingly in the second sum \(g_{i-1} g_{j-1} (\Delta W_i^2 - \Delta t_i)\) and \((\Delta W_i^2 - \Delta t_i)\) are statistically independent from each other because the function \(g\) is nonanticipating and because of the properties of the Wiener process. This statistical independence permits to factorize the mean value. So we find

\[
I = 2 \lim_{n \to \infty} \left[ \sum_i \Delta t_i \left( (g_{i-1})^2 \right) >
\]

where we have used the following properties of the Wiener process

\[
\langle \Delta W_i^2 \rangle = \Delta t_i
\]

and

\[
\langle (\Delta W_i^2 - \Delta t_i)^2 \rangle = 2\Delta t_i^2.
\]
Table 7.1 Multiplication table for products of stochastic differentials.

<table>
<thead>
<tr>
<th>×</th>
<th>dW</th>
<th>dW²</th>
<th>dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>dW</td>
<td>dt</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dW²</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dt</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Hence we can conclude that

\[ \text{ms-\lim}_{n \to \infty} \left( \sum_i g_{i-1} \Delta W_i^2 - \sum_i g_{i-1} \Delta t_i \right) = 0. \]

Since

\[ \text{ms-\lim}_{n \to \infty} \sum_i g_{i-1} \Delta t_i = \int_{t_0}^{t} dt' g(t') \]

we have completed the proof of Eq. (7.8). The importance of Eq. (7.8) is the following one. Because of the definition of stochastic differential equations \( dW(t) \) occurs only in integrals, so that we can explicitly write

\[ dW(t)^2 \equiv dt. \]

Accordingly, it is straightforward to show that in the same sense

\[ dW(t)^2+\lambda \equiv 0, \quad \text{for} \quad \lambda > 0. \]

In the following it will be of some importance to have multiplication rules for stochastic differentials. The following multiplication table sums up the rules for products of stochastic differentials. As an example of the application of the above formulas we consider the integration of a polynomial. Let us look at

\[ d[W(t)]^n = [W(t) + dW(t)]^n - W(t)^n = \sum_{r=1}^{n} \binom{n}{r} W(t)^{n-r} dW(t)^r. \]

Using the fact that \( dW(t)^r = 0 \) for \( r > 2 \) we conclude that

\[ d[W(t)]^n = nW(t)^{n-1} dW(t) + \frac{n(n-1)}{2} W(t)^{n-2} dt \]

so that

\[ \int_{t_0}^{t} W(t')^n dW(t') = \frac{1}{n+1} [W(t)^{n+1} - W(t_0)^{n+1}] - \frac{n}{2} \int_{t_0}^{t} W(t')^{n-1} dt'. \]

### 7.3 Ito Stochastic Differential Equations

Having defined stochastic integrals the proper definition of a stochastic differential equation can be given (Again we follow Gardiner [1990]. The mathematically interested reader should consult Gard [1988]; Kloeden und Platen [1992]; Potter [1990]). The stochastic variable \( X(t) \) obeys the Ito stochastic differential equation

\[ dX(t) = a(X(t),t) dt + b(X(t),t) dW(t) \quad (7.9) \]
if for all \( t \) and \( t_0 \) the following integral equation holds

\[
X(t) = X(t_0) + \int_{t_0}^{t} da(X(s),s) + \int_{t_0}^{t} dW(s)b(X(s),s).
\]  

(7.10)

### 7.3.1 Ito’s Formula

In this subsection we want to consider a function \( f \) of the stochastic variable \( X(t) \) and derive an Ito stochastic differential equation for \( f \). We begin by expanding the differential \( df(x(t)) \) to second order in \( dW(t) \)

\[
\begin{align*}
    df(X(t)) &= f(X(t) + dX(t)) - f(X(t)) \\
    &= f'(X(t))dX(t) + \frac{1}{2}f''(X(t))dX(t)^2 + \ldots.
\end{align*}
\]

Inserting the Ito stochastic differential equation (7.9) for \( dX(t) \) we get

\[
\begin{align*}
    df(X(t)) &= f'(X(t))[a(X(t),t)dt + b(X(t),t)dW(t)] \\
    &+ \frac{1}{2}f''(X(t))b(X(t),t)^2[dW(t)]^2 + \ldots,
\end{align*}
\]

where we have discarded all other terms of higher order. Using finally \([dW(t)]^2 = dt \) we get

\[
\begin{align*}
    df(X(t)) &= \{a(X(t),t)f'(X(t)) + \frac{1}{2}f''(X(t))b(X(t),t)^2\}dt \\
    &+ b(X(t),t)f'(X(t))dW(t).
\end{align*}
\]

(7.11)

The above equation is Ito’s formula and expresses the fact that in general for stochastic differential equations the change of variables is not given by the rules of ordinary calculus.

### 7.3.2 The Equivalence of Stochastic Differential Equations and of the Fokker–Planck Equation

Let us now look at the time development of the expectation value of an arbitrary function \( f(X(t)) \). Using Ito’s formula we immediately have

\[
< df(X(t)) > = < \frac{df(X(t))}{dt} > = \frac{d}{dt} < f(X(t)) >
\]

\[
= < a(X(t),t)f'(X(t)) + \frac{1}{2}f''(X(t))b(X(t),t)^2 >.
\]

Since, \( X(t) \) is a Markov process it does have a conditional probability density \( T(x,t|x_0,t_0) \) and accordingly we can write

\[
\frac{d}{dt} < f(X(t)) > = \int dx f(x) \frac{\partial}{\partial t} T(x,t|x_0,t_0)
\]

\[
= \int dx \{a(X(t),t)f'(X(t)) + \frac{1}{2}f''(X(t))b(X(t),t)^2\}T(x,t|x_0,t_0).
\]

The above equation can now be integrated by parts. Disregarding surface terms we obtain

\[
\int dx f(x) \frac{\partial}{\partial t} T = \int dx f(x) \left[ - \frac{\partial}{\partial x} [a(x,t)T(x,t)] + \frac{1}{2} \frac{\partial^2}{\partial x^2} [b(x,t)^2 T(x,t)] \right].
\]

Since, by construction \( f \) is an arbitrary function of \( x \) we can conclude that

\[
\frac{\partial}{\partial t} T(x,t|x_0,t_0) = - \frac{\partial}{\partial x} [a(x,t)T(x,t|x_0,t_0)] + \frac{1}{2} \frac{\partial^2}{\partial x^2} [b(x,t)^2 T(x,t|x_0,t_0)].
\]

We immediately recognize that the above equation is a Fokker–Planck equation. Hence we have shown the equivalence of a diffusion process defined in terms of a stochastic differential equation with drift coefficient \( a(x(t),t) \) and a diffusion coefficient \( b(X(t),t)^2 \) and the above Fokker–Planck equation.
7.4 The Stratonovich Stochastic Differential Equation

In subsection 7.2.2 we have seen that it is possible to give other definitions of the stochastic integral. One such definition is the Stratonovich stochastic integral defined in Eq. (7.7). It is clear that it is then possible to define stochastic differential equations using the Stratonovich integral, i.e.,

\[
X(t) = X(t_0) + \int_{t_0}^{t} ds \alpha(x(s), s) + S \int_{t_0}^{t} dW(s) \beta(x(s), s),
\]

(7.12)

In the mathematical literature it is customary to write the Stratonovich integral in the form

\[
\int_{t_0}^{t} \beta(x(s), s) \circ dW(s) \equiv S \int_{t_0}^{t} \beta(x(s), s) dW(s),
\]

where the notation \( \circ \) is called the Ito circle. From here on we will also stick to this notation. It is the aim of this subsection to show that stochastic differential equations defined in terms of the Stratonovich integral are equivalent to some appropriate Ito stochastic differential equations.

To this end we assume that the above \( x(t) \) is also a solution of the Ito stochastic differential equation

\[
dx(t) = a(x(t), t) dt + b(x(t), t) dW(t)
\]

(7.13)

and try to derive expressions for the corresponding \( \alpha \) and \( \beta \) in Eq. (7.12).

We begin by establishing the relation between the Ito and the Stratonovich integral. By definition of the Stratonovich integral we have

\[
\int_{t_0}^{t} \beta(x(s), s) \circ dW(s) \approx \sum_{i} \beta \left( x(t_i) + \frac{x(t_{i-1})}{2}, t_{i-1} \right) [W(t_i) - W(t_{i-1})],
\]

(7.14)

Using

\[
x(t_i) = x(t_{i-1}) + dx(t_{i-1})
\]

the argument of the \( \beta \) function can be written as

\[
\beta \left( x(t_i) + \frac{x(t_{i-1})}{2}, t_{i-1} \right) = \beta \left( x(t_{i-1}) + \frac{1}{2} dx(t_{i-1}), t_{i-1} \right).
\]

Then, with the help of the Ito stochastic differential equation (7.13) in the form

\[
dx(t) = a(x(t_{i-1}), t_{i-1})(t_i - t_{i-1}) + b(x(t_{i-1}), t_{i-1})(W(t_i) - W(t_{i-1}))
\]

and of Ito’s formula we get

\[
\beta \left( x(t_i) + \frac{x(t_{i-1})}{2}, t_{i-1} \right) = \beta(t_{i-1}) + \left[ a(t_{i-1}) \frac{\partial}{\partial x} \beta(t_{i-1}) + \frac{1}{4} b^2(t_{i-1}) \right] \frac{1}{2} (t_i - t_{i-1})
\]

\[
+ \frac{1}{2} b(t_{i-1}) \frac{\partial}{\partial x} \beta(t_{i-1}) [W(t_i) - W(t_{i-1})].
\]

The above expression can now be inserted back into the Eq. (7.14). Exploiting the fact that Ito calculus allows us to set \( dW^2 = dt \) and to drop the terms \( dt^2 \) and \( dt dW \) we find

\[
\int_{t_0}^{t} \beta(x(s), s) \circ dW(s) \approx \sum_{i} \beta(x(t_{i-1}), t_{i-1}) [W(t_i) - W(t_{i-1})]
\]

\[
+ \frac{1}{2} \sum_{i} b(x(t_{i-1}), t_{i-1}) \frac{\partial}{\partial x} \beta(x(t_{i-1}), t_{i-1}) (t_i - t_{i-1}),
\]
Since the first term on the right side of the above equation is a partial sum of an Ito integral we conclude from the above discrete formula that

\[ \int_{t_0}^{t} \beta(x(s),s) \circ dW(s) = \int \beta(x(t'),t')dW(t') + \frac{1}{2} \int b(x(t'),t') \frac{\partial}{\partial x} \beta(x(t'),t')dt'. \]  \hspace{1cm} (7.15)\]

The above formula gives us the relation between the Ito and the Stratonovich integral of a function \( \beta(x(t),t) \) in which \( x(t) \) is the solution of the Ito stochastic differential equation (7.13). The relation between the Ito and the Stratonovich form of stochastic differential equations can be seen by setting

\[
\begin{align*}
\alpha(x(t),t) &= a(x(t),t) - \frac{1}{2} b(x(t),t) \frac{\partial}{\partial x} b(x(t),t), \\
\beta(x(t),t) &= b(x(t),t).
\end{align*}
\]

We then get the following important equivalence.

The Ito stochastic differential equation

\[ dx = adt + b dW(t) \]  \hspace{1cm} (7.16)\]
is equivalent to the Stratonovich stochastic differential equation

\[ dx = [a - \frac{1}{2} b \frac{\partial}{\partial x} b] dt + b \circ dW(t). \]  \hspace{1cm} (7.17)\]

Conversely, the Stratonovich stochastic differential equation

\[ dx = \alpha dt + \beta \circ dW(t) \]
is equivalent to the Ito stochastic differential equation

\[ dx = [\alpha + \frac{1}{2} \beta \frac{\partial}{\partial x} \beta] dt + \beta dW(t), \]

### 7.4.1 Ito or Stratonovich?

We have just seen that a given stochastic differential equation can be interpreted in two ways: in the sense of Ito and in the sense of Stratonovich. Each of the two interpretations can be converted equivalently in the other version of the stochastic differential equation. Thus, the question arises: When modeling a physical system which interpretation should we use?

At the basis of the problem is the fact that in the more physical Langevin equations we are confronted with a \( \delta \) correlated white noise process. Hence, the proper mathematical analysis of stochastic differential equations was based on the mathematically safe Wiener process and we were led automatically to the ambiguities of defining Riemann sums for stochastic integrals.

All these difficulties can be circumvented by the following reasoning. Real processes in nature do have finite correlation times. Their spectrum might be flat, but not up to infinite frequencies. Such a noise term is called colored noise and could have zero mean and the following correlation function

\[ <\eta(t)\eta(t+\tau)> = \frac{\sigma^2}{2m} \exp(-m|\tau|). \]

The corresponding colored noise Langevin equation would read

\[ \dot{X}(t) = a(X(t),t) + b(X(t),t)\eta(t). \]  \hspace{1cm} (7.18)\]

For such a colored noise process the Riemann sums do converge and no ambiguity exists in choosing an interpretation.

In other words the ambiguities vanish by performing first the integration of the above colored noise Langevin equation and then perform the white noise limit \( \sigma \rightarrow \sigma m \) and \( m \rightarrow \infty \). Proceeding in this way we automatically get the Stratonovich interpretation of the stochastic differential equation. This is the content of the Wong-Zakai theorem (Hortshemke und Lefever [1984]). A good discussion of the Ito–Stratonovich dilemma can be found in van Kampen [1992].
The numerical integration of stochastic differential equations: The Euler–Maruyama method

Let us begin this section by review some basic facts of numerical methods for the simulation of deterministic ordinary differential equations (Garcia [1994]; Press et al. [1992]). To this end we consider the initial value problem

\[
\frac{dx}{dt} = a(t,x), \\
x(t_0) = x_0. 
\]

The most widely used numerical algorithms for the solution of the above problem are techniques. The simplest such method is the Euler method. The basic idea of the Euler method is to approximate the derivative on the right hand of the differential equation by the first order approximation

\[
\frac{dx}{dt} = \frac{x(t + \Delta t) - x(t)}{\Delta t} + O(\Delta t). 
\]

An approximate solution of the initial value problem can then be constructed by iterating the following recursion relation

\[
x(t + \Delta t) = x(t) + a(t,x)\Delta t. 
\]

Alternatively, introducing the time discretization \( t_0 < t_1 < \cdots < t_n \) with equal increments \( \Delta t \) the Euler algorithm can be formulated as

\[
x_{n+1} = x_n + a(t_n,x_n)\Delta t, 
\]

where it is intended that \( x_n \equiv x(t_n) \). Once, the initial value \( x_0 \) has been specified the approximation \( x_1,x_2,\ldots,x_n \) can be determined by applying Eq. (7.19) recursively.

Let us now turn our attention to the easiest finite difference method for the integration of stochastic differential equations Honerkamp [1990]; Kloeden et al. [1994]; Öttinger [1996]. Essentially, we have already met the easiest method for the numerical integration of stochastic differential equation while motivating them at the beginning of this chapter. Assume that we are interested in the solution of the following initial value problem for an Ito stochastic differential equation

\[
dX(t) = a(X(t),t)dt + b(X(t),t)dW(t), \\
X(t_0) = X_0, 
\]

where \( X_0 \) is the initial condition at time \( t_0 \). The simplest discretization scheme for the above differential equation is the Euler scheme, which in the context of stochastic differential equations is sometimes called the Euler–Maruyama method. For a given partition \( t_0 < t_1 < \cdots < t_{n-1} < t_n = t_{end} \) the Euler scheme is given by

\[
\tilde{X}_n = \tilde{X}_{n-1} + a(\tilde{X}_{n-1},t_{n-1})\Delta t + b(\tilde{X}_{n-1},t_{n-1})\Delta W_n, \\
\tilde{X}_0 = X_0, 
\]

where \( \Delta t_n = t_n - t_{n-1} \) and \( \Delta W_n \) is the Wiener increment

\[
\Delta W_n = W(t_n) - W(t_{n-1}). 
\]

Usually, we have \( \Delta t_n = \Delta t = \text{const} \), so that we can generate the Wiener increment with the help of the formula

\[
\Delta W_n = \xi \sqrt{\Delta t}, 
\]

where \( \xi \) is a gaussian distributed random variable with mean zero and unit variance. The random variable \( \tilde{X}_n \) generated by this iterative scheme is expected to approximate the stochastic process \( X(t) \). Sometimes the
above scheme is also termed the stochastic difference equation associated with the corresponding stochastic differential equation.

In order to characterize the quality of the approximation schemes for stochastic differential equations we have to introduce the concept of strong convergence. We say that a discrete approximation $\hat{X}$ with maximum time step $\Delta t$ converges strongly to $X$ at time $t_{\text{end}}$ if

$$\lim_{\Delta t \to 0} |X(t_{\text{end}}) - \hat{X}(t_{\text{end}})| = 0.$$ 

The order of convergence $v$ determines the numerical efficiency of a given numerical approximation scheme. If there exists a positive constant $c$, which is independent of $\Delta t$ such that for sufficiently small $\Delta t$ we have

$$|X(t_{\text{end}}) - \hat{X}(t_{\text{end}})|^{1/2} \leq c(\Delta t)^v,$$

then we say that the approximation scheme converges strongly with order $v$. The above criterion is simply the generalization of the usual deterministic convergence criterion and reduces to it when the diffusion coefficient vanishes and the initial condition is deterministic (Kloeden und Platen [1992]). The concept of a strongly convergent scheme is relevant for the following reason: A strongly convergent scheme gives approximations to the individual trajectories of the stochastic process. This is very important when the simulation is expected to resolve characteristic features of the trajectories of a stochastic process.

It can be shown that the Euler scheme has strong order of convergence $v = 0.5$. Thus the order of strong convergence of the stochastic Euler scheme is quite poor.

Fortunately, quite often one is not interested in constructing the individual realizations of the stochastic process but only in some averaged quantities, e.g., in some moments of the stochastic process. This is always the case, if the stochastic differential equation is regarded as an efficient numerical tool for the solution of a given Fokker–Planck equation. Namely, the latter contains only information about the moments of the stochastic process and not about the trajectories themselves. In these cases one is interested in the weak solutions of stochastic differential equations. An approximation scheme is said to converge weakly with order $v$ at time $t_{\text{end}}$ if for sufficiently smooth functions $g$ there exists a positive constant $c$, which does not depend on $\Delta t$ such that for sufficiently small $\Delta t$ we have

$$|<g(X(t_{\text{end}}))> - <g(\hat{X}(t_{\text{end}}))>| \leq c(\Delta t)^v.$$ 

Under suitable smoothness conditions for the functions $g$ the Euler scheme can be shown to have order of weak convergence $v = 1$.

### 7.5.1 The Ornstein-Uhlenbeck Process

As a first example of the application of the Euler algorithm we consider again the Ornstein–Uhlenbeck process. The implementation of the stochastic Euler algorithm has been realized in the program SDE.java with the help of the OrnsteinUhlenbeck.java method in the simulation.SDE package.
7.5. THE EULER–MARUYAMA METHOD

```java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.Random;

import simulation.*;
import simulation.SDE.*;
import VisualNumerics.math.*;
import ptolemy.plot.*;

/**
 * Stochastic Differential Equations
 *
 * A GUI for simulating SDEs of different kinds.
 */

public class SDE extends Applet
   implements ActionListener, Runnable {

   private void output() {
      double std, yLow, yHigh;

      // Output
      realPlot.setTitle("One\_Realization");
      realPlot.setXLabel("time\_t");
      realPlot.setYLabel("x(t)");
      meanPlot.setTitle("Mean\_<x(t)>");
      meanPlot.setXLabel("time\_t");
      meanPlot.setYLabel("<x(t)>");
      boolean connect=false;
      for (int k=0; k<nstep; k++) {
         // compute standard deviation of mean
         std=Math.sqrt((xstd[k]-Math.pow(xpos[k],2))/nReal);
         yLow=xpos[k]-std;
         yHigh=xpos[k]+std;
         meanPlot.addPointWithErrorBars(color, deltaT * k, xpos[k],
                                        yLow, yHigh, connect);
         realPlot.addPoint(color, deltaT * k, xreal[k], connect);
         if (connect==false) connect=true;
      }
      realPlot.repaint();
      meanPlot.repaint();
      color++;
      if (color>10) {
         color =0;
      }
   }

   // The three possible parameters for the different SDEs
   private static double param1 = 1;
   private static double param2 = 1;
   private static double param3 = 1.0;
   private static double param4 = 1.0;
   private static SDEfunction process = new OrnsteinUhlenbeck();

   private static double xstart = 1;
   private static double tend = 5;
```
// public int istep = 4;
private static double deltat = 0.1;
private static int nReal = 1000;
private static double tstart = 0;

private static double t, x, normal, sigma, muconst;
private static int nstep = (int) (tend / deltat);
private static double[] xpos = new double[nstep];
private static double[] xstd = new double[nstep];
private static double[] xfour = new double[nstep];
private static double[] xreal = new double[nstep];
private static Random rand;

private void compute() {
    double dW;
    double[] terms = new double[2];

    // for (int is = 1; is <= istep; is++) {
    // nstep = tend/deltat[is];
    // }

    border4.setCursor(new Cursor(Cursor.WAIT_CURSOR));

    for (int i = 1; i < nstep; i++) {
        xpos[i] = 0;
        xstd[i] = 0;
    }

    xpos[0] = xstart;
    xstd[0] = 0;
    xreal[0] = xstart;
    xfour[0] = xstart;

    // realization loop
    rand = new Random();

    for (int j = 0; j < nReal; j++) {
        t = tstart;
        x = xstart;
        if (threadActive == false) return;

        // time loop
        for (int i = 1; i < nstep; i++) {
            t += deltat;
            /* get the terms:
               drift     : a(x,t) = terms[0]
               diffusion : b(x,t) = terms[1] */
            terms = process.SDEterms(x, t);
            dW = Math.sqrt(deltat) * rand.nextGaussian();
            x += (terms[0] * deltat + terms[1] * dW);

            // store values in arrays for statistics
            xpos[i] += x;
            xstd[i] += x * x;
            xfour[i] = x * x * x * x;
            if (j == 0) {
                xreal[i] = x;
            }
        }
    }

    // end realizations

    for (int i = 1; i < nstep; i++) {
        xpos[i] /= nReal;
        xstd[i] /= nReal;
        xfour[i] = nReal;
    }
}
7.5. THE EULER–MARUYAMA METHOD

```java
private static Plot meanPlot, realPlot;
private static TextField textFieldParam1, textFieldParam2, textFieldReal, textFieldDT, textFieldTend, textFieldXstart, textFieldParam3, textFieldParam4;
private Choice choiceProcess;
private Button gobutton = new Button("go");
private Button stopbutton = new Button("stop");
private Button clearbutton1 = new Button("clear_left_plot");
private Button clearbutton2 = new Button("clear_right_plot");
private Button buttonPrintLeft = new Button("print_left");
private Button buttonPrintRight = new Button("print_right");
private Panel grid13 = new Panel(new GridLayout(1,4,5,5));
private Label tendLabel = new Label("t_end", Label.LEFT);
private Label xstartLabel = new Label("x_start", Label.LEFT);
private Label dTLabel = new Label("delta=t", Label.LEFT);
private Label labelParam1 = new Label("Param1", Label.LEFT);
private Label labelParam2 = new Label("Param2", Label.LEFT);
private Label labelParam3 = new Label("Param3", Label.LEFT);
private Label labelParam4 = new Label("Param4", Label.LEFT);
private Label labelReal = new Label("N_Real", Label.LEFT);
private Panel gridSouth = new Panel(new GridLayout(2,1,5,5));
private Panel flow7 = new Panel(new FlowLayout(FlowLayout.LEFT,5,5));
private Panel flow8 = new Panel(new FlowLayout(FlowLayout.LEFT,5,5));
private Panel border4 = new Panel(new BorderLayout(5,5));
private Label labelProcess = new Label("Which SDE?", Label.LEFT);
// private static ProgressBar progressBar = new Progressbar(0,100);

public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == clearbutton1) {
        realPlot.clear(true);
        realPlot.repaint();
    }
    if (evt.getSource() == clearbutton2) {
        meanPlot.clear(true);
        meanPlot.repaint();
    }
    if (evt.getSource() == stopbutton) {
        if (threadActive == true) {
            threadActive = false;
            border4.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
            // progress.setValue(0); // progress bar
        }
    }
    if (application == true) {
        if (evt.getSource() == buttonPrintLeft) {
            PrintComponent.Dialog(frame, realPlot, "Print_Left_Plot");
        } else if (evt.getSource() == buttonPrintRight) {
            PrintComponent.Dialog(frame, meanPlot, "Print_Right_Plot");
        }
    }
    else {
        if (evt.getSource() == gobutton) {
            if (threadActive == false) {
                // dereference the old thread to stop it completely!
            }
        }
    }
}
```
calcThread = new Thread(prg);
param1 = Double.valueOf(textFieldParam1.getText()).doubleValue();
param2 = Double.valueOf(textFieldParam2.getText()).doubleValue();
param3 = Double.valueOf(textFieldParam3.getText()).doubleValue();
param4 = Double.valueOf(textFieldParam4.getText()).doubleValue();
nReal = Integer.valueOf(textFieldReal.getText()).intValue();
deltat = Double.valueOf(textFieldDT.getText()).doubleValue();
tend = Double.valueOf(textFieldTend.getText()).doubleValue();
xstart = Double.valueOf(textFieldXstart.getText()).doubleValue();
nstep = (int)(tend / deltat);
xpos = new double[nstep];
xstd = new double[nstep];
xfour = new double[nstep];
xreal = new double[nstep];
// create the desired process
String dummyString = choiceProcess.getSelectedItem();
if (dummyString.equals("MeanRevertingRandomWalk"))
    process = new MeanRevertingRW(param1, param2, param3);
else if (dummyString.equals("OrnsteinUhlenbeckProcess"))
    process = new OrnsteinUhlenbeck(param1, param2);
else if (dummyString.equals("LognormalRandomWalk"))
    process = new LognormalRW(param1, param2);
else if (dummyString.equals("NoiseInducedTransition"))
    process = new NoiseInducedTransition(param1);
else if (dummyString.equals("StochasticResonance"))
    process = new StochasticResonance
        (param1, param2, param3, param4);
threadActive = true;
calcThread.start();
}

public static Thread calcThread, current;
private static boolean threadActive = false; // control thread execution

public static int color = 0;

public static SDE prg;
public static Frame frame;
public static boolean application = false;

public SDE() {
    this.setLayout(new BorderLayout());

    meanPlot = new ptoley.plot.Plot();
    realPlot = new ptoley.plot.Plot();
    choiceProcess = new Choice();

    meanPlot.setFont("Serif-bold-16");
    meanPlot.setTitleFont("Serif-bold-24");

    realPlot.setFont("Serif-bold-16");
    realPlot.setTitleFont("Serif-bold-24");

    textFieldTend = new TextField(new Double(tend).toString(), 5);
    textFieldXstart = new TextField(new Double(xstart).toString(), 5);
    textFieldDT = new TextField(new Double(deltat).toString(), 6);
7.5. THE EULER–MARUYAMA METHOD

textFieldReal = new TextField ( new Integer ( nReal ). toString (), 8 );
textFieldParam1 = new TextField ( new Double ( param1 ). toString (), 5 );
textFieldParam2 = new TextField ( new Double ( param2 ). toString (), 5 );
textFieldParam3 = new TextField ( new Double ( param3 ). toString (), 5 );
textFieldParam4 = new TextField ( new Double ( param4 ). toString (), 5 );

grid13 . add ( gobutton );
grid13 . add ( stopbutton );
grid13 . add ( clearbutton1 );
grid13 . add ( clearbutton2 );
if ( application == true ) {
    grid13 . add ( buttonPrintLeft );
    grid13 . add ( buttonPrintRight );
}

// Action Listener
stopbutton . addActionListener ( this );
gobutton . addActionListener ( this );
clearbutton1 . addActionListener ( this );
clearbutton2 . addActionListener ( this );
buttonPrintLeft . addActionListener ( this );
buttonPrintRight . addActionListener ( this );

textFieldTend . setBackground ( SystemColor . window );
textFieldTend . setForeground ( SystemColor . windowText );
textFieldDT . setBackground ( SystemColor . window );
textFieldDT . setForeground ( SystemColor . windowText );
textFieldParam1 . setBackground ( SystemColor . window );
textFieldParam1 . setForeground ( SystemColor . windowText );
textFieldParam2 . setBackground ( SystemColor . window );
textFieldParam2 . setForeground ( SystemColor . windowText );
textFieldReal . setBackground ( SystemColor . window );
textFieldReal . setForeground ( SystemColor . windowText );
gridSouth . add ( flow7 );
gridSouth . add ( flow8 );
flow7 . add ( labelParam1 );
flow7 . add ( textFieldParam1 );
flow7 . add ( labelParam2 );
flow7 . add ( textFieldParam2 );
flow7 . add ( labelParam3 );
flow7 . add ( textFieldParam3 );
flow7 . add ( labelParam4 );
flow7 . add ( textFieldParam4 );
choiceProcess . addItem ( "Ornstein–Uhlenbeck Process" );
choiceProcess . addItem ( "Lognormal Random Walk" );
choiceProcess . addItem ( "Mean Reverting Random Walk" );
choiceProcess . addItem ( "Noise Induced Transition" );
choiceProcess . addItem ( "Stochastic Resonance" );
flow7 . add ( labelProcess );
flow7 . add ( choiceProcess );
flow8 . add ( labelReal );
flow8 . add ( textFieldReal );
flow8 . add ( dtlabel );
flow8 . add ( textFieldDT );
flow8 . add ( tendlabel );
flow8 . add ( textFieldTend );
flow8 . add ( xstartlabel );
flow8 . add ( textFieldXstart );
realPlot . setButtons ( true );
realPlot . setCursor ( new Cursor ( Cursor . HAND_CURSOR ) );
meanPlot.setButtons(true);
meanPlot.setCursor(new Cursor(Cursor.HAND_CURSOR));
border4.add("North", grid13);
border4.add("South", gridSouth);
Panel grid = new Panel(new GridLayout(1,2));
grid.add(realPlot);
grid.add(meanPlot);
border4.add("Center", grid);
add("Center", border4);
}

public static void main(String[] args) {
    application = true;
    prg = new SDE();
    frame = new Frame("Stochastic Differential Equations");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    frame.add(BorderLayout.CENTER, prg);
    frame.pack();
    frame.setVisible(true);
    prg.init(); // start applet
    prg.start(); // start Thread
}

public void init() {
    if (prg == null) {
        prg = new SDE();
    }
    current = Thread.currentThread();
calcThread = new Thread(prg);
calcThread.setPriority(current.getPriority() - 1);
threadActive = true;
calcThread.start(); // calls the run() method !!
}

public void start() {
    border4.setCursor(new Cursor(Cursor.WAIT_CURSOR));
    threadActive = true;
}

public void stop() {
    border4.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    threadActive = false;
}

public void run() {
    if (threadActive == false) return;
    // progress.setValue(0); // progress bar
    compute();
    if (threadActive == false) return;
    output();
    threadActive = false;
}

} // SDE

package simulation.SDE;
The algorithm is very similar to the algorithm for the generation of exact trajectories of the Ornstein–Uhlenbeck process. Instead of looking at the realizations we estimate the expectation value for $<X^2>$ at the fixed final time $t_{\text{end}}=4$ ($t_{\text{start}}=0$). Choosing $q=1$ and $D=1$ the exact value of $<X^2(t=4)>$ is
Table 7.2 Results of the simulation of the Ornstein–Uhlenbeck process with the stochastic Euler method for different values of the time step. The parameters of the Ornstein-Uhlenbeck process are $q=1, D=1$. The simulation was run from $t_{\text{start}}=0$ to $t_{\text{end}}=4$ for 50000 realizations. The timesteps used are $\Delta t=0.2, 0.1, 0.05, 0.025$.

<table>
<thead>
<tr>
<th>$\Delta t$</th>
<th>$\langle X^2 \rangle$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.557979</td>
<td>0.00354552</td>
</tr>
<tr>
<td>0.1</td>
<td>0.523506</td>
<td>0.00331771</td>
</tr>
<tr>
<td>0.05</td>
<td>0.511391</td>
<td>0.00322181</td>
</tr>
<tr>
<td>0.025</td>
<td>0.506859</td>
<td>0.00320129</td>
</tr>
</tbody>
</table>

Figure 7.1: Results of the simulation of the Ornstein–Uhlenbeck process with the stochastic Euler method for different values of the time step. The parameters of the Ornstein-Uhlenbeck process are $q=1, D=1$. The simulation was run from $t_{\text{start}}=0$ to $t_{\text{end}}=4$ for 50000 realizations. The timesteps used are $\Delta t=0.2, 0.1, 0.05, 0.025$.

expected to be

$$\langle X^2(t=4) \rangle = \frac{1}{2}(1 - \exp(-8)) = 0.4998.$$ 

Since we know from the general discussion of the Euler algorithm that the expectation values converge to the exact result linearly with the time step we have included in the program a for loop over istep time steps $\Delta t$ to see explicitly this dependence. At the end of the simulation we perform a linear fit of the results with the help of the function polyfit in order to be able to extrapolate the results to $\Delta t = 0$.

The results of the simulation for 50,000 realizations and $\Delta t=0.2, 0.1, 0.05, 0.025$ are summarized in the following table. The same data have been plotted in Fig. (7.5.1). The figure clearly shows the expected linear convergence of the estimate to the expected exact result. The linear extrapolation leads to the estimate $\langle X^2 \rangle \vert_{\Delta t=0} = 0.497182$, which is in very good agreement with the expected exact result.

7.5.2 Noise Induced Transitions

In this second example of the application of the stochastic Euler method we want to simulate a stochastic differential equation with multiplicative noise and consider noise induced transitions.
Let us begin by looking at the following deterministic dynamical system

\[ \frac{d}{dt} x(t) = \frac{1}{2} - x(t), \]

for \( x \in [0, 1] \). Obviously, this dynamical system has one fix point at \( x_0 = 1/2 \). This fix point can be shown to be asymptotically stable.

We now want to perturb this system by adding a multiplicative noise term on the right hand of the above equation of motion. To be precise we want to replace the deterministic equation of motion by the Itô stochastic differential equation

\[ dX(t) = \left( \frac{1}{2} - X(t) \right) X(t) dt + \epsilon X(t)(1 - X(t)) dW(t), \tag{7.20} \]

where \( \epsilon < 0 \).

In order to understand the results of the simulation we want to look at the stationary solution of the corresponding Fokker–Planck equation. The latter reads

\[ \frac{\partial}{\partial t} P(x,t) = -\frac{\partial}{\partial x} [a(X(t)) P(X,t)] + \frac{1}{2} \frac{\partial^2}{\partial x^2} [b(X(t)) P(X,t)], \]

where we have used

\[ a(X(t)) = \left( \frac{1}{2} - X(t) \right) X(t) \]

and

\[ b(X(t)) = [\epsilon X(t)(1 - X(t))]^2. \]

Of course, the stationary solution has to satisfy

\[ \lim_{t \to \infty} \frac{\partial}{\partial t} P(X,t) = 0 \]

and hence

\[ \frac{d}{dx} \left[ a(X(t)) P(X,t) \right] - \frac{1}{2} \frac{\partial^2}{\partial x^2} [b(X(t)) P(X,t)] = 0. \]

The above equation can be written as

\[ \frac{d}{dx} J(x) = 0, \tag{7.21} \]

with

\[ J(x) = a(X(t)) P(X,t) - \frac{1}{2} \frac{\partial}{\partial x} [b(X(t)) P(X,t)]. \]

In order to satisfy Eq. (7.21) \( J \) must be constant. If we assume that the stationary density \( P_S(x) \to 0 \) for \( |x| \to \infty \) then the constant in question must be zero and we can conclude that

\[ \frac{d}{dx} [b(x) P_S(x)] = 2a(x) P_S(x). \]

Dividing both sides by \( b(x) P_S(x) \) we get

\[ \frac{d}{dx} \left[ \frac{b(x) P_S(x)}{b(x) P_S(x)} \right] = 2a(x) \frac{b(x)}{b(x)}. \]
Integrating the above expression gives
\[
\ln[b(x)P_S(x)] = \int_c^x dx' \frac{2a(x')}{b(x')}
\]
or
\[
P_S(x) = \frac{N}{b(x)} \exp\{-\phi(x)\},
\]
where
\[
\phi(x) = -\int_c^x dx' \frac{2a(x')}{b(x')}
\]
The factor \(N\) is a normalization constant to be chosen such that
\[
\int_a^b P_S(x)dx = 1.
\]
Transposing the above sketched general theory to the stochastic differential equation of interest we get the stationary density, which is sometimes also called the invariant density,
\[
P_S(x) = \frac{N}{x(1-x)} \exp\left(-\frac{1}{e^2x(1-x)}\right)
\]
Now we are in the position to simulate the stochastic differential equation (7.20). This will be done with the help of the program \texttt{SDE.java} and the function \texttt{simulation.SDE.NoiseInducedTransition.java}.

```java
package simulation.SDE;

/*** NoiseInducedTransition.java

* A class implementing the SDEfunction interface for solving
* a stochastic differential equation (SDE).
*<p>
* You can access all parameters by using the standard bean
* method names. For example to get a parameter "test" you simply
* call the method getTest() of the object and to set a parameter
* "sigma" you call setSigma(value).
*<p>
* Here an example of a noise induced transition. The process is
* defined by a potential U(x) = x(1-x).
*<p>
* dX = X(0.5-X) dt + ( epsilon x(1-x))'2 dW
*<p>
*/
```
7.5. THE EULER–MARUYAMA METHOD

```java
/*
public class NoiseInducedTransition implements SDEfunction {

  private double epsilon;

  public NoiseInducedTransition () {
    this(1.0); }

  public NoiseInducedTransition (double param) {
    epsilon = param; }

  public double getEpsilon () {
    return this.epsilon; }

  public void setEpsilon (double param) {
    this.epsilon = param; }

  public double[] SDEterms (double x, double t) {
    double[] retVal = new double[2];
    double dummy = x * (1 - x);
    retVal[0] = dummy;
    retVal[1] = Math.pow(epsilon * dummy, 2);
    return retVal;
  }
}
*/ // NoiseInducedTransition
```

The program generates trajectories of the stochastic process with the help of the Euler algorithm. At
the end of the simulation we evaluate numerically the stationary distribution with the help of the plotting
functions in the simulation package.

In a first run we perform a simulation of 5000 trajectories for the following parameters
\(x_{\text{start}}=0.5\),
\(\epsilon=1\),
\(t_{\text{end}}=4\), and \(\Delta t=0.01\). The initial condition was always chosen to be
\(x_{\text{start}}=0.5\).
The resulting histogram of the invariant density can be seen in Fig. (7.5.2).

It is clear from the histogram that the most probable value of \(X\) lies around 0.5 and is therefore identical
with the fixed point of the corresponding deterministic process.

Now we run the program with the same parameters as above but choose the multiplicative noise constant
to be \(\epsilon=3\). The result of this second simulation can be seen in Fig. (7.5.2). It is evident from this
figure that the invariant density changes its character. Now there is no longer one value of \(X\) which is more
probable. The histogram shows a minimum for \(X \approx 0.5\) and two equally high maxima at around 0.1 and
0.9. As a consequence of the larger noise the system undergoes a "stochastic bifurcation" which changes
the number of the maxima of the invariant density. Such a phenomenon is called a noise induced transition.

Let us now try to see whether this observation is in agreement with the stationary density we have
derived at the beginning of this subsection. The maxima of the stationary distribution are easily evaluated
from the equation

\[
0 = dP_S(x_m) \\
\]

which explicitly reads

\[
0 = (1 - 2x_m)[1 - \epsilon^2x_m(1 - x_m)].
\]

For \(0 < \epsilon < 2\) the invariant density has an extremum, namely a maximum at \(x_m = x_0 = 1/2\), which is the
fixed point of the deterministic equation of motion. For \(\epsilon > 2\) the invariant density posses a minimum at
\(x_0 = 1/2\) and two maxima of equal height at \(x_{m1,m2}\)

\[
x_{m1,m2} = \frac{1}{2} \left(1 \pm \sqrt{1 - (4/\epsilon^2)} \right).
\]
Figure 7.2: Histogram of the invariant density of the stochastic differential equation with multiplicative noise. The simulation was run from $t_{\text{start}}=0$ to $t_{\text{end}}=4$ for 5000 realizations. The initial condition was chosen to be $x_{\text{start}}=0.5$. The timestep used was $\Delta t=0.01$ and the multiplicative noise constant was $\epsilon=1$.

Figure 7.3: Histogram of the invariant density of the stochastic differential equation with multiplicative noise. The simulation was run from $t_{\text{start}}=0$ to $t_{\text{end}}=4$ for 5000 realizations. The initial condition was chosen to be $x_{\text{start}}=0.5$. The timestep used was $\Delta t=0.01$ and the multiplicative noise constant was $\epsilon=3$. 
Thus for the value of $\epsilon=3$ chosen in the second simulation the maxima are expected to be at 0.8727 and 0.1273 respectively. The histogram reproduced in Fig. (7.5.2) is in agreement with this theoretical prediction.

### 7.6 Stochastic Resonance

In order to get aquainted with the numerical integration of stochastic differential equations we discuss a phenomenon which occurs as the response of a nonlinear system in the presence of noise: stochastic resonance. For a comprehensive introduction see Lanzara et al. [1997]; Bulsara and Gammaitoni [1996]. We already know that the effect of noise on the time evolution of deteministic linear system is rather trivial. If the statistical properties of the input noise are known it is straightforward to compute the statistical properties of the output signal. For nonlinear systems the situation changes dramatically. The presence of the noise influences the evolution of the system often in a counterintuitive way. The numerical integration of the corresponding stochastic differential equations allows us to look at the realizations of the process and to gain insight in these interesting phenomena.

The phenomenon of stochastic resonance was proposed by Benzi et al. Benzi et al., [1982, 1981] in a series of papers in which they address the problem of the periodic switching of the Earth’s climate between periods of relative warmth and ice ages. It is known from the statistical analysis of continental ice volume series of papers in which they address the problem of the periodic switching of the Earth’s climate between one and the other of the two stable climate states. They name this phenomenon stochastic resonance for the following reason: the signal–to–noise ratio, i.e. the response of the system, is maximized when a parameter of the stochastic force is tuned to an optimal value.

It is the aim of this section to introduce the basic principles of stochastic resonance McNamara und Wiesenfeld [1989]; Gammaitoni et al. [1998] and to simulate a simple model showing this phenomenon. It is clear from the example discussed above that the basic mechanism of stochastic resonance relies upon three essential ingredients: a bistable system, a periodic driving signal, and a noise signal.

The simplest version of a one–dimensional nonlinear dynamical system is the damped anharmonic oscillator with the following equation of motion

$$m \frac{d^2x}{dt^2} + \gamma \frac{dx}{dt} = -\frac{dU(x)}{dx} + \sqrt{D}\chi(t).$$

(7.22)

The above Langevin equation describes the motion of a classical particle of mass $m$ in a potential $U(x)$ and with an additive stochastic force $\chi(t)$, where $\chi(t)$ is a Gaussian white noise characterized by

$$\langle \chi(t) \rangle = 0; \quad \langle \chi(t)\chi(t') \rangle = \delta(t-t').$$

(7.23)

The potential $U$ is bistable and we assume that it has the simple form (see figure 7.4)

$$U(x) = -\frac{a^2}{2} x^2 + bx^4.$$  

(7.24)

For $a > 0$ the potential $U$ is bistable with an unstable state at $x = 0$ and two stable states at $x_\pm = \pm \sqrt{a/b}$. The stable states are separated by a barrier of height $\Delta U = a^2/4b$. The system remains dynamically stable for $b > 0$, and becomes monostable for $a \leq 0$. Furthermore we assume that the system is overdamped by neglecting the inertial term $md^2x/dt^2$. Rescaling the resulting Langevin equation with the damping constant $\gamma$ we finally obtain the so–called stochastic Ginzburg–Landau equation

$$\frac{dx(t)}{dt} = ax - bx^3 + \sqrt{D}\chi(t),$$

(7.25)
Figure 7.4: The potential $U(x) = -a\frac{x^2}{2} + b\frac{x^4}{4}$ for $a = 1$ and $b = 1$. 
an equation which is often encountered in the theory of nonequilibrium critical phenomena.

7.6.1 Reaction Rate Theory

Let us now investigate some fundamental properties of the above dynamical system. We begin by considering the case $D = 0$ (no fluctuations). If the system is initially in a stable state it remains there for ever. The typical local time scale is the relaxation time, inside the wells. This time scale can be determined by considering small variations around the minimum of the well. To do so we linearize Eq. (7.25) around the stable state $x_s = \sqrt{a/b}$ with the help of

$$x(t) = x_s + \delta x(t)$$

and obtain

$$\frac{d(\delta x)}{dt} = -2a \delta x.$$  

(7.27)

Thus, the time scale $\tau_r$ of the relaxation inside the wells is

$$\tau_r = \frac{1}{2a} = \frac{1}{U''(x_s)}.$$  

(7.28)

The situation changes in the presence of noise. The fluctuating forces allow the system to jump between the two stable states. If the noise strength $D$ is small compared to the barrier height, these jumps are rare events. It is a well–know result of Kramers reaction–rate theory (see e.g. Hänggi et al. [1990]) that the mean escape time $T_e(x)$ out of one basin of attraction can be written for sufficiently low noise ($\Delta U/D \gg 1$) in the form of the Arrhenius law

$$T_e(x) \approx A \exp(\Delta U(x)/D),$$

(7.29)

where $A$ is a prefactor which depends on the form of the potential. For the special potential we consider here, performing a Gaussian approximation of the potential around the minimum and the maximum and using the condition $\Delta U/D \gg 1$ one gets the so–called Kramers formula (or Arrhenius formula) for the escape time of a bistable system

$$T_e \approx \frac{2\pi}{\sqrt{U''(0)U''(x_s)}} \exp[2\Delta U/D].$$

(7.30)

The escape time $T_e$ is the global time scale of the bistable system. The physical content of the assumptions we stated above is the following one: The Kramers time $T_e$ is derived under the assumption that the probability density within a well is roughly in equilibrium when the escape takes place. Thus, the condition $\Delta U/D \gg 1$ guarantees that the two time scales, the relaxation time and the Kramers time are different. The system relaxes quickly (on a short time scale) to a local equilibrium at the stable states and approaches global equilibrium (transitions over the barrier) on a slow time scale. It follows from Eq. (7.30) that

$$\frac{T_e}{\tau_r} = 2\sqrt{2\pi} \exp(2\Delta U/D) \gg 1.$$  

(7.31)

The rate to jump over the barrier $W_k$ is obviously the inverse escape time

$$W_k = \tau_e^{-1} = \frac{a}{\sqrt{2\pi}} \exp(-2\Delta U/D).$$

(7.32)

7.6.2 The Stochastic Resonance

Having reviewed some basic results of the theory of nonlinear stochastic systems we are now ready to look at the phenomenon of stochastic resonance. In the preceeding subsections we made use only of two of the basic ingredients of the recipe for stochastic resonance. The phenomenon only occurs in the presence of a
periodic driving signal. If we add such a signal to the bistable system just considered its dynamics will be governed by the following Langevin equation

$$\frac{dx}{dt} = -\frac{\partial U(x,t)}{\partial x} + \sqrt{D} \xi(t), \tag{7.33}$$

where the bistable potential takes now the form

$$U(x,t) = -\frac{x^2}{2} + b \frac{x^4}{4} - Ax \cos(\omega_s t), \tag{7.34}$$

where $A$ and $\omega_s = 2\pi/T_s$ are the amplitude and the frequency of the periodic signal. As a consequence of the periodic forcing term the potential tilts periodically between up and down. When the periodic force is at its maximum (or minimum) the difference between the escape rates from the two states is maximum. The periodic forcing is assumed to be so weak that it can not let the particle roll periodically from one potential well into the other one. Nevertheless, the noise–induced hopping between the two wells may become synchronized with the weak periodic forcing. This is signature of stochastic resonance. Stochastic resonance manifests itself by a synchronization of activated hopping between the potential minima with the weak periodic forcing.

The manifestation of the phenomenon may be visualized with the help of the output signal. It is clear that in the absence of the periodic driving the escape process is induced by the fluctuating force and is random. Thus, in this case the output signal $x(t)$ looks like dichotomous noise. The probability density of residence times between two jumps is exponential

$$P(t) = \frac{1}{\tau_k} \exp(-t/\tau_k), \tag{7.35}$$

where the Kramers time can be interpreted as the mean residence time spent by the system in one well. The periodic driving force alters this situation. The modulation synchronizes the hopping. The output signal reveals a quasiperiodic contribution to the jump process, which has a maximum if the system jumps, on average, two times per cycle of the external forcing. This is the stochastic resonance condition, which for small driving frequencies $\omega_s \ll \omega_k$ can be approximated by Jung [1993]

$$T_s = 2\tau_k = \frac{2\pi\sqrt{2}}{a} \exp(2\Delta U/D). \tag{7.36}$$

The same condition formulated in terms of the noise intensity reads

$$D_0 = \frac{2 \Delta U}{\ln(a/(\sqrt{2} \omega_s))}. \tag{7.37}$$

At this resonance condition the coherent contribution of the jump process has a maximum.

Now that we have learned the basic aspects of stochastic resonance let us write a program to simulate it. Before doing so it is very helpful to write the basic equations of motion in dimensionless form. To this end we write the bistable potential (7.34) in the form

$$U(x,t) = \Delta U \left[ -2 \left( \frac{x}{x_s} \right)^2 + \left( \frac{x}{x_s} \right)^4 \right] - U_1 \left( \frac{x}{x_s} \right) \cos(\omega_s t), \tag{7.38}$$

where $U_1 = Ax_s$. The Langevin equation of motion can then be written as

$$\frac{dx}{dt} = 4\Delta U \left( \frac{x}{x_s^2} - \frac{x^3}{x_s^4} \right) + \frac{U_1}{x_s} \cos(\omega_s t) + \sqrt{D} \xi(t). \tag{7.39}$$

If we devide the above equation through $4\Delta U$ and multiply it by $x_s$

$$\frac{1}{x_s} \frac{dx}{dt} = \left( \frac{x}{x_s} - \frac{x^3}{x_s^3} \right) + \frac{U_1}{4\Delta U} \cos(\omega_s t) + \frac{x_s}{4\Delta U} \sqrt{D} \xi(t). \tag{7.40}$$
If we now choose the unit of length to be $x_s$ and the time unit to be $1/a$, note that $x_s^2/4\Delta U = 1/a$, the dimensionless form of the above equation reads

$$dx = (x - x_s^3)dt + \frac{U_1}{4\Delta U} \cos(\omega_s t) dt + \sqrt{D}\eta(t)\sqrt{dt},$$

(7.41)

where for notational convenience we have named $x$ and $t$ to denote the corresponding dimensionless quantities and $D$ denotes the dimensionless noise intensity. Recall, that in the units defined above $D$ has the dimension $a x_s^2$.

Before running the program let us formulate the resonance condition in dimensionless units. It follows from Eq. (7.37) that

$$D_0 = \frac{1}{2 \ln(1/\sqrt{2}\omega_s)},$$

(7.42)

To give a numerical example, for $\omega_s = 0.1$ the time scale matching condition states that we have to choose $D = 0.2556$. Let us run now the program keeping first $\omega_s$ fixed and varying $D$ and then keeping $D$ fixed and varying $\omega_s$.

Here comes the simulation and the discussion.

Now we could discuss:

- the periodic response
- the signal to noise ratio

Let us finally look at the spectrum of the output signal, ...

Since its original formulation the phenomenon of stochastic resonance has been observed in several physical and biological systems. These observations are reviewed in Gammaitoni et al. [1998] and the introductory article Bulsara und Gammaitoni [1996].

### 7.7 Exercises

**Exercise 7.1 Johnson Noise [Gillespie, 1996]**

Johnson noise is the thermally generated electrical noise appearing in a conductor. Assume we have a rigid wire loop of self-inductance $L$ and resistance $R$ at absolute temperature $T$. We can visualize this using the figure below.

![Diagram of a Johnson noise circuit](image)

There is no external potential, just the interactions between the conducting electrons and the vibrations of the atomic lattice give rise to a temporally varying electromotive force in the loop (for details see [Gillespie, 1996]).

The circuit equation gives (the integral of the electric potential around the loop must be zero)

$$-RI(t) + V(t) - L\frac{dI(t)}{dt} = 0.$$

Taking averages gives (assume $\langle V(t) \rangle = 0$)

$$-L\frac{d\langle I(t) \rangle}{dt} = R\langle I(t) \rangle.$$
If we can measure \( I(t) \) we have an experimental way of determining \( R \) and \( L \).

We rewrite the first equation using \( \tau \) := \( \frac{L}{R} \) and \( V(t) := Lc^{1/2} \Gamma(t) \), where \( \Gamma(t) \) is a Gaussian white noise and \( c > 0 \) constant:

\[
\frac{dI(t)}{dt} = \frac{1}{\tau} I(t) + c^{1/2} \Gamma(t).
\]

This is just the Langevin-equation for an Ornstein-Uhlenbeck process with relaxation time \( \tau \) and diffusion constant \( c \). The diffusion constant \( c \) can be calculated by using the equipartition theorem of statistical mechanics and applying the results for the Ornstein-Uhlenbeck process. We get \( c = 2kT R/L^2 \).

Write a program to solve the Langevin equation for \( I(t) \) with the initial condition \( I(t) = i_0 \). Use the Euler method for solving stochastic differential equations. Use the three different sets of parameters:

- \( \tau = 1, c = 1, i_0 = 0, \Delta t = 0.001 \)
- \( \tau = 1, c = 1, i_0 = 0, \Delta t = 0.0001 \)
- \( \tau = 0.001, c = 1,000,000, i_0 = 0, \Delta t = 0.001 \)

(Remark: the first and the third parameter set lead to the same constant \( c^{1/2} \) for discussing \( \tau \to 0 \), the limit to Gaussian white noise.)

The exact solution is (\( N(\cdot) \) denotes a normal distribution):

\[
I(t) = N \left( \frac{i_0 e^{-t/t_0}}{L} \frac{kT}{L} \left( 1 - e^{-2t/t_0} \right) \right).
\]

Also calculate the (auto-)covariance function \( C_1(t') := \text{cov}(I(t), I(t + t')) \) (the subscript \( s \) indicates that the process should be stationary) and analyze the spectrum thereof - use a log-log plot for the spectrum. The spectrum is the cosine transform of \( C_1 \).
Bibliography


Part IV

Advanced Simulation Techniques
Chapter 8

Molecular Dynamics

8.1 Introduction

8.1.1 Statistical Properties of Fluids

With this chapter we begin the description of the application of stochastic methods to the simulation of physical systems. Computer simulations are an essential tool for the understanding of complex physical systems which complement the more traditional theoretical and experimental approaches. Computer simulations offer the possibility to investigate the structural, dynamical and thermodynamical properties of interesting systems. Within statistical physics essentially two different simulation approaches have been developed: Molecular Dynamics (MD) methods, which we will consider in this chapter and Monte Carlo methods (MC), which will be introduced in the next one.

The basic idea of the Molecular Dynamics approach is the following one: We know that fluids, e.g. gases and liquids, are systems which are composed of a large number of particles which are mutually interacting. Although the most fundamental description of matter at the microscopic level is, of course, quantum mechanical, in the framework of Molecular Dynamics it is assumed that we can describe the motion of a single atom or molecule by the classical laws of motion, i.e., by Hamilton equations of motion. This assumption is justified, if the mean de Broglie wavelength of an atom is much smaller then the mean distance between atoms. This condition is satisfied if the density of the fluid is sufficiently low or if the temperature of the fluid is sufficiently high. Thus, it is the aim of Molecular Dynamics to understand the macroscopic thermodynamic and dynamical properties of a fluid starting from the microscopic equations of motions of its atoms. To put it differently: Molecular Dynamics is the direct simulation of the equations of motion of a system composed of $N$ classical mutually interacting particles. The main numerical problem we have to address in this chapter is the development of an algorithm for the numerical integration of the equations of motion of $N$ particles.

The numerical solution of classical equations of motion looks like a deterministic process. Why is it then relevant for us? The answer is very easy. Molecular Dynamics is a beautiful example of a deterministic Markov process. The temporal evolution of the system is of course deterministic, but the initial condition, the initial configuration of the atoms in the fluid is random! The initial position as well as the initial velocities of the particles are random. A Molecular Dynamics simulation solves the Liouville equation, which as we know is the typical example of the differential Chapman–Kolmogorov equation of a deterministic Markov process.

8.1.2 Some Historical Comments

The Molecular Dynamics method has been proposed by Alder and Wainwright in 1957 Alder und Wainwright [1957, 1959]. They investigated a fluid composed of hard spheres. In this pioneering simulation the particles moved with constant velocity between two elastic collisions. The first simulation of a real fluid was published in 1964 by Rahman Rahman [1964]. He simulated liquid Argon assuming a Lennard-Jones potential for the pair interaction between the atoms (see below). The equations of motion were integrated.
using a stepwise algorithm. It is worth mentioning a citation classic in this context, namely the paper by Verlet [1967, 1968], in which the algorithm for the integration of the equations of motion, which bears his name has been proposed. Since this time Molecular Dynamics developed to a standard tool in statistical physics. We just want to mention some important steps. 1968 a diatomic molecular fluid was simulated for the first time ?. 1971 Rahman and Shillinger reported the simulation of liquid water. In the same year Woodcock investigated a fluid with Coulomb interactions - melted KCl. In 1973 Barojas, Levesque and Quentrec simulate rigid molecules. In 1977 Camman, Gelin and Karplus apply Molecular Dynamics to the investigation of proteins. The Molecular Dynamics simulation we mentioned were all concerned with equilibrium properties of fluids.

At the beginning of the seventies Non–Equilibrium Molecular Dynamics (NEMD) was developed, which allowed also the investigation of the behaviour of fluids in typical non-equilibrium situations, e.g., fluids in thermal gradients, or fluid in shear flow.

At the beginning of the eighties two important papers appeared which reported on the possibility to include also quantum mechanical effects in the MD simulations.

8.1.3 The Equations of Motion

The classical Molecular Dynamics algorithm solves the equation of motion for a set of \( N \) particles. A mechanical system with \( 3N \) degrees of freedom is fully characterised by a set of \( 3N \) generalised coordinates \( q_i, i = 1, \ldots, 3N \) and of \( 3N \) generalised momenta \( p_i, i = 1, \ldots, 3N \). For a given Hamiltonian \( H \) the equations of motion read

\[
\dot{q}_i = \frac{\partial H}{\partial p_i} \quad \text{for} \quad i = 1, \ldots, N
\]

and

\[
\dot{p}_i = -\frac{\partial H}{\partial q_i} \quad \text{for} \quad i = 1, \ldots, N.
\]

In many cases the potential is conservative and Cartesian coordinates \( x_i \) and the velocities \( v_i = p_i/m_i \) are used, where \( m_i \) denotes the mass of particle \( i \). In these cases the Hamiltonian may be written as

\[
H(p_i, x_i) = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + V(x_i).
\]

For this special case the equations of motion read

\[
\dot{x}_i = v_i \quad \text{and} \quad m_i \ddot{v}_i = F_i, \quad (8.1)
\]

where the forces are

\[
F_i = -\frac{\partial}{\partial x_i} V(x_i),
\]

It is clear from Eq. (8.1) that the calculation of the velocities \( v_i \) is not essential, since the Hamiltonian equations of motion (8.1) are equivalent to the Newtonian ones

\[
m_i \ddot{v}_i = F_i(x_i).
\]

Having recalled the equations of motion of classical system composed of \( N \) particles we have to address the two following questions which are essential to all Molecular Dynamics simulations: How do we calculate the forces? How do we integrate the equations of motion? These questions will be addressed in the following two sections.
8.2 Simple Models and Interaction Potentials

In the first Molecular Dynamics simulation the classical fluid of interest was liquid Argon. In liquid argon the atoms are chemically inert and can be assumed to be spherical. For this reason the forces between any two atoms depend only on the distance between the atoms. Thus, considering a system of $N$ atoms the potential energy may be written as

$$V = \sum_i V_i + \sum_{i,j} V_{ij} + \sum_{i,j,k} V_{ijk} + \cdots,$$

where the first sum on the right-hand side represents the potential of an external field, the second sum represents the pair interactions, the third sum represents the three particle interaction, and so on. Of course, the calculation of the three particle interaction and of interaction of higher order are very expensive from a computational point of view. Fortunately, in praxis these contributions to the potential can be neglected. For simple fluids, like Argon, the pairwise interaction $V_{ij}$ depends only on the magnitude of the separation $r_{ij}$ between the atoms $i$ and $j$. For the moment we will also assume that there is no external field acting on the particles, so that we are left only with the pair interaction potential $V_{ij}$.

This potential can, of course, be determined from first principles using quantum mechanics. These calculations turn out to be very difficult and so one relies upon phenomenological forms for the potential $V_{ij}$. For argon and other noble atoms with closed electronic shells, the mutual polarisation of each atom induces an attractive interaction, which is known as the van der Waals interaction. This interaction has the form

$$V_{vdW} \approx -I \left( \frac{\sigma}{r} \right)^6,$$

where $I$ is the ionisation potential of the atom, and $\sigma$ is a length scale characterising the size of the atom.

When two argon atoms approach each other so closely that their electron shells overlap, the Pauli exclusion principle causes an effective repulsive force. This force increases rapidly with decreasing separation between the atoms. This effect is called core repulsion. The most common phenomenological form of the potential $V_{ij}$ is the Lennard–Jones potential

$$V_{LJ} = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right].$$

A plot of the Lennard–Jones potential can be seen in Fig. (8.2). It is important to notice that $V_{LJ} = 0$ at $r = \sigma$ and that $V_{LJ}(r)$ is essentially equal zero for $r \geq 3\sigma$. The parameter $\varepsilon$ is the depth of the potential at the minimum. The minimum occurs at the separation $r = 2^{1/6}\sigma$. The parameters $\varepsilon = 1.65 \times 10^{-21} J$ and $\sigma = 3.4 \AA$ are in good agreement with the experimental properties of liquid argon.

In order to study some special properties of fluids it is sometimes useful to consider less general potentials. The hard–sphere potential is defined as

$$V_{hs}(r) = \begin{cases} \infty, & (r < \sigma) \\ 0, & (\sigma \leq r) \end{cases}.$$  

The hard sphere potential is depicted in Fig. (8.2).

The Lennard–Jones potential may be approximated by the square well potential

$$V_{sw}(r) = \begin{cases} \infty, & (r < \sigma_1) \\ -\varepsilon, & (\sigma_1 \leq r < \sigma_2) \\ 0, & (\sigma_2 \leq r) \end{cases}.$$
The square well potential is depicted in Fig. (8.2).

The soft sphere potential is defined as

$$ V_{ss}(r) = \varepsilon \left( \frac{\sigma}{r} \right)^v, $$

where $v$ is a parameter controlling the strength of the potential. For increasing $v$ the potential gets "harder". In Fig. (8.2) we plot the soft sphere potential for two values of $v$. We mentioned in the introduction to this chapter that Molecular Dynamics simulations may also be used to investigate fluids with Coulomb interaction. In this case the potential reads

$$ V_{zz}(r_{ij}) = \frac{z_i z_j}{4\pi e_0 r_{ij}}, $$

where $z_i$ and $z_j$ denote the charge of the ion $i$ and $j$, respectively.

### 8.3 Algorithms for the Integration of Newton’s Equations of Motion

In this section we want to get acquainted with algorithms for the numerical integration of Newton’s equations of motion. We will write them for notational convenience as

$$ \frac{d^2}{dt^2} x(t) = f(x), $$

where $x(t)$ and $f$ are vectors with $3N$ coordinates. Probably, you will already have heard a course on numerical methods for physics and will therefore suppose that the right algorithms are Runge–Kutta algorithms. As you may remember Runge–Kutta algorithms require the evaluation of the force several times per time step. Unfortunately, in Molecular Dynamics simulations the evaluation of the forces is the most time consuming step during the simulation. So, Runge–Kutta methods are not a good choice. We will therefore have to look at algorithms which require at most one or two evaluations of the forces per time step.

In the following we will consider finite difference methods in order to compute the values of $x_{n+1}$ and $v_{n+1}$ at time $t_{n+1} = t_n + h$, where $h$ denotes the time step of the integration. $\Delta t$ must be chosen in such a way that the integration methods generates a stable solution. For a conservative system $h$ must be chosen small enough to ensure that the total energy is conserved within the required accuracy.

#### 8.3.1 Euler Methods

The easiest but not particularly precise choice are Euler algorithms. We write Newton’s equation of motion in the following form

$$ \frac{d}{dt} x = v(t) $$

and

$$ \frac{d}{dt} v = f(t). $$

In order to derive an integration algorithm we look at $x_{n+1} = x(t_n + h)$ and $v_{n+1} = v(t_n + h)$ and expand in a Taylor series around $x(t_n)$ and $v(t_n)$, respectively. Keeping terms up to second order in $h$ we obtain the
Euler method

\[
x_{n+1} = x_n + hv_n + \frac{1}{2}h^2f_n \\
v_{n+1} = v_n + hf_n.
\]

The Euler algorithm can be improved by expanding it to a predictor–corrector scheme. The predictor step of the modified Euler algorithm reads

\[
y_{n+1} = x_n + hv_n + \frac{1}{2}h^2f_n \\
f^e = \frac{1}{2}(f_n + f(y_{n+1}))
\]

and the corresponding corrector step is

\[
x_{n+1} = x_n + hv_n + \frac{1}{2}hf_n \\
y_{n+1} = y_n + \frac{1}{4}(f(y_{n+1}) - f_n) \\
v_{n+1} = v_n + hf^e
\]

where \( f_{n+1} = f(y_{n+1}) \). This modified Euler algorithm is more accurate than the original version, but it implies a double evaluation of the forces. By choosing

\[
f_{n+1} = f(y_n)
\]

only one evaluation of the forces is necessary. This last predictor/corrector Euler algorithm turns out to be less accurate than the modified Euler algorithm.

### 8.3.2 The Gear Algorithm

Some Molecular Dynamics simulations employ the Gear algorithm. The Gear algorithm is a systematic predictor–corrector scheme. In the first predictor step the positions and the velocities are developed in a Taylor series. Then the accelerations, which in general are different from those predicted by the Taylor expansion, are calculated from the new positions. From the difference of the two accelerations one calculates in the second corrector step corrections to the positions and the velocities. Depending on the order of the Taylor expansion which are considered one obtains Gear algorithms of different order.

Here, we just want to present the algorithm and refer the interested reader to the original literature Gear [1971]. The algorithm is based on the definition of the vector

\[
\mathbf{x}_n = (x_n, h^2x'_n, \frac{1}{2}h^2x''_n, \frac{1}{3!}h^3x'''_n, \ldots).
\]

With the help of this vector the predictor step is formulated as

\[
\mathbf{y}_{n+1} = A\mathbf{x}_n,
\]

and the corrector step reads

\[
\mathbf{x}_{n+1} = \mathbf{y}_{n+1} + \alpha\frac{1}{2}h^2(\mathbf{f}(\mathbf{y}_{n+1}) - \mathbf{f}(\mathbf{y}_{n+1})).
\]

The coefficients of the matrix \( A \) and of the vector \( \alpha \) in the predictor and in the corrector step have to be chosen differently for every different order. To give an example, in the Gear algorithm of 4th order the matrix \( A \) is

\[
A = \begin{pmatrix}
1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 3 \\
0 & 0 & 0 & 3
\end{pmatrix}
\]
and the vector $\vec{a}$ is given by

$$\vec{a} = \begin{pmatrix} 1/6 \\ 5/6 \\ 1 \\ 1/3 \end{pmatrix}.$$ 

For the Gear algorithm of 5th order $A$ and $\vec{a}$ have to be chosen as

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 & 6 \\ 0 & 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$\vec{a} = \begin{pmatrix} 19/20 \\ 3/4 \\ 1/2 \\ 1/12 \end{pmatrix}.$$ 

### 8.3.3 Verlet and Beeman Algorithm

#### The Verlet Algorithm

One of the most popular algorithms for the integration of Newton’s equations of motion is the Verlet algorithm. The Verlet algorithm is less accurate than the Gear algorithm but it is simpler and requires much less memory. Let us consider the Taylor expansion of $x(t_n + h)$ and of $x(t_n - h)$

$$x_{n+1} = x_n + h\gamma_n + \frac{1}{2} h^2 \gamma_n' + \frac{1}{3!} h^3 \gamma_n'' + \cdots$$

$$x_{n-1} = x_n - h\gamma_n + \frac{1}{2} h^2 \gamma_n' - \frac{1}{3!} h^3 \gamma_n'' + \cdots$$

Adding the two above expansions we get

$$x_{n+1} = 2x_n - x_{n-1} + h^2 f(x_n).$$

This is the Verlet algorithm. Note that velocities do not appear in the above equation. If knowledge about the velocities is needed, i.e., for computing the kinetic energy, they can be computed as central differences

$$v_n = (x_{n+1} - x_{n-1})/2h.$$ (8.2)

Note, that the Verlet algorithm is not self–starting, since it requires the knowledge of $x_n$ and $x_{n-1}$. If initially only $x_n$ and $v_n$ are known, $x_{n+1}$ can be computed in the first step as

$$x_{n+1} = x_n + hv_n + \frac{1}{2} h^2 f(x_n).$$

The global errors associated with the Verlet algorithm are third order for the positions and second–order for the velocities.

#### The Velocity Form of the Verlet Algorithm

A mathematically equivalent version of the Verlet algorithm is the so called velocity Verlet algorithm, which is given by

$$x_{n+1} = x_n + v_n h + \frac{1}{2} f_n h^2.$$
and
\[ v_{n+1} = v_n + \frac{1}{2} (f_{n+1} + f_n) h. \]

The velocity Verlet algorithm is self-starting and minimises the errors. The equivalence of the two algorithms is best seen in the following way. Adding the expressions for \( x_{n+1} \) and for \( x_{n-1} \) and eliminating the velocities with the help of the corresponding expression we obtain immediately the Verlet algorithm from the velocity Verlet algorithm. In the following we will refer the velocity Verlet algorithm as the Verlet algorithm.

The Beeman Algorithm

Another well-known algorithm is the Beeman algorithm, which also avoids the round-off errors of the original Verlet algorithm. It can be written in the following form
\[ x_{n+1} = x_n + v_n h + \frac{1}{6} (4f_n - f_{n-1}) h^2 \]
and
\[ v_{n+1} = v_n + \frac{1}{6} (2a_{n+1} + 5f_n - f_{n-1}) h. \]

The equivalence of the Beeman algorithm with the Verlet algorithm can be demonstrated in the following way. It follows from (8.4) that
\[ h v_{n-1} = x_n - x_{n-1} - \frac{1}{6} h^2 (4f_{n-1} - f_{n-2}). \]
Inserting (8.5) into Eq. (8.4) for \( v_n \) we obtain
\[ v_n = v_n - \frac{1}{3} \frac{h^2}{f_n} + \frac{1}{6} \frac{h^2}{f_{n-1}}. \]

Finally, the original Verlet algorithm is recovered by inserting the above equation (8.6) into Eq. (8.3).

The velocity Verlet algorithm is probably the most popular integrator for MD applications. It is however important to remark that it is not necessary to use it. The only criterion for the quality of an algorithm is, for systems without thermalization, the conservation of energy. All the algorithms we presented have a stable mean value of the total energy. Of course, the actual value of the energy may fluctuate. As a matter of fact such deviations are less relevant in algorithms with a higher precision, however the latter may show a drift in the mean value of the energy.

8.3.4 The Comparison of the Algorithms

In order to compare the algorithms we consider the harmonic oscillator
\[ \frac{d^2}{dt^2} x = -x, \]
with initial condition \( x(0) = 1 \) and \( v(0) = 0 \). The exact solution to the above equation is of course
\[ x(t) = \cos t \quad v(t) = -\sin t. \]

Das gibt eine schoene Uebung!!!!!!!! Plot Vergleich error,...
8.4 The Algorithm for the Simulation

8.4.1 Periodic Boundary Conditions

We have seen in the previous section that a Molecular Dynamics code has to store at least 3N coordinates of the N molecules we are interested in. To this minimal requirements we have to add 3N velocities, N(N - 1)/2 pair energies, 3N forces and so on. It is clear that depending on the computer at our disposal the number N of particles can not be made arbitrarily large. Since the aim of a Molecular Dynamics simulation is to understand the properties of a bulk system, which is typically composed of \( N = 10^{23} \) to \( 10^{25} \) particles, it is clear that we can only simulate a fraction of this particles. Typically Molecular Dynamics simulation operate with \( 10^3 \) up to \( 10^5 \) particles. It is clear that it would not be clever to enclose these particle in a box. In contrast to bulk systems the fraction of particles near the walls would be very large, and surface effects would have a dominant role. For example, if we consider a cubic lattice of 512 particles, 296 particles will be on the surface of the cube. This amounts to 58% of the total number of the particles.

In order to minimise surface effects and simulate more closely the properties of a bulk system the following trick is used. One considers a cubic box. The length of the side of the box is L. The box contains N particles and it is assumed to be in the bulk of the fluid. The box is understood as an element of a lattice of identical cells, which also contain N particles, which are spatially distributed as in the original cell. A particle near the border of the cell is surrounded not only from the particles in the original box but also from the neighbouring particles in the surrounding boxes. If a particle leaves the box in the course of the simulation through one side of the cubic cell, an identical particle enters the box from the opposite side, so that the number of particles in the box remains constant. These kind of boundary conditions are called periodic boundary conditions they are illustrated in Fig. (8.5). Usually, only the coordinates of the original particles or of their images in the simulation box are saved.

**Figure 8.5:** Periodic boundary conditions.

In praxis we have to proceed as follows. Let \( x[i], y[i], z[i] \) be the coordinates of the particle with number i, and let L be the length of the side of the simulation box. After each change of the position of the particles during the MD simulation we have to perform the following operations

\[
\begin{align*}
\text{if (} x[i] > L \text{ ) } & x[i] = x[i] - L; \\
\text{if (} x[i] < 0 \text{ ) } & x[i] = x[i] + L; \\
\text{if (} y[i] > L \text{ ) } & y[i] = y[i] - L; \\
\text{if (} y[i] < 0 \text{ ) } & y[i] = y[i] + L; \\
\text{if (} z[i] > L \text{ ) } & z[i] = z[i] - L; \\
\text{if (} z[i] < 0 \text{ ) } & z[i] = z[i] + L;
\end{align*}
\]

8.4.2 Potential Cutoff

Because of the periodic boundary conditions we have to compute the forces in an infinite system. Of course, this is not feasible. Consider the Lennard–Jones potential. If the distance between two particles is 3\( \sigma \) the potential energy is about a half % of the minimal value. It would imply a waste in computational power to try to calculate this small rest. In praxis, for all neutral systems with Lennard–Jones like potential it is safe to calculate pair–interaction energies only up to the so–called cutoff radius, \( r_c \), and to set the potential energy and the forces for large distances equal to zero, i.e.,

\[
V(r) = \begin{cases} 
V(r), & \text{for } r < r_c \\
0, & \text{otherwise,}
\end{cases}
\]

For the Lennard–Jones potential the cutoff radius is usually chosen to be \( r_c \approx 2.5\sigma \).
8.4.3 The Minimum Image Convention

If the cutoff radius $r_c$ is smaller than half of the length of the simulation box $L$, the minimum image convention may be used. The interaction partners of particle $i$ are those $N-1$ particles (images) which lie within an imaginary box, which has the same size as the original simulation box, but is centred around the position of particle $i$. Through this convention the particles in the simulation box interact only with particles which are either themselves in the simulation box or which are shifted in each coordinate by a factor of $+L$ or $-L$ (see Fig. (8.6)).

Figure 8.6: The minimum image convention.

We consider the interaction between particle $i$ and particle $j$. The coordinate differences for the particles in the simulation box are denoted by $x_{ij}, y_{ij}$, and $z_{ij}$. These can be simply evaluated as

$$x_{ij} = x[i] - x[j];$$
$$y_{ij} = y[i] - y[j];$$
$$z_{ij} = z[i] - z[j];$$

The minimum image convention is realized through the following code

```java
if (x_{ij} > L/2) x_{ij} = x_{ij} - L;
if (y_{ij} < -L/2) x_{ij} = x_{ij} + L;
```

and similarly for $y_{ij}$ and $z_{ij}$. From the infinite number of images of particle $j$ we pick out that copy of it, whose distance from particle $i$ may be less than $r_c$. The minimum distance convention pair-distance is then computed as

$$r_{ij} = \text{Math.sqrt}(x_{ij}^2 + y_{ij}^2 + z_{ij}^2);$$

It is important to remark that in the minimum image convention each particle interacts exactly with $(N-1)$ particles.

8.4.4 Reduced Variables

In a MD simulation the equations of motion of the particles are integrated many times. It is therefore important to keep numerical roundoff errors as small as possible. A prerequisite for small roundoff errors is to choose units in such a way that the quantities which we want to compute are of the order of unity. The fluids we are concerned with here are Lennard–Jones fluids. So it is natural to assume that the units of distance and energy are Lennard–Jones parameters $\sigma$ and $\epsilon$. The units of mass is typically chosen to be the mass of one atom $m$. All other quantities can be expressed in terms of $\sigma$, $\epsilon$ and $m$. Accordingly, velocities are measured in units of $(\epsilon/m)^{1/2}$, and time is measured in units of $(\sigma m/\epsilon)^{1/2}$. All program variables are expressed in reduced units. Table (8.1) summarises the system of units used in typical molecular dynamics simulations and shows the corresponding values for argon.

As an example, let us consider a typical molecular program with a dimensionless time step of $\Delta t = 0.01$ which runs for 10000 steps. The total time of the run is $10000 \times 0.01 = 100$ in reduced units or $2.17 \times 10^{-10}$s for argon.

8.5 Advanced AWT Features and GUIs

Before going on, we want to discuss some advanced features of the Java AWT package, so that you are able to understand all the code of the upcoming simulations.
Table 8.1 The system of units used in molecular dynamics simulation of particles interacting via a Lennard–Jones potential. The numerical values for $\sigma$, $\epsilon$ and $m$ are for argon. The quantity $k$ is Boltzmann’s constant and has the value $k = 1.38 \times 10^{-23} J/K$. The unit of pressure is for a two–dimensional system.

<table>
<thead>
<tr>
<th>quantity</th>
<th>unit</th>
<th>value for argon</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>$\sigma$</td>
<td>$3.4 \times 10^{-10} m$</td>
</tr>
<tr>
<td>energy</td>
<td>$\epsilon$</td>
<td>$1.65 \times 10^{-21} J$</td>
</tr>
<tr>
<td>time</td>
<td>$\sigma (m / \epsilon)^{1/2}$</td>
<td>$2.17 \times 10^{-12} s$</td>
</tr>
<tr>
<td>velocity</td>
<td>$(\epsilon / m)^{1/2}$</td>
<td>$1.57 \times 10^{2} m/s$</td>
</tr>
<tr>
<td>force</td>
<td>$\epsilon / \sigma$</td>
<td>$4.85 \times 10^{-12} N$</td>
</tr>
<tr>
<td>temperature</td>
<td>$\epsilon / k$</td>
<td>$120 K$</td>
</tr>
<tr>
<td>pressure</td>
<td>$\epsilon / \sigma^2$</td>
<td>$1.43 \times 10^{-2} N m^{-1}$</td>
</tr>
</tbody>
</table>

8.5.1 Mouse Cursor

If you want to use different cursors for the mouse, you have to take a look at the java.awt.Cursor class. You can define a cursor for every component displayed on the screen. So if you have a canvas and want to notify the user that there is a calculation going on in this canvas, you just employ the commands:

```java
Canvas canv;
canv.setCursor(Cursor.WAIT_CURSOR); // calculation starts
.....
canv.setCursor(Cursor.DEFAULT_CURSOR); // calculation is finished
```

8.5.2 ScrollPanes

To use a ScrollPane you just instantiate one, use a Panel inside the ScrollPane if you want to put many components in it and at last add the ScrollPane to a Container (like a Frame).

```java
import java.awt.*;

/**
 * ScrollPaneDemo.java
 */

public class ScrollPaneDemo {

    public ScrollPaneDemo() {
    }

    public static void main(String[] args) {
        Frame f = new Frame("ScrollPaneDemo");
        f.setSize(50,100);

        ScrollPane sp = new ScrollPane(ScrollPane.SCROLLBARS ALWAYS);

        Label l = new Label("A very long test text for demonstration purposes.");
```
8.5.3 Properties and Resources

Properties are like Xdefaults in a UNIX environment and deliver information about the run-time environment. By reading the property list a program can set defaults, choose colors and fonts and more. In Java 1.1 there are 21 system properties defined. Only 9 of them can be accessed by applets, but an application has access to all of them.

So for example, you can get the file separator to distinguish between the / on UNIX systems and the \ on Windows systems. You could just use (both as application or applet accessible):

```java
String s = System.getProperty("file.separator");
System.out.println(s);
```

Some important properties are listed in table 8.2. The properties are usually defined in a file somewhere in the java installation. You can also define your own properties and use them for example to read parameters from local files in a browser. This is the only way to do this, because of security reasons.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Sample Values</th>
<th>Access in Applet</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.separator</td>
<td>File separator</td>
<td>/ or \</td>
<td>NO</td>
</tr>
<tr>
<td>java.vendor</td>
<td>JVM vendor</td>
<td>Netscape Communications</td>
<td>YES</td>
</tr>
<tr>
<td>line.separator</td>
<td>Line separator</td>
<td>\</td>
<td>YES</td>
</tr>
<tr>
<td>os.arch</td>
<td>Operating system architecture</td>
<td>x86 or 80486</td>
<td>YES</td>
</tr>
<tr>
<td>os.name</td>
<td>Operating system name</td>
<td>Linux or Windows NT</td>
<td>YES</td>
</tr>
<tr>
<td>path.separator</td>
<td>Path separator</td>
<td>; or ;</td>
<td>YES</td>
</tr>
<tr>
<td>user.home</td>
<td>Users home directory</td>
<td>/home/john</td>
<td>NO</td>
</tr>
<tr>
<td>user.name</td>
<td>Users login name</td>
<td>john</td>
<td>NO</td>
</tr>
</tbody>
</table>

Another interesting topic are the resource bundles. With this you can supply a file containing language dependent information to your Java program. This is similar to using your own properties (called server properties). The package responsible for the functionality is the `java.util.Properties` package. For a complete description look at the API documentation or take a look at [Zukowski, 1997].

8.5.4 paint(), repaint() and update()

It is now time to sort out these three methods, often used and often confused with each other. To get a deeper understanding of this topic and some of the other advanced aspects of the AWT, you actually need an understanding of threads, which will be discussed in great detail in section 10.3.5. For now it is enough to know that a thread is a “separate” program running beside yours.

In the molecular dynamics simulation we want to paint the trajectories of the individual particles in a display area (in Java this is called a Canvas). But if we change the positions of one particle (or more), we want the display to be refreshed, so that we can actually see the change on the display. Because this is not automatically done in Java if you change a component, you have to tell the JVM to update the component in question.

For example if you have a textfield, where you display the actual time of the simulation, it changes very often and of course you want the display to be updated all the time (do not forget that this is very time consuming). The first and easiest possibility is to call the `repaint()` method of a component (or a
container). The repaint method then in turn calls the `update()` method of all components contained in the component for which the repaint method has been called. Each of the update methods then calls the corresponding `paint()` method, which finally displays the change on the screen (see also figure 2.5).

Here an example: you create a `Panel`, which contains two labels. Now you start a simulation and change the second label. Now to update the display you have to call the `repaint()` method of the `Panel` object, which then updates the two labels.

The drawback (or advantage) of the `repaint()` method is that it does not update the components at once, but waits for some time to do it. If you call it again before `update()` starts, it does not start it twice, but only once. So it kind of collects and directs all calls to the update methods.

In our example you might want to redisplay the labels at once, because your simulation calculation is going on and does not allow the repaint method to start. In this case you have to call the `update()` method of the label components yourself and it will redisplay it at once.

A third possibility would be to call the `paint()` methods of the components to be repainted, but this is mostly cumbersome and should be avoided.

All this is very important for laying out a new program like the one we want to write now: a molecular dynamics program.

### 8.5.5 Events

Now we come to a very important part of writing GUIs: user interaction. A program which has graphical output is nice, but we still miss the possibility top input data or text using the graphical interface. In Java this is hidden in the `java.awt.event` class.

The event model we are discussing in a moment has been introduced into Java in version 1.1 and we are not going to discuss the old event model of Java 1.0.

If a user presses a key or pushes a mouse button or moves the mouse an event occurs. You can specify an object (e.g. a button), which should be notified if a special event, like e.g. a mouse button is pushed, occurs. There are many different events (mouse button, key pressed, etc.) and you can register an event listener for a certain event to an object (called event source, mostly components). For example you could register a listener for a mouse button pushed on a button. For that you have to implement the appropriate interface in your program and register the event listener.

As an example we have already met the case of a closable frame. By default the frame has a close button decoration on the right upper corner. But pushing the mouse button on this decoration does not have any impact. Usually you want the frame to be closed and the program to exit in that case. So we have to implement an interface for the pushing of a mouse button and register a listener with the frame:

```java
/* A closable Frame to demonstrate the use of listener and events */
import java.awt.*;
import java.awt.event.*;

public class ClosableFrame {
    public static void main(String[] args) {
        Frame f;

        f = new Frame("ClosableFrame");
        f.setSize(200,200);
        f.show();

        // Close Window event
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

In line 9-11 we instantiate a frame and set the size thereof. Then we display it on screen. In line 13-15 we add the listener (here a `WindowListener`) to the frame and in the same lines implement the interface
8.5. ADVANCED AWT FEATURES AND GUIS

(here the WindowAdapter with the windowClosing event). This shortcut of using events is called “inner classes”. It is the most easy way, if you do not want to write a separate class in a separate file. A second possibility is to implement the listener and override the corresponding methods (here you would implement a WindowListener and override the windowClosing() method).

Each event source (component) can have multiple event listeners registered on it. So you could specify a reaction for a mouse button on a label or a key pressed on the label. You could also register the same event listener on different event sources: a mouse button can be pressed on a button or on a canvas.

The events are represented by objects, which gives information about the event and identifies the event source (remember there could be more event sources for a certain event). For example the Button component has a method called addActionListener, which you can use. If you have registered this listener and the user clicks the mouse button on the button, the program stops and the listener is notified by firing an action event. The listener then calls the actionPerformed(ActionEvent event) method, which is the only available one for this listener, and executes the code therein. The arguments of the call to this method is a only a single ActionEvent. This object tells you exactly which kind of event has occured.

To see what kind of events are possible for a particular event source (component) you can take a look at the API documentation. Look at the methods of the listener classes of the java.awt.event package. Because for some of the listeners there are many different events possible (like the MouseListener) and you do not want to overwrite all the methods to implement the listener, you can use the so-called adapter classes. We have already met them above, where we have used the WindowAdapter. The WindowAdapter implements all 7 empty methods necessary to have a WindowListener and you can just override the methods you really need, like window closing above.

![Event Classes Diagram](image_url)

**Figure 8.7:** The most important event classes in Java and their structure. In these classes you can find the events available in Java. There are many events also outside of the AWT, which are not relevant for us.

At last a short summary or instruction of how to implement events in Java:

1. Choose a component (=event source), which you want to be responding to an event, e.g. a Button.
   ```java
   Button but = new Button("Push Me");
   ```
2. Take a look at the API documentation of the component, here the class java.awt.Button. At the beginning of the description of the class methods, you find all relevant listeners available. Here only the addActionListener() method is meaningful.
3. Now you know what listener to take and can register the listener with the component by using the above method. You only have to write the code for the reaction to the event (=to implement the listener interface). If there is only one method to be implemented or you want to implement all possible
**Figure 8.8:** The most important AWT listeners and their class structure. Look at the API documentation of the listeners to find all the available methods to be overridden, this shows you what kind of actions are possible to detect.

**Figure 8.9:** The available adapter interfaces for the AWT listener. These classes ease the writing of listeners by only overriding the methods you need, you just implement the appropriate event adapter interfaces.
methods of a listener, you just write your own listener by either using the implement keyword for your class and overriding the method names or you can use inner classes.

```java
/** inner class version */
but.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // here we could react to the pushed button
    }
});
```

```java
/** The implement and override version of the Event Listener Demonstration */
import java.awt.*;
import java.awt.event.*;

public class ButtonListenerTest implements ActionListener {
    // overriding the appropriate method
    public void actionPerformed(ActionEvent e) {
        // here we could react to the pushed button
        System.out.println("You pushed the Button");
    }
    // empty constructor
    public ButtonListenerTest() {}

    public static void main(String[] args) {
        // get a reference to the class
        // with the Listener implementation
        ButtonListenerTest blt = new ButtonListenerTest();
        // create a window to position the button inside it
        Frame f = new Frame("TestButtonListener");
        f.setSize(200, 200);
        // create the event source
        Button but = new Button("Push Me");
        // register the Listener
        but.addActionListener(blt);
        // display the window with the button
        f.add(but);
        f.show();
        // Wait for a long time
        try {
            Thread.sleep(100000);
        } catch (InterruptedException e) {}}
}
```

You should always remember not to write time intensive code in listeners, because you interrupt the whole program and only one listener method is awake at one time. So if you push the button again it will wait until the first listener method of the pushed button is done. This makes programs not very responsive.

### 8.5.6 A Complete GUI for the Molecular Dynamics Program

putting it all together ! ????
8.5.7 Features not Discussed in this Book

Not mentioned: Cut and Paste (Clipboard), Images, Sound

8.6 A Molecular Dynamics Program


Check der Gesamtenergie, Trajektorien angucken. Sensitivity of initial conditions: Drift in der Geschwindigkeit; alle Geschwindigkeiten gleich
Nur Listing mit dem code; kein GUI!!!
GUI als Appendix !!!

8.7 The Analysis of the Results

We already mentioned that the aim of a Molecular Dynamics simulation is to investigate macroscopic properties of a fluid starting from the microscopic dynamics of a system of particles. In this section we want to sketch some of the basic techniques which allow the extraction of useful information from the raw data. It is important to keep in mind that the conventional molecular dynamics simulation we described so far keep the energy constant. In other words, it is based upon the micro canonical \((E_0, V, N)\) ensemble.

8.7.1 The Pair Correlation Function

The best way to characterise the structure of a monoatomic fluid is to look at the pair-correlation function \(g(r)\). The radial distribution function \(g(r)\) is a measure of the correlations of the positions of the particles, which are induced by the interactions. Let \(n(r)\) denote the local particle density. This quantity will vary from point to point, and it will fluctuate around the mean value

\[
n = \langle n(r) \rangle = \frac{N}{V}.
\]

The local densities at two neighbouring points \(r_1\) and \(r_2\) will not be completely independent from each other. In fact, in the time average they show certain correlations. The right measure for these statistical correlations is the density–density pair correlation function

\[
G(r_1, r_2) = \frac{n(r_1)n(r_2)}{n^2}.
\]

The microscopic definition of the local density is, of course, a sum of \(\delta\)-functions concentrated around the position of the particles

\[
n(r) = \sum_{i=1}^{N} \delta(r - r_i),
\]

and the density–density correlation function is accordingly given by

\[
G(r_1, r_2) = \frac{1}{n^2} \left\langle \sum_{i=1}^{N} \sum_{j=1}^{N} \delta(r_i - r_1) \delta(r_j - r_2) \right\rangle.
\]

Introducing \(r_{12} = r_2 - r_1\) and \(r_{ij} = r_j - r_i\) the above equation can be written in the equivalent form

\[
G(r_1, r_2) = G(r_{12}) = \frac{1}{n} \delta(r_{12}) + \frac{V}{N^2} \left\langle \sum_{i=1}^{N} \sum_{j=1 \neq i}^{N} \delta(r_{12} - r_{ij}) \right\rangle.
\]
The function

\[
g(r_{12}) = \frac{V}{N^2} \left( \sum_{i=1}^{N} \sum_{j=1; j \neq i}^{N} \delta(r_{12} - r_{ij}). \right)
\]

is called the pair correlation function. For an isotropic Lennard–Jones fluid the pair correlation function does not depend upon the direction of the vector \( r_{12} \), but only on its absolute value,

\[
g(r_{12}) = g(r) = \frac{V}{4\pi r^2 N^2} \left( \sum_{i=1}^{N} \sum_{j=1; j \neq i}^{N} \delta(r_{12} - r_{ij}). \right)
\] (8.7)

The pair correlation function \( g(r) \) has a clear physical meaning. \( ng(r) \) is nothing but the mean number of particles in a spherical shell between \( (r, r + dr) \) around a given particle. For large values of \( r \) the product \( ng(r) \) converges against the mean particle density \( n \), i.e. we have

\[
g(r \to \infty) = 1.
\]

For short distances the interaction between the particles is repulsive, so that we must have

\[
g(r \to 0) = 0,
\]

The qualitative behaviour of \( g(r) \) can be seen in Fig. (8.10). After a forbidden region for \( r < \sigma \) which is caused by the fact that particles cannot penetrate each other, there is a sharp maximum which corresponds to the first shell of neighbouring molecules. This first shell induces a second forbidden region characterised by a minimum in \( g(r) \) which in turn induces again a maximum characterising a second shell of atoms, and so on until \( g(r) \) reaches its asymptotic value of 1.

\textbf{Figure 8.10:} Qualitative behaviour of the pair correlation function \( g(r) \) for a Lennard–Jones fluid.

In a simulation the function \( g(r) \) can be determined in the following way. \( ng(r) \) denotes the probability to find given a particle at \( r = 0 \) another particle in a distance \( r \). This probability can be estimated from the relative frequency, with which distances between the particles are found. For the computation of \( g(r) \) for \( 0 < r < 3\sigma \) we divide this length \( (3\sigma) \) into 100 parts \( i = 1, \ldots, 100 \) of equal length \( \Delta r = 3\sigma/100 \). In the code we will denote this quantity by \( \text{deltar} \). For a given configuration of the \( N \) particles we count how many of the pair distances fall in each of the intervals. Algorithmically, we have to calculate the distance \( r_{ij} \) for each pair of particles. For each \( r_{ij} \) we determine the number of the corresponding channel in the histogram according to

\[
n = [r_{ij}/\Delta r] + 1,
\]

where \([x]\) denotes the nearest integer number smaller than \( x \). The value of \( g(n) \) in the \( n \)th channel can then be increased by 1, and so on. This part of the code may look like

```c
for (int i=1; i<N; i++) {
    for (int j=i+1; j<N+1; j++) {
        xx = x[i] - x[j];
        yy = y[i] - y[j];
        zz = z[i] - z[j];
        // periodic boundary conditions
        if (xx > L/2) xx = xx - L;
        if (xx < -L/2) xx = xx + L;
        if (yy > L/2) yy = yy - L;
        if (yy < -L/2) yy = yy + L;
```
Given a configuration in the mean state, say \( r_i \), and a certain configuration, say \( r_j \), we would need to check \( \Delta g = g_j - g_i \) significantly. In order to obtain \( \Delta g \) we would use \( g_j \), after using \( g_i \). This exact relation is the starting point for several theoretical investigations. E.g., with some simplifying assumptions this relation makes possible the theoretical calculation of \( g(r) \).

For a given pair potential \( U(r) \) the canonical configurational partition function of a \( N \) particle system reads

\[
Q = \frac{1}{N!} \int \cdots \int \exp \left( -\frac{1}{2} \sum_{i,j \neq i} U(r_{ij})/kT \right) d\vec{r}_1 \cdots d\vec{r}_N.
\]

The above simply means that the probability to find a certain configuration, say \((\vec{r}_1, \ldots, \vec{r}_N)\), is given by

\[
\frac{1}{N!Q} \exp \left( -\frac{1}{2} \sum_{i,j \neq i} U(r_{ij})/kT \right) d\vec{r}_1 \cdots d\vec{r}_N.
\]

Hence, the probability to find a particle at \( \vec{r}_1 \) and, at the same time a particle at \( \vec{r}_2 \) regardless of the position of the other particles is given by the two-particle distribution function

\[
n(\vec{r}_1, \vec{r}_2) = \frac{1}{(N-2)!Q} \int \cdots \int \exp \left( -\frac{1}{2} \sum_{i,j \neq i} U(r_{ij})/kT \right) d\vec{r}_3 \cdots d\vec{r}_N.
\]

Hence, the probability to find another particle at \( \vec{r}_2 \) if \( n < 100 \) is

\[
g[n] = g[n] + 2
\]

\[
\text{for} \quad \text{int \ } i = 1; \quad i < 101; \quad i++\{
\quad \text{deltan} = \text{Math.Pi}*(N-1)/(3*\text{Volume})
\quad \quad * (\text{Math.pow}(i, 3) - \text{Math.pow}(i-1, 3)*\text{Math.pow}((\text{deltan}, 3));
\quad g[i] = g[i] / (\text{Nconfig*delran*}(N-1));
\}
\]

where \( \text{Nconfig} \) denotes the number of configurations which have been analysed.

The computation of \( g(n) \) can, of course, be placed in a part of the program where the distances are computed anyway. However, since we have to compute the function only every 10–50 steps an eventual redundant calculation of the distances does not affect significantly the performance of the code.

The pair correlation function is of great theoretical and practical importance. The function \( g(r) \) is directly accessible in neutron and Roentgen scattering experiments. From a theoretical point of view there is an exact relation between the pair potential \( U(r) \) and \( g(r) \). This exact relation is the starting point for several theoretical investigations. E.g., with some simplifying assumptions this relation makes possible the theoretical calculation of \( g(r) \).
It is now possible to demonstrate that the above function is related to the pair correlation function through the equation

\[ g(r) = g(r_1, r_2) = \frac{1}{n^2} n(q_1, q_2). \]

It is this equation which is at the basis of several microscopic theories of fluids. By far the most important property of the function \( g(r) \) is that it allows the determination of thermodynamic equilibrium quantities. The internal energy \( U_i \) is nothing but the mean potential energy of the total system,

\[ U_i = \left\langle \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} U(r_{ij}) \right\rangle. \]

Writing

\[ U(r_{ij}) = \int dr U(r) \delta(r - r_{ij}), \]

we obtain the following expression for the internal energy

\[ U_i = \frac{1}{2} \int dr U(r) \left\langle \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} \delta(r - r_{ij}) \right\rangle. \]

Inserting the definition (8.7) of the pair correlation function \( g(r) \) in the above equation we get the energy equation

\[ U_i = \frac{N^2}{2V} \int dr U(r) g(r). \]

In a similar way we can deduce from the virial theorem the following equation for the pressure \( p \)

\[ p = \frac{NkT}{V} - \frac{1}{6V} \left\langle \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} \mathbf{r}_{ij} \cdot \mathbf{r}_{ij} \right\rangle, \]

which can be expressed with the help of the pair correlation function in the following form

\[ p = \frac{NkT}{V} - \frac{n^2}{6} \int_V dV \frac{\partial U(r)}{\partial r} g(r). \]

**Figure 8.11:** The pressure

### 8.7.2 Thermodynamic Quantities

From the principle of equipartition of energy it follows that at equilibrium each degree of freedom in the system has a kinetic energy \( \frac{1}{2} kT \). Hence for \( N \) particles in a 3d simulation the kinetic energy will be given by

\[ \left\langle \frac{m_i v_i^2}{2} \right\rangle = \frac{3}{2} NkT. \]

The above expression is at the basis of the evaluation of the temperature, since the quantity between angular brackets is easily evaluated as a time average.

Hier fehlen noch einige Bemerkungen.

Def. von Pot. Energie, usw. Diesen Abschnitt vor \( g(r) \)!!!!!!
8.7.3 Dynamical Quantities

Up to now we have considered only static equilibrium properties of the fluid. Time-dependent correlation functions are important tools for the description of dynamical phenomena. In particular we are interested in understanding the transport of particles from the equilibrium properties of the fluid. A typical example is the calculation of the diffusion coefficient from the one particle velocity autocorrelation function \( C(\tau) \)

\[
C(\tau) = \frac{\langle \vec{v}(\tau) \cdot \vec{v}(0) \rangle}{v^2},
\]

or from the square of the displacement \( \Delta(t) = \vec{r}(t) - \vec{r}(0) \). As we will see shortly the two methods are equivalent.

In a molecular dynamics simulation the velocity autocorrelation function is computed as a histogram of the function \( C(\tau) \) from the velocities of the particles \( \vec{v}_i(t_k) \):

\[
C(n\Delta t) = \frac{1}{N} \left\{ \frac{1}{S} \sum_{t=1}^{S} \sum_{i=1}^{N} \vec{v}_i(t_k) \cdot \vec{v}_i(t_k + n\Delta t) \right\}.\]

Of course, the time \( t_k \) have to be chosen such that only statistical independent pieces of the trajectories are evaluated. Usually it is sufficient to choose \( \Delta t - s = 50\Delta t \).

An autocorrelation function for a Lennard-Jones fluid can be seen in Fig. (8.13).

Figure 8.13: Plot of the velocity autocorrelation function of a Lennard–Jones fluid.

The figure makes evident that the particles forget rather rapidly their starting velocity. As a consequence of the interaction with the neighbouring atoms the magnitude and the direction of the velocity changes rapidly and the product \( \vec{v}(0) \cdot \vec{v}(\tau) \) decreases to zero. Since each particle is surrounded by a shell of neighbours after some time there is a reversal of the direction of motion. This explains the negative part of the autocorrelation function. For higher temperature or lower densities this effect does not occur. For larger time the memory of the original motions completely lost and the correlation function tends definitely to zero.

As we mentioned at the beginning of this subsection the velocity autocorrelation function is deeply related to the diffusive Brownian motion of the atoms in the fluid. In particular we have the simple relation between the velocity autocorrelation function and the displacement

\[
\frac{d^2}{dt^2} \langle \Delta(\tau)^2 \rangle = 2 \langle \vec{v}(0) \cdot \vec{v}(\tau) \rangle \tag{8.8}
\]

The above relation is easily demonstrated by looking at

\[
\frac{d}{dt} \Delta(t)^2 = \frac{d}{dt} \left[ \vec{v}(t) - \vec{r}(0) \right]^2 = 2 \vec{v}_i \cdot \left[ \vec{v}(t) - \vec{r}(0) \right] = 2 \int_0^t d\tau \vec{v}(\tau) \cdot \vec{v}(\tau).
\]

Because of the symmetry of \( \langle \vec{v}(0) \cdot \vec{v}(\tau) \rangle \) for arbitrary times \( t \) we conclude that

\[
\frac{d}{dt} \Delta(t)^2 = 2 \int_0^t d\tau \vec{v}(\tau) \cdot \vec{v}(0).
\]
The statement (8.8) follows immediately.

We know also that the displacement $\langle \Delta^2 \rangle$ is related to the diffusion constant by the relation

$$\lim_{\tau \to \infty} \langle \Delta^2(\tau) \rangle / \tau = 6D.$$  

A very important relation follows now

$$D = \frac{1}{3} \int_0^\infty d\tau \langle \Psi(0) \Psi(\tau) \rangle.$$  

The above equation is a typical example of a Kubo relation. Similar relations hold also for other thermodynamic transport coefficients.

8.8 Molecular Dynamics at Constant Temperature

Up to now the equilibrium properties of the system were fixed by the volume of the simulation cell, by the initial position of the particles and by the initial velocities. In such a micro canonical ensemble ($NV$) it is easy to determine the constant energy as the sum of the potential and the kinetic energy. As we have seen in the previous runs of the program MolDyn the temperature and the pressure of the fluid fluctuate around some mean values, which can be computed as time averages. In many thermodynamic applications we are interested in having an ensemble at constant temperature, i.e. a canonical ensemble ($NT$). It is therefore the subject of the section to derive a canonical molecular dynamics simulation algorithm.

8.8.1 Velocity Rescaling

The easiest way to keep the temperature fixed to a wished temperature is to rescale the velocities at each time step. Let us be more precise. We denote by $T_0$ the desired temperature and by $K_0$ the corresponding kinetic energy. By $T$ and $K$ we denoted the actual temperature, respectively kinetic energy of the $N$-particle system. After each time step we rescale the velocities according to the prescription

$$v_i \rightarrow \left( \frac{T_0}{T} \right)^{1/2} v_i = \left( \frac{K_0}{K} \right)^{1/2} v_i.$$  

To put it differently, after each time step $\Delta t$ we correct the velocity by an amount $\delta v_i$

$$v_i \rightarrow v_i + \delta v_i$$  

where

$$\delta v_i = \left[ \left( \frac{K_0}{K} \right)^{1/2} - 1 \right] v_i = \left[ \frac{K_0}{K + \delta K} \right]^{1/2} v_i.$$  

In the above expression we have introduced the change in the kinetic energy $\delta K = K - K_0$, $|\delta K| \ll 1$. Expanding the terms on the squared brackets up to terms of first order in $\delta K$ we obtain

$$\delta v_i \approx -\frac{K}{2K_0} v_i.$$  

Since, the total energy $K + E_{pot}$ is constant the change in the kinetic energy corresponds to a change of the potential energy

$$\delta K = -\delta E_{pot}.$$  

The change in the velocity can therefore be written as

$$\delta v_i \approx -\frac{\delta E_{pot}}{v_i} \delta t.$$  

The statement (8.8) follows immediately.

We know also that the displacement $\langle \Delta^2 \rangle$ is related to the diffusion constant by the relation

$$\lim_{\tau \to \infty} \langle \Delta^2(\tau) \rangle / \tau = 6D.$$  

A very important relation follows now

$$D = \frac{1}{3} \int_0^\infty d\tau \langle \Psi(0) \Psi(\tau) \rangle.$$  

The above equation is a typical example of a Kubo relation. Similar relations hold also for other thermodynamic transport coefficients.

8.8 Molecular Dynamics at Constant Temperature

Up to now the equilibrium properties of the system were fixed by the volume of the simulation cell, by the initial position of the particles and by the initial velocities. In such a micro canonical ensemble ($NV$) it is easy to determine the constant energy as the sum of the potential and the kinetic energy. As we have seen in the previous runs of the program MolDyn the temperature and the pressure of the fluid fluctuate around some mean values, which can be computed as time averages. In many thermodynamic applications we are interested in having an ensemble at constant temperature, i.e. a canonical ensemble ($NT$). It is therefore the subject of the section to derive a canonical molecular dynamics simulation algorithm.

8.8.1 Velocity Rescaling

The easiest way to keep the temperature fixed to a wished temperature is to rescale the velocities at each time step. Let us be more precise. We denote by $T_0$ the desired temperature and by $K_0$ the corresponding kinetic energy. By $T$ and $K$ we denoted the actual temperature, respectively kinetic energy of the $N$-particle system. After each time step we rescale the velocities according to the prescription

$$v_i \rightarrow \left( \frac{T_0}{T} \right)^{1/2} v_i = \left( \frac{K_0}{K} \right)^{1/2} v_i.$$  

To put it differently, after each time step $\Delta t$ we correct the velocity by an amount $\delta v_i$

$$v_i \rightarrow v_i + \delta v_i$$  

where

$$\delta v_i = \left[ \left( \frac{K_0}{K} \right)^{1/2} - 1 \right] v_i = \left[ \frac{K_0}{K + \delta K} \right]^{1/2} v_i.$$  

In the above expression we have introduced the change in the kinetic energy $\delta K = K - K_0$, $|\delta K| \ll 1$. Expanding the terms on the squared brackets up to terms of first order in $\delta K$ we obtain

$$\delta v_i \approx -\frac{K}{2K_0} v_i.$$  

Since, the total energy $K + E_{pot}$ is constant the change in the kinetic energy corresponds to a change of the potential energy

$$\delta K = -\delta E_{pot}.$$  

The change in the velocity can therefore be written as

$$\delta v_i \approx -\frac{\delta E_{pot}}{v_i} \delta t,$$
where
\[ \zeta_G = -\frac{1}{2K_0} \frac{E_{pot}}{\delta \tau}. \]

The parameter \( \zeta_G \) is proportional to the change of the potential energy in a time step. In the next subsection, we will see that the simple rescaling algorithm introduced here may be given a more profound theoretical foundation.

### 8.8.2 The Gaussian Thermostat

In 1929 Gauss formulated a general principle of mechanics for the description of systems with holonom and nonholonom constraints. The Gaussian principle of least Zwang introduces as a measure of the constraint

\[ Z = \sum_k m_k (\ddot{x}_k - \ddot{F}_k)^2, \]

where, of course,
\[ \ddot{x}_k = \ddot{F}_k \]
describes the free motion. In the Gaussian notation the quantity \( Z \) is called the sum over the squared of the "lost forces". The constraint equations of motion are obtained from the variational principle

\[ \delta Z = 0. \]

During the variation of the functional \( Z \) we have to keep fixed (i) the state of the system

\[ \delta \dot{x}_k = 0; \quad \delta \ddot{x}_k = 0, \]

(ii) the (nonholonomic) constraint

\[ f_i(\ldots, \dot{x}^r_k; \ddot{x}^r_k, \ldots) \quad i = 1, \ldots, r \]

and (iii) the forces acting on the system and the masses

\[ \delta F_k = 0; \quad \delta m_k = 0. \]

By the method of Lagrangian multipliers we obtain

\[ \delta Z = 2 \sum_k (m_k \dot{x}_k - \dot{F}_k - \sum_i \lambda_i \frac{\partial f_i}{\partial \dot{x}_k}) \delta \dot{x}_k = 0. \]

Note that we made use of the fact that it follows from

\[ \sum_k \left\{ \frac{\partial f_i}{\partial \dot{x}_k} \delta \dot{x}_k + \frac{\partial f_i}{\partial \ddot{x}_k} \delta \ddot{x}_k \right\} = 0. \quad (8.9) \]

that

\[ \sum_k \frac{\partial f_i}{\partial \ddot{x}_k} \delta \ddot{x}_k = 0. \]

It follows now from Eq. (8.9) that the equations of motion read

\[ m_k \ddot{x}_k = \ddot{F}_k + \sum_{i=1}^r \lambda_i \frac{\partial f_i}{\partial \dot{x}_k}. \quad (8.10) \]
Let us now apply the Gaussian principle to a dynamical system which we want to keep at a fixed temperature. The constraint is

$$f(\dot{x}_1, \ldots, \dot{x}_N) = \sum_{i=1}^{N} \frac{1}{2} m_i \dot{x}_i^2 - \frac{3}{2} N k T_0 = 0.$$  \quad (8.11)

Since,

$$\frac{\partial f}{\partial \dot{x}_i} = m_i \ddot{x}_i$$

the constraint equation of motion follows immediately from the Gaussian principle (Eq. (8.10))

$$m_i \ddot{x}_i = \vec{F}_k + \lambda m_i \dot{x}_k.$$  \quad (8.12)

In order to determine the Lagrange multiplier $\lambda$, we take the time derivative of the constraint (8.11) and find

$$\sum_i m_i \ddot{x}_i \cdot \dot{x}_i = 0.$$  \quad (8.13)

Multiplying Eq. (8.12) by $\dddot{x}_k$ and summing over all particles we get

$$\sum_k m_k \dddot{x}_k \cdot \ddot{x}_k = \sum_k \dddot{x}_k \cdot \vec{F}_k + \lambda \sum_k m_k \dot{x}_k^2.$$  

Exploiting Eq. (8.13) we obtain the Lagrangian multiplier $\lambda$

$$\lambda = -\frac{\sum_k \dddot{x}_k \cdot \vec{F}_k}{\sum_k m_k \dot{x}_k^2}.$$  \quad (8.14)

The equations of motion thermalized with the Gaussian "thermostat" finally read

$$m_k \ddot{x}_k = \vec{F}_k - \zeta_G m_k \dot{x}_k$$

where the constant factor $\zeta_G = -\lambda$. The equivalence of the above thermostat and of the rescaling of the velocities is now easily recognised. It follows immediately from Eq. (8.14) that

$$2 K \zeta_G = -\frac{\text{d} E_{\text{pot}}}{\text{d} t}.$$  

Thus, if at the beginning of the simulation the desired kinetic energy is $K_0$ then, the rescaling the velocities is equivalent to modifying the equations of motion according to the Gaussian principle.

### 8.9 Non–Equilibrium Molecular Dynamics

So far we have considered systems at equilibrium. Non–equilibrium molecular dynamics (NEMD) is a generalisation of molecular dynamics which tries to investigate non equilibrium properties of fluids Evans und Morriss [1984]. A typical non equilibrium situation is plane Couette flow. A simple shear flow in the $x$ direction with the velocity gradient in the $y$ direction is characterized by the shear rate $\gamma$ which is given by

$$\gamma = \frac{\partial v_x}{\partial y}.$$  

In other words the velocity gradient is of the form

$$\nabla \vec{v}(\vec{r}) = \gamma \vec{n} \hat{n}_{xy}$$
where \( \hat{n}_x \) is the unit vector in the \( x \)-direction. Such a steady uniform shearing motion may be imagined to be driven by moving boundaries normal to the \( y \) axis at \( y = \pm \infty \). The velocity field of a plane Couette flow is depicted in Fig. (8.14).

A typical question to be answered by a NEMD simulation could be the following one: Does the viscosity \( \eta \) of the fluid depend on the shear rate? We will not discuss here the theoretical foundations of this question. Quantities like the viscosity may be calculated directly by evaluating the corresponding kinetic expressions or with the help of Green–Kubo formulas. Here, we want only to sketch how a NEMD simulation may be performed Evans [1987]. A peculiarity of the driven Couette flow is that it is possible to design an algorithm which uses only the boundary conditions to drive the non-equilibrium state.

Again we consider a cubic simulation cell with periodic boundary conditions. At the origin of the simulation cube the streaming velocity is chosen to be zero,

\[
\mathbf{u}(0) = 0.
\]

It is important to be aware of the fact that in a flow the velocity of a particle, say \( i \), is the sum of two contributions: a peculiar velocity \( \mathbf{\dot{r}}_i \) and a streaming velocity

\[
\mathbf{\dot{r}}_i = \mathbf{\dot{r}}_i + \mathbf{u}(\mathbf{\dot{r}}_i).
\]

Of course, the peculiar velocity is periodic, while the streaming velocity is not!

At time \( t = 0 \), i.e. before the shear flow is switched on, we have the usual periodic boundary condition

\[
\mathbf{r}_i = (\mathbf{r}_i) \mod L.
\]

Simple shear flow may be be generated by moving image particles undergoing an ideal Couette flow with the prescribed shear rate. Let us switch on the flow at time \( t = 0 \). Then, at a later time \( t \) the image cells above (below) the simulation box have moved in the \( x \) direction to the right (left) by a distance \( \gamma L \mod L \) where \( L \) is the linear dimension of the cubic box (see Fig. (8.15)).

**Figure 8.14:** The velocity field in a plane Couette flow.

Formally we have the following situation. Particle \( i \) and its two images \( i' \) and \( i'' \) are located at time \( t_1 \) at the positions

\[
\mathbf{r}_i(t_1) = \int_0^{t_1} dt \left( \mathbf{\dot{r}}_i + \gamma \hat{n}_x \right) = \mathbf{\dot{r}}_i + \gamma L \hat{n}_x t_1
\]

and, respectively,

\[
\mathbf{r}_{i'}(t_1) = \mathbf{\dot{r}}_i - \gamma L \hat{n}_x t_1.
\]

Thus, the method of imaging particles in a system under shear is easily formulated. If particle \( i \) passes through either face of the cube which is parallel to the \( y \)-axis, the periodic boundary conditions are unchanged

\[
\mathbf{r}_{i'}^{\text{new}} = (\mathbf{r}_i) \mod L,
\]

If particle \( i \) passes through either face which is parallel to the \( x \) axis then it is replaced in the simulation cube by one of its images \( i' \) or \( i'' \). If particle \( i \) passes through the top face \( (y = L) \) then its coordinate \( \mathbf{r}_i \) is replaced by \( \mathbf{r}_{i'}^{\prime} \),

\[
\mathbf{r}_{i'}^{\text{new}} = (\mathbf{r}_{i'}) \mod L = (\mathbf{r}_i - \gamma L \hat{n}_x t_1) \mod L.
\]

**Figure 8.15:** Moving periodic images for the simulation of a plane Couette flow.
8.9. NON–EQUILIBRIUM MOLECULAR DYNAMICS

Obviously, the streaming velocity at \( \vec{r}_i \) is different to that at \( \vec{r}_j \), so we have to correct the velocity accordingly and \( \vec{r}_i \) becomes
\[
\vec{r}_i^{\text{new}} = \vec{r}_i' = \vec{r}_i - \gamma \vec{L} \hat{x}.
\]

Analogously, if the particle \( i \) passes through the bottom face of the simulation cube (\( y = 0 \)) then \( \vec{r}_i \) is replaced by \( \vec{r}_i' \)
\[
\vec{r}_i^{\text{new}} = (\vec{r}_i')_{\text{mod} L} = (\vec{r}_i' + \gamma \vec{L} \hat{x}, 0, 0)_{\text{mod} L}.
\]

and its velocity becomes
\[
\dot{\vec{r}}_i^{\text{new}} = \dot{\vec{r}}_i' = \dot{\vec{r}}_i + \gamma \vec{L} \hat{x}.
\]

These are the so–called ”sliding brick” periodic boundary conditions.

There are also alternative methods to simulate Couette flow. These methods are based on the use of shearing periodic boundary conditions as well as on Non–Newtonian equations of motion. They can be applied to shear flows with time dependent shear rates. The discussion of these more refined simulation techniques (SLLOD dynamics) Allen und Tildesley [1987] are beyond the scope of the present book.

With the help of the above NEMD algorithm it is possible to calculate the (Non–Newtonian) viscosity of the fluid from the Cartesian components of the stress tensor \( \sigma_{\mu \nu} = -\rho_{\mu \nu} \) or of the pressure tensor \( p_{\mu \nu} \).

The latter is the sum of the kinetic and the potential contributions:
\[
\rho_{\mu \nu} = \rho_{\mu \nu}^{\text{kin}} + \rho_{\mu \nu}^{\text{pot}},
\]
\[
\rho_{\mu \nu}^{\text{kin}} = \sum_{\nu=1}^{N} m_i c_i^j \epsilon_i^j,
\]
\[
\rho_{\mu \nu}^{\text{pot}} = \frac{1}{2} \sum_{ij} \epsilon_{\mu \nu}^{ij} \cdot \vec{F}^{ij}.
\]

In the above equations \( \vec{r}^{ij} = \vec{r}_i - \vec{r}_j \) is the relative position vector of particles \( i \) and \( j \) and \( \vec{F}^{ij} \) is the force acting between them. The Greek subscripts \( \mu, \nu \) assume the values \( x, y, z \) and denote the Cartesian components of the corresponding vectors. For the flow geometry we consider here the (non–Newtonian) viscosity \( \eta \) is obtained through
\[
\eta = \frac{\sigma_{xy}}{\gamma} = -\frac{\rho_{xy}}{\gamma}.
\]

In the simulation the kinetic and the potential contribution to the pressure tensor and to the viscosity can be computed separately. Usually it is necessary to estimate them as time averages over \( 10^3 \) to \( 10^6 \) time steps. For dense fluids the potential contribution will be the dominant terms, whereas for dilute gases the kinetic contribution will be the more important. It is interesting to look at the shear–rate dependence of the viscosity. Four regimes can be identified. In the first Newtonian regime at low shear rates (\( \gamma < 0.1 \) in Lennard–Jones units) the viscosity is independent from the shear rate. For \( 0.2 \gamma < 2 \) a week shear thinning is observed. This means that the viscosity decreases for increasing shear rate. At larger shear rates for \( 2 < \gamma < 20 \) a strong shear thinning is observed. And finally for even larger shear rates \( \gamma > 20 \) shear thickening is observed, i.e., the viscosity increases with increasing shear rate Hess [1996].

Code der Simulation: Nur periodic boundary conditions; Trajektorien anschauen fuer grosse Scherraten; \( g(r) \), usw.

Appendix: The GUI!!!
Bibliography


Chapter 9

Monte-Carlo Methods

9.1 The \(M(RT)^2\) Algorithm

In a certain sense this chapter is the logical continuation of Chap. xx, in which we have learned how to sample random variables according to a prescribed distribution. We want to discuss an advanced sampling technique first described in a paper by Metropolis, Rosenbluth, Rosenbluth, Teller and Teller, which we will name the \(M(RT)^2\)-algorithm Metropolis et al. [1953]. In the literature the same algorithms often referred to simply as the Metropolis algorithm. The method bears some relation to the rejection techniques: tentative values that are proposed explicitly may be accepted or rejected.

The \(M(RT)^2\)-algorithm is often used as synonym for Monte Carlo algorithms. This stems from the fact that it is very simple and, at the same time, very powerful. It can be used to sample essentially any density function. The only disadvantage of the method is that the sampling is correct only asymptotically and that successive sampled variables are often strongly correlated. Besides the original paper there are many excellent introductions to the Monte Carlo algorithm Binder [1997]; Binder und Heermann [1988]. Here, we will follow a more general approach Kalos und Whitlock [1986]

It is well-known from statistical physics that the equilibrium properties are essentially independent from the kinetics of the system. We consider a system described by a point \(X\) in some phase space \(\Omega\). The time evolution of the system is governed by a stochastic transitions. Thus the kinetics of the system is defined in terms of a conditional transition probability density function \(W(X|Y)\), describing the probability for a system known to be in state \(Y\) to jump in state near \(X\). The quantity \(W(X|Y)\) models the physical process.

A system evolves towards equilibrium and stays there if on average the system is likely to move from \(Y\) into \(X\) as to move exactly in the reverse direction. If the probability density to observe the system in equilibrium is \(P_{eq}(X)\) then the kinetics must satisfy

\[ W(X|Y)P_{eq}(Y) = W(Y|X)P_{eq}(X). \]

Of course, \(W(X|Y)P_{eq}(Y)\) is the probability for moving from \(Y\) to \(X\), since it is the product of the probability of being in \(Y\) times the conditional probability to move from \(Y\) to \(X\). The above relation is called detailed balance.

In statistical physics, a model of the system would be defined in terms of \(W(X|Y)\). A typical task would be the determination of the equilibrium distribution. It is important to realize that the aim of the \(M(RT)^2\) algorithm is the opposite! The typical task of a MC simulation, as we will see in the applications discussed below, is to calculate averages of some quantities for a given ensemble, i.e., for the canonical one. Formulated more abstractly, the task of the \(M(RT)^2\) algorithm is to find a convenient kinetics, i.e. a \(W(X|Y)\), that will equilibrate the system towards a given \(P_{eq}(X)\). In this way we are then able to compute in a simple way ensemble averages. The basic idea of the \(M(RT)^2\) algorithm is to write \(W(X|Y)\) in the form

\[ W(X|Y) = \lambda(Y|X)\tau(Y|X), \]
where \( T(X'|Y) \) is a distribution which proposes the transition from \( Y \) to \( X' \). The \( T(X'|Y) \) are normalised such that
\[
\int dXT(X'|Y) = 1.
\]

By comparison of \( P_{\text{eq}}(X') \) with \( P_{\text{eq}}(Y) \) and taking into account \( T \), the system is either moved to \( X' \) (move accepted) or the system stays at \( Y \) (move rejected). Of course, the probability of acceptance \( A(X|Y) \) has to be calculated so that the detailed balance condition is satisfied. In other words:
\[
A(X|Y)T(X'|Y)P_{\text{eq}}(Y) = A(Y|X)T(Y|X)P_{\text{eq}}(X).
\]

It is helpful to define at this point the quantities
\[
q(X|Y) = \frac{T(Y|X)P_{\text{eq}}(X)}{T(X'|Y)P_{\text{eq}}(Y)} \geq 0.
\]

This quantity plays a central role in the calculation of the probabilities to accept or reject a move. A possibility to define the acceptance probability \( A \) is namely
\[
A(X|Y) = \min(1, q(X|Y)).
\]

For a given \( P_{\text{eq}}(X) \), when \( X \) is assumed to be some vector in \( \mathbb{R}^n \), the M(RT)\(^2 \) algorithm establishes a random walk. At each step in the random walk there is a transition density \( T(X|Y) \) for a move from \( Y \) to \( X \). Let us now denote the steps of this random walk by \( X_1, X_2, \ldots, X_N \). Each of the \( X_i \) is a random variable with a corresponding probability density \( \Phi_1, \ldots, \Phi_N \). As we will demonstrate, by a correct choice of \( A \), the asymptotic distribution of \( X \) will be \( P_{\text{eq}}(X) \).

We are now in the position to describe the algorithm. We assume that at the \( n \)th step of the random walk the value of \( X \) is \( X_n \).

(i) Sample a possible next value for \( X \), say \( X_{n+1}' \), from \( T(X_{n+1}'|X) \).

(ii) Compute the probability for accepting \( X_{n+1}' \). If \( q(X_{n+1}'|X) > 1 \) then \( A(X_{n+1}'|X) = 1 \) and \( X_{n+1}' \) is accepted. If \( q(X_{n+1}'|X) < 1 \), then \( A(X_{n+1}'|X) = q(X_{n+1}'|X) \). Thus, with probability \( A(X_{n+1}'|X) \) we accept the move \( X_{n+1} = X_{n+1}' \), otherwise we reject the move and \( X_{n+1} = X_n \).

Having defined the algorithm let us now look at the asymptotic properties of the random walk. Let \( \Phi_n(X) \) be the distribution of the values \( X_n \). The distribution \( \Phi_{n+1} \) is the sum of two contributions. The first one is the probability of moving in the vicinity of \( dX \) of \( X \) when we successfully move from any point \( Y \):
\[
\int dYA(X|Y)T(X|Y)\Phi_n(Y),
\]

The second one is the probability not to move away from \( X \), i.e., that a move out of \( X \) is not accepted:
\[
\Phi_n(X) \int dY[1 - A(Y|X)]T(Y|X).
\]

Summing up the two contributions we find
\[
\Phi_{n+1}(X) = \int dYA(X|Y)T(X|Y)\Phi_n(Y) + \Phi_n(X) \int dY[1 - A(Y|X)]T(Y|X),
\]

which is a recursion relation to the determination of \( \Phi_n \). It can be proven that the dynamical system generated by the M(RT)\(^2 \) algorithm is ergodic: the random walk starting at \( X \) may return in the neighbourhood of \( X \) but does not do so periodically. According to a theorem by Feller, if the random walk defines a system that is ergodic, then there exists an asymptotic probability density function and is unique if, \( P_{\text{eq}}(X) \) is a fixed point of the above recursion
\[
\Phi_n(X) = P_{\text{eq}}(X) \Rightarrow \Phi_{n+1}(X) = P_{\text{eq}}(X),
\]
9.2. **THE ISING MODEL**

A typical application of the M(RT)²-algorithm in statistical physics is the numerical investigation of the order–disorder transitions. The Ising model is the easiest model for interacting microscopic degrees of freedom. One considers a lattice composed of two different types of objects, say $A$ and $B$. The objects interact only with their nearest neighbours. The interaction between object $A$ and object $B$ is $V_{AB}$. The qualitative behaviour of the system at temperature $T = 0$ is easily investigated. We consider two cases:

(i) If $V_{AB} > (V_{AA} + V_{BB})/2$ then the configuration which is energetically favoured is one in which objects $A$ all neighbour one another and objects $B$ all neighbour one another. The lattice is split into domains, one containing only objects $A$, the other containing only objects $B$.

(ii) If $V_{AB} < (V_{AA} + V_{BB})/2$, the configuration in which objects $A$ and $B$ alternate will be energetically favoured.

Obviously the above configurations both belong to ordered states. Raising the temperature of the system, the thermal energy $kT$ will tend to randomise the positions of $A$ and $B$, which at some temperature of the system will behave completely disordered. The mathematical model of such systems is the Ising model, and because of the normalisation of the conditional transition probability $T$ we finally get

$$\Phi_{n+1}(X) = P_{eq}(X).$$

Thus, we have shown that $P_{eq}(X)$ is the asymptotic distribution of the random walk. From the practical point of view, this means that we have always to throw away $L$ steps of the random walk until the steps are sampled from $P_{eq}(X)$. $L$ may be difficult to estimate in advance. However it is clear that it may be minimised by choosing $\Phi_1(X)$ as close as possible to $P_{eq}(X)$.

In a Monte Carlo simulation we want to evaluate quantities of the form

$$G = \frac{\int dX \sum_{n=0}^{L} g(X_n) \rho_{eq}(X_n)}{\int dX \rho_{eq}(X)}.$$  

For example, in a Monte Carlo simulation of a many body system $G$ might be the energy. In view of the above remark, the averaging in the MC simulation begins only after the initial $L$ steps have been thrown away:

$$G = \sum_{n=L+1}^{L+N} \frac{g(X_n)}{N}.$$  

It is also important to keep in mind that the successive $X_n$ are not independent and that there may be correlations. Thus the calculated variance of $G$ will be larger than if the steps were independent. We will return to this point later.

A last remark. The form for the probability of accepting a move is not restricted to the one given above. Another relation which may be used is

$$A'(X|Y) = \frac{q(X|Y)}{1+q(X|Y)}.$$  

### 9.2.1 The Ising Model

A typical application of the M(RT)²-algorithm in statistical physics is the numerical investigation of the order–disorder transitions. The Ising model is the easiest model for interacting microscopic degrees of freedom. One considers a lattice composed of two different types of objects, say $A$ and $B$. The objects interact only with their nearest neighbours. The interaction between object $A$ and object $B$ is $V_{AB}$. The qualitative behaviour of the system at temperature $T = 0$ is easily investigated. We consider two cases:

(i) If $V_{AB} > (V_{AA} + V_{BB})/2$ then the configuration which is energetically favoured is one in which objects $A$ all neighbour one another and objects $B$ all neighbour one another. The lattice is split into domains, one containing only objects $A$, the other containing only objects $B$.

(ii) If $V_{AB} < (V_{AA} + V_{BB})/2$, the configuration in which objects $A$ and $B$ alternate will be energetically favoured.

Obviously the above configurations both belong to ordered states. Raising the temperature of the system, the thermal energy $kT$ will tend to randomise the positions of $A$ and $B$, which at some temperature of the system will behave completely disordered. The mathematical model of such systems is the Ising model.
model. It was originally proposed as a model of ferromagnetism. It has been also applied successfully to lattice gases, binary alloys, melting of DNA, econophysics. The Ising model can be solved analytically in 1 and 2 dimensions. No analytical solution in 3 dimensions is known so far. In one dimension it does not exhibit a phase transition, but in two and three it does.

We now want to apply the M(RT)^2 algorithm to the Ising model. The Ising model is a simple model of magnetism and is one of the most studied models in physics. It is known from experiment that the magnetisation of a permanent magnet diminishes in strength as the magnet is heated, i.e., temperature is increased. Above a certain temperature, called the Curie point, the magnetisation disappears completely. The Ising model is a simple physical model to explain this phenomenon from a microscopic point of view. Historically, the model was first proposed by Lenz in 1920 and it was investigated by his student Ernst Ising [1925], and usually bears his name in the literature. The Ising model is the standard model to study phase transitions. As we will see analytical solutions exist only in 1 and 2 dimensions. In 3 dimensions it has to be studied numerically. The fact that it can be easily simulated is probably one of the reasons of its great popularity. The history of the Ising model has been reviewed in Ref. Brush [1967].

9.2.1 The Model

The microscopic Ising model is based upon the observation that the atoms of a magnetic substance are themselves tiny small magnets. The spontaneous magnetisation in the bulk of the substance is explained as the result of the alignment of the “atomic” magnets, usually called “spins”, as a consequence of their mutual interaction. The assumptions at the basis of the Ising model are the following ones: (1) the spins are arranged on a regular lattice; (2) each spin can point only in one of two directions, “up” and “down”; (3) there exists an interaction energy −J between two neighbouring spins which point in the same direction, and an energy J if they point in opposite directions. It is clear that at temperature T = 0 for positive J, the Ising model will be in a configuration (of lowest energy) with all spins aligned. This alignment persist, at least partially, up to some nonzero but finite temperature, the Curie point. We are now in the position to specify mathematically the Ising model.

We consider a solid of N identical atoms arranged on a regular lattice. Each atom has a net electron spin \( \vec{S} \) and an associated intrinsic magnetic moment \( \vec{\mu} \)

\[
\vec{\mu} = g \mu_0 \vec{S},
\]

where \( \mu_0 \) is the Bohr magneton and \( g \) is a factor of order O(1). In the presence of an external magnetic field \( H_0 \) in the z-direction the Hamiltonian describing the interaction between the magnetic field and the atoms reads

\[
H = -g \mu_0 \sum_{j=1}^{N} \vec{S}_j \cdot \vec{H}_0 = -g \mu_0 \sum_{j=1}^{N} S_j H_0,
\]

where the index \( j \) runs over all atoms. Of course, atoms interact also with neighbouring atoms. The magnetic dipole–dipole interaction is generally too small to explain ferromagnetism. The dominant interaction is the exchange interaction between neighbouring spins, which is a direct consequence of the Pauli exclusion principle. Sloppy speaking, two electrons with parallel spins in neighbouring atoms can not come arbitrarily near since they occupy the same state. In the opposite case, two electrons with anti-parallel spins are in different states, and the Pauli principle does not forbid their approaching. This electrostatic interaction between neighbouring atoms depends on the relative orientation of the spins. The exchange interaction between two atoms \( i \) and \( j \) can be written as

\[
H_{ij} = -J S_i S_j,
\]

where \( J \) measures the strength of the interaction. If \( J > 0 \), the interaction energy is lower if the spins are parallel. The state of minimal energy is one which favours configurations in which all the spins are parallel. This is the situation leading to ferromagnetism. If \( J < 0 \) it is energetically favourable for the spins to be antiparallel.
Since $J$ is a function of the overlapping of the electron wave functions, it is a function of the distance between the atoms and decreases rapidly. It is therefore safe to assume that atoms interact only with nearest neighbours and to regard consequently $J$ as constant.

According to the assumption of the Ising model, we consider only components of the spin in the $z$-direction and we write the exchange interaction as

$$H_{ij} = -J s_i s_k.$$ 

In the above formula we have adopted the notation $s_i = \pm 1$, where $s_i$ is the $z$-component of the spin at lattice site $i$. $s_i = +1$ is an "up" spin, and $s_i = -1$ is a "down" spin. The total Hamiltonian $H'$ of the exchange interaction reads

$$H' = J \sum_{nn(i)} s_i s_k,$$

where $nn()$ denotes a sum only over nearest neighbours. Finally, the Ising Hamiltonian is

$$H = -J \sum_{nn(i)} s_i s_j - H_0 \sum_i s_i,$$

where we have absorbed the factors $\mu_0$ and $g$ into $H_0$.

The problem is now to calculate the thermodynamic functions of the system as, e.g., the mean magnetic moment $\bar{M}$ as a function of the temperature and of the external magnetic field. According to the principles of statistical mechanics, the thermodynamics of the system can be derived from the partition function

$$Z = \sum_{\text{configurations}} \exp\left(-\frac{H}{kT}\right).$$

The summation has to be taken over all possible spin configurations. Mathematically, the point is to find a closed expression for $Z$. If there is a Curie point it will manifest itself in this expression as a point of non-analyticity in the variable $T$.

In 1925 Ising solved this problem for the 1 dimensional case and found no Curie point. The existence of a spontaneous magnetisation in 2 and 3 dimensions was established by Peierls in 1936, and in 1944 Onsager published the exact solution for the 2 dimensional case ($H_0 = 0$) Onsager [1944]. The analytical solution of the Ising model in 3 dimensions and of the 2 dimensional Ising model with external field remain unknown. The thermodynamic properties can be studied by computer simulation as we will see shortly. We will not present here the exact solution of the ising model in 1 and 2 dimensions and refer the reader to the original literature and to excellent textbooks. Here we prefer, in order to review some of the basic features of the physics of phase transitions to recall the mean field approach of Pierre Weiss.

### 9.2.2 The Mean Field Theory

In order to understand better the physics behind the Ising model, we want to treat it in the most simple approximation showing phase transitions and some of their characteristic features. The basic idea of mean field approximations is the following one: Pick up a specific spin, say $j$. The interaction of this spin with the external magnetic field and the internal field due to the other spins in the lattice is described by the Hamiltonian

$$H_j = -g \mu_0 H_0 s_j - J s_j \sum_{k=nn(j)} s_k.$$ 

In order to simplify the second term in the above equation we assume that each spin interacts with the same internal field due to the other spins in the lattice. Thus, we replace the sum over all neighbors by the mean value

$$J \left\langle \sum_{k=nn(j)} s_k \right\rangle \equiv g \mu_0 H_0 = J q m,$$
where $H_m$ is a parameter having the dimensions of the magnetic field. $m$ denotes the mean magnetisation per spin and $q$ is the number of nearest neighbours. The mean field Hamiltonian reads

$$
\langle H_1 \rangle = -g\mu_0H_0 s_j - 2J_s q m = -g\mu_0H_0 + 2J q m s_j.
$$

We have reduced the original $N$–spin Hamiltonian to an effective 1 spin Hamiltonian. Accordingly, the partition function for 1 spin reads

$$
Z_1 = \sum_{s_1=\pm 1} \exp(-\beta s_1 \langle H_1 \rangle) = 2 \cosh(\beta q m + H).
$$

We know from statistical mechanics, that the free energy per spin is given by

$$
f = -\frac{1}{\beta} \ln Z_1 = -kT \ln(2 \cosh(\beta q m + H)).
$$

So that the mean magnetization per spin is

$$
m = -\frac{\partial f}{\partial H} = \tanh(\beta q m + H),
$$

(Eq. 9.2)

Evidently the above equation is a self–consistent equation for $m$. The solution to this equation can be found at best graphically. The correct value of $m$ is the crossing point of the plots of the two sides of the equation.

**Figure 9.1:** The graphical solution of Eq. (9.2).

Let us look at the case $H = 0$. Obviously, $m = 0$ is a solution and it corresponds to a disordered paramagnetic state. However, it is more interesting to look at the possibility of spontaneous magnetisation and to this end we set $H = 0$. The slope of the function $\tanh(\beta q m)$ varies monotonically from the initial value $\beta q J$ to zero. Since the slope of the function $m$ is unity a nonzero solution to Eq. (9.2) exists for $\beta q J \geq 1$ or $kT \leq q J$. The critical temperature $T_c$ separating the $m = 0$ solution from the $m \neq 0$ solution is given by

$$
kT_c = Jq.
$$

For $H = 0$ the magnetisation is small near $T = T_c$, and we can expand the $\tanh(\beta q m)$ term in Eq. (9.2) to obtain

$$
m = \beta q J m - \frac{1}{3}(\beta q J m)^3 + \ldots .
$$

(Eq. 9.3)

Again we find the disordered paramagnetic state $m = 0$ as a solution to Eq. (9.3). The second solution is

$$
m = \frac{\sqrt{3}}{(\beta q J)^{3/2}}(\beta q J - 1)^{1/2},
$$

(Eq. 9.4)

corresponding to the ordered ferromagnetic state. By substituting the two solutions for $m$ into the expression for the mean field mean energy at $H = 0$ it is easy to verify, that the $m = 0$ solution provides a lower free energy for $T > T_c$ and conversely the $m \neq 0$ solution at $T < T_c$. We can now set $kT_c = q J$ into the expression (9.4) and we find for $T$ near $T_c$ that the magnetization vanishes as

$$
m \approx \left( \frac{T_c - T}{T_c} \right)^{1/2}.
$$

Another quantity of interest near $T_c$ is the zero–field susceptibility (per spin) $\chi$

$$
\chi = \lim_{H \to 0} \frac{\partial m}{\partial H}.
$$
With the help of Eq. (9.2) we obtain
\[ \chi = \frac{\beta(1 - \tanh^2 \beta q J_m)}{1 - \beta q J (1 - \tanh^2 (\beta q J_m))}. \]

For \( T \) near \( T_c \) we find
\[ \chi \approx \frac{1}{T - T_c}, \]
which is the famous Curie–Weiss law.

Let us consider shortly the case \( H \neq 0 \). Expanding Eq. (9.2) to third order in \( H \) with \( \beta = \beta_c = 1/(qJ) \) the magnetisation at \( T_c \) is found to be
\[ m = m + \beta_c H - \frac{1}{3} (m + \beta_c H)^3 + \ldots. \]

If \( m \) and \( H \) are so small that
\[ \beta_c H \ll m \]
then
\[ 0 = \beta \alpha_c H - \frac{1}{3} m^3 \]
and hence
\[ m = (3\beta_c H)^{1/3}. \]

Further quantities of interest are the mean energy and the heat capacity. The mean energy per spin is simple the average value of the interaction energy
\[ \langle \epsilon \rangle = \frac{1}{2} q J m^2. \quad (9.5) \]

For \( T > T_c \) we know that the magnetisation vanishes, \( m = 0 \), and hence the mean energy as well as the heat capacity vanish for all temperatures \( T > T_c \). For \( T > T_c \) the mean energy is found by inserting (9.3) into (9.5)
\[ \langle \epsilon \rangle = -\frac{1}{2} q J [\tanh(\beta(q J m + H))]^2. \]
The specific heat \( C \) for \( T < T_c \) is now easily evaluated with the help of the equation
\[ C = \frac{\partial \langle \epsilon \rangle}{\partial T}. \]

For \( T \to T_c \) from below \( C \to 3k/2 \). Hence mean field theory predicts a jump in the specific heat at \( T = T_c \).

Remarkably mean field theory predicts a critical point. The critical point is characterized by an order parameter. Here it is the magnetisation. Unfortunately, the theory is too simple. For example it predicts a critical temperature which does not depend upon the dimension of the system but only on the number of nearest neighbours. The prediction of a phase transition for the 1d Ising model is qualitatively incorrect. The mean field prediction for \( T_c \) in a 2d square lattice is \( kT_c/(Jm) = 4 \) whereas the exact result is
\[ \frac{kT_c}{J} = \frac{2}{\ln(1 + \sqrt{2})} \approx 2.269. \]

However, mean field theory predicts correctly that many physical quantities exhibit a power law behaviour near the critical point
\[ m(T) \approx (T_c - T)^{\beta} \quad (T < T_c) \]
\[ \chi(T) \approx \left| T - T_c \right|^{-\gamma} \]
\[ m(T = T_c) = \approx H^{1/\beta}. \]
Table 9.1 Comparison of the critical exponents for the 2 and 3 dimensional Ising model with mean field theory.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Exponent</th>
<th>d=2 (exact)</th>
<th>d=3 (sim)</th>
<th>MFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>specific heat</td>
<td>$\alpha$</td>
<td>0 (log)</td>
<td>0.113</td>
<td>0 (jump)</td>
</tr>
<tr>
<td>order parameter</td>
<td>$\beta$</td>
<td>1/8</td>
<td>0.324</td>
<td>1/2</td>
</tr>
<tr>
<td>susceptibility</td>
<td>$\gamma$</td>
<td>7/4</td>
<td>1.238</td>
<td>1</td>
</tr>
<tr>
<td>$M \approx H^{-1/8}$</td>
<td>$\delta$</td>
<td>15</td>
<td>4.82</td>
<td>3</td>
</tr>
<tr>
<td>Correlation length</td>
<td>$\nu$</td>
<td>1</td>
<td>0.629(4)</td>
<td>1/2</td>
</tr>
<tr>
<td>$c(r)$ at $T = T_c$</td>
<td>$\eta$</td>
<td>1/4</td>
<td>0.031(5)</td>
<td>0</td>
</tr>
</tbody>
</table>

The quantities $\beta$, $\gamma$ and $\delta$ are called critical exponents. A comparison of the mean field exponents with the analytical values in 2 dimensions and simulation results in 3 dimensions are found in table 9.1.

Additional information about the phase transition can be obtained from the behaviour of the spin correlation function $c_{ij}$

$$c_{ij} = \langle s_i s_j \rangle - m^2,$$

where we have assumed that $\langle s_i \rangle = \langle s_j \rangle = m$. It can be shown, that the spin correlation function is related to the susceptibility

$$\chi = \frac{1}{NkT} \sum_{i,j=1}^{N} (\langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle),$$

$$= \frac{1}{NkT} \sum_{i,j=1}^{N} c_{ij},$$

$$= \frac{1}{kT} \sum_{j=2}^{N} c_{1,j},$$

since all lattice sites are equivalent. Near $T_c$ the susceptibility $\chi$ diverges, and hence the number of correlated spins must increase as $(T_c - T)/T_c \to 0$. In fact the neighbourhood of the critical point is characterized by long–range correlations. To characterize these correlations we write $c_{ij}$ as

$$c(r) = \frac{\exp(-r/\xi)}{r^{d-2+\eta}}.$$

$r$ is the distance between the two spins. $\xi$ is called the correlation length and increases as $T \to T_c$, as

$$\xi \approx \left( \frac{T - T_c}{T_c} \right)^{\nu}.$$

At $T = T_c$, $\xi$ is infinite and $c(r)$ decays as a power law characterized by the critical exponent $\nu$

$$c(r) \approx \frac{1}{r^{d-2+\eta}}.$$

As a last comment let us remark that the critical exponents $\alpha$, $\beta$, $\gamma$, $\delta$, and $\eta$ are not independent, but obey some scaling relations. Gould et al. [1995]

9.3 The Monte Carlo Simulation

In the preceeding section we have convinced ourselves of the necessity of an efficient numerical algorithm for the Ising model. We will now apply the M(RT)$^2$ algorithm to the Ising model in order to compute
numerically the expectation values we are interested in, e.g., the mean energy and th mean magnetisation. The necessity of a good algorithm based on the idea of importance sampling is made evident by the huge number of possible configurations. Consider for example the the 2 dimensional Ising model on a $50 \times 50$ square lattice. The number of terms to be summed up in order to obtain the state function is $2^{50 \times 50} \approx 2^{2500} \approx 10^{753}$. It is therefore impossible to sum up equally distributed random configurations.

Of course, we will have to sample configurations which are distributed according to the Boltzmann distribution, i.e.,

$$P_{eq}[[\sigma_i]] \approx \exp(-\beta H[[\sigma_i]]) .$$

In order to extend the idea of the random walk in the space of spin configurations we have to construct the conditional transition probability $W(\{\sigma'_j\}|\{\sigma_i\})$ according to the prescription of the $M(RT)^2$ algorithm

$$w \rightarrow \{\sigma_i\} \rightarrow \{\sigma'_i\} \rightarrow \{\sigma''_i\} \rightarrow \cdots .$$

Of course, $W$ will have to satisfy the detailed balance condition

$$W(\{\sigma'_j\}|\{\sigma_i\})P_{eq}[[\sigma_i]] = W(\{\sigma_i\}|\{\sigma'_j\})P_{eq}[[\sigma'_j]] .$$

The Markov chain, i.e., the random walk, is realized through local updates of single spins. Formally, we can write for the transition probability for flipping one spin and thus for "jumping" from the configuration $X = (\ldots, s_{\alpha}, \ldots)$ to the next configuration $X' = (\ldots, -s_{\alpha}, \ldots)$

$$T(\{\sigma_i\}|\{\sigma'_i\}) = \sum_{i,j=1}^N f_{ij} \delta(\sigma'_{ij} + \sigma_{ij}) ,$$

where $\delta(\sigma'_{ij} + \sigma_{ij})$ describes the flipping of spin $\sigma_{ij}$: $\sigma'_{ij} \rightarrow -\sigma_{ij}$. Of course $f_{ij}$ is constant and $f_{ij} = 1/N^2$. Since $T$ is essentially constant $T(X'|X) = T(Y'|X)$ and the quantity $q(X'|Y)$, which as we remember enters the expression for the acceptance of the move is simply given by the ratios of the $P_{eq}$

$$q(X'|X) = \frac{P_{eq}[X]}{P_{eq}[X']} .$$

Denoting by $E(X)$ and by $E(X')$ the the energy of the configuration before and after the spin flip we have

$$q(X'|X) = \exp(-\beta(E(X) - E(X'))) .$$

The new configuration can be accepted with probability

$$A(X'|X) = \min(1, q(X'|X)) = \begin{cases} 
\exp(-\beta(E(X) - E(X'))): & E(X') > E(X) \\
1: & E(X') < E(X) .
\end{cases}$$

The simplicity of the $M(RT)^2$ algorithm is made evident by formulating it locally. It is clear from the expression of $q$ that we need only know the energy difference $E(X') - E(X)$. Assume that the spin $\sigma_v$ has been flipped $\sigma_v \rightarrow -\sigma_v$ in changing the configuration from $X$ to $X'$. The index $v$ denotes a pair of indices $i,j$. So we can write for the energy difference

$$\Delta E_v = E(-s_v) - E(+s_v) ,$$

where all other $(n-1)$ spins remain unchanged. We consider first the case of vanishing external magnetic field ($H = 0$). We have

$$E(+s_v) = -J_{sv} u_v + \cdots$$

$$E(-s_v) = +J_{sv} u_v + \cdots ,$$
where
\[ u_v = \sum_{m(n)} s_{mn}. \]

The terms indicated by \(*\) do not contain \( s_v \) and are equal. Thus the energy difference can be written as
\[ \Delta E_v = E(-s_v) - E(+s_v) = 2J s_v u_v. \]

Hence we can write \( q \) as
\[ q(-s_v | s_v) = \exp(-\beta Js_v u_v). \]

In two dimensions we have of course 4 nearest neighbours and \( U_v \) can assume only the values 0, \( \pm 2, \pm 4 \). Hence \( s_v U_v \) and consequently \( q \) can assume only 10 different values, which, in principle, can be computed in advance.

Summarising the M(RT)\(^2\) algorithm schematically reads:
1. Choose a spin \( s_v \).
2. Calculate \( u_v, \Delta E_v = 2J s_v u_v \), and \( \eta = \beta \Delta E_v \).
3. If \( \eta < 0 \), then flip the spin \( s_v \rightarrow -s_v \); go back to 1.
4. If \( \eta > 0 \), choose a random number \( r \) equally distributed in \([0,1]\). If \( r < \exp(-\eta) \) flip the spin \( s_v \rightarrow -s_v \), else do not flip the spin; go back to 1.

We have seen that \( A \) can also be chosen to be
\[ A(Y | X) = \frac{q(Y | X)}{1 + q(Y | X)}. \]

For the Ising model this means
\[
A(Y | X) = \frac{\exp(-\beta \Delta E)}{1 + \exp(-\beta \Delta E)} = \frac{1}{2}[1 - \tanh(\beta J s_v)],
\]

This is the so-called Glauber version of the M(RT)\(^2\) algorithm, which explicitly reads:
1. Choose a spin \( s_v \).
2. Calculate \( u_v, \Delta E_v = 2J s_v u_v \), and \( \eta = \beta \Delta E_v \).
3. Choose a random number \( r \) equally distributed in \([0,1]\). If
\[
r < \frac{1}{1 + \exp(\eta)}
\]

perform the flip \( s_v \rightarrow -s_v \), otherwise \( s_v \) is unchanged; go back to 1.

In principle, the choice of the spin \( s_v \) in step 1. of both versions of the algorithm should occur at random. In practice, in order to save CPU time, one sweeps across the lattice.

### 9.3.1 The Code

Listing und Beschreibung.

**Figure 9.2:** Configurations of the two dimensional Ising model on a 100 \( \times \) 100 lattice at \( \beta / \beta_c = 0.5, 0.7, 0.9, 0.95, 0.98 \). Notice the growth of correlations from high temperatures to the critical region.
Figure 9.3: Magnetisation as a function of the reduced temperature $kT/2J$ for $L = 40, L = 50$ and $L = 100$.

Figure 9.4: Magnetic susceptibility.

9.4 Data Analysis

This section is devoted to the analysis of the data generated with the help of the Monte Carlo simulation of the Ising model. The thermal quantities of interest are, as we know, the mean energy $<E>$ and the heat capacity $C$. It is clear that we will calculate the mean energy in the simulation as the average of the energy of each configuration of the lattice, i.e.,

$$<E> = \langle -J \sum_{m(i,j)} s_i s_j - H_0 \sum_i s_i \rangle.$$

A straightforward way to calculate the heat capacity at constant external magnetic field is from its definition $C = \partial <E>/\partial T$. However, it is better to determine $C$ from the statistical fluctuations of the energy in the canonical ensemble. To recall this formula we write for the heat capacity at constant volume

$$C_V = \frac{1}{kT^2} \frac{\partial <E>}{\partial \beta}.$$

From the definition of $<E>$

$$<E> = \frac{1}{Z} \sum_s E_s \exp(-\beta E_s),$$

where the index $s$ runs over all $M$ accessible micro states of the system, we have

$$<E> = -\frac{\partial}{\partial \beta} \ln Z,$$

Hence, we find

$$\frac{\partial <E>}{\partial \beta} = -\frac{1}{Z^2} \frac{\partial Z}{\partial \beta} \sum_s E_s \exp(-\beta E_s) - \frac{1}{Z} \sum_s E_s^2 \exp(-\beta E_s),$$

which finally can be written in the desired form for the specific heat

$$C = \frac{1}{kT^2} \left( <E^2> - <E>^2 \right).$$

Other quantities of interest are the mean magnetisation $<M>$ and the zero field magnetic susceptibility $\chi$. The former is calculated from its definition

$$M = \sum_{i=1}^N s_i$$

and the latter from the fluctuations of the magnetisation

$$\chi = \frac{1}{kT} \left( <M^2> - <M>^2 \right).$$

Figure 9.5: The energy as a function of $T$. 

9.4. DATA ANALYSIS
9.4.1 Estimation of Errors

The Monte Carlo as well as the Molecular Dynamics simulation of physical systems produce raw data, e.g. the energy or the magnetisation of the system, in form of a finite time series of correlated data. Typically, stationary states, i.e. the equilibrium state, are investigated and the first step in the data analysis is the estimation of the mean values which are computed as time averages. Since the times is finite the mean values do fluctuate: they are random variables as well. Performing the simulation another time will lead to different estimate. So the second step of the data analysis is the estimation of the variance of finite time averages. We now discuss in some detail the analysis of the data Flyvbjerg und Petersen [1989]; H.Flyvbjerg [1998].

Preliminary Considerations

Let us consider a Monte Carlo simulation in which some quantity, say \( x \) is computed. The simulation is performed over \( n \) steps and \( x_1, x_2, \ldots, x_n \) denote the result of the \( n \) consecutive measurements of the fluctuating quantity \( x \). In order to be precise we denote by \( \langle \cdots \rangle \) the expectation value with respect to the exact, in general unknown, probability distribution \( p(x) \)

\[
\langle x \rangle = \int dx x p(x),
\]

By \( \bar{x} \) we denote the average over the set \( \{ x_1, x_2, \ldots, x_n \} \)

\[
\bar{x} \equiv \frac{1}{n} \sum_{i=1}^{n} x_i,
\]

which is the quantity we compute in practice. As is usual we assume ergodicity: the ensemble average \( \langle x \rangle \) is equal to the "time" average \( \lim_{n \to \infty} \bar{x} \). As we know already, we estimate the expectation value \( \mu \equiv \langle x \rangle \) by the average value

\[
m \equiv \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \quad \text{(9.6)}
\]

What we now need is the estimator for the variance of \( m \),

\[
\sigma^2(m) \equiv \langle m^2 \rangle - \langle m \rangle^2. \quad \text{(9.7)}
\]

Inserting, Eq. (9.6) into Eq. (9.7) we find that

\[
\sigma^2(m) = \frac{1}{n^2} \sum_{i,j} \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle
\]

\[
= \frac{1}{n^2} \sum_{i,j} \gamma_{ij} \quad \text{(9.8)}
\]

where we have introduced the correlation function \( \gamma_{ij} = \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \). Using the invariance of the correlation function under time translations we define

\[
\gamma_t \equiv \gamma_{ij}; \quad t = |i - j|
\]
and rewrite Eq. (9.8) in the form
\[ \sigma^2(m) = \frac{1}{n} \left[ \gamma_0 + 2 \sum_{t=1}^{n-1} \left( 1 - \frac{t}{n} \right) \gamma_t \right]. \] (9.9)

The above equation is often used as an estimate for \( \sigma^2(m) \) inserting an estimate for \( \gamma_t \) at the appropriate place. Doing so requires some care since the most obvious estimator \( c_t \) for \( \gamma_t \)
\[ c_t = \frac{1}{n-t} \sum_{k=1}^{n-t} (x_k - \bar{x})(x_{k+t} - \bar{x}) \] (9.10)
is a biased estimator. It is easy to check that the expectation value of \( c_t \) is not \( \gamma_t \), but
\[ \langle c_t \rangle = \gamma_t - \sigma^2(m) + \Delta_t, \] (9.11)
where \( \Delta_t \) depends on the correlation functions \( \gamma_{i,j} \). Although it is possible to construct with the help of some approximations of the above expression unbiased estimators based on the correlation functions we prefer to describe here another way for the estimation of \( \sigma^2(m) \).

The "Blocking" Method

The method we want to describe now is to be preferred from a computational point of view and because it gives information about the quality of the estimate of \( \sigma^2(m) \). The method is based on the repeated "blocking" of data and works in the following way.

The data \( x_1, \ldots, x_n \) are transformed into a new data set \( x'_1, \ldots, x'_{n'} \) which is half as large
\[ x'_t = \frac{1}{2}(x_{2t-1} + x_{2t}), \] (9.12)
\[ n' = \frac{n}{2}. \] (9.13)

For the new "blocked" data set we define a new average value as
\[ m' = \bar{x}' = \sum_{t=1}^{n'} x'_t. \]

Obviously, the mean averages \( m \) and \( m' \) for the original and for the "blocked" data sets are equal
\[ m' = m. \]

Accordingly, we define "blocked" correlation functions \( \gamma'_{i,j} \) and \( \gamma''_{i,j} \) for the new primed data set
\[ \gamma'_{i,j} = \langle x'_i x'_j \rangle - \langle x'_i \rangle \langle x'_j \rangle. \]

It is easy to check that \( \sigma^2(m) \) is invariant under the blocking transformation. To this end we calculate first
\[ \gamma''_t = \begin{cases} \gamma_0/2 + \gamma_1/2 & \text{for } t = 0, \\ 1/2\gamma_{2t-1} + 1/2\gamma_{2t} + 1/2\gamma_{2t+1} & \text{for } t > 0. \end{cases} \] (9.14)

and then
\[ \sigma^2(m') = \frac{1}{n'^2} \sum_{i,j=1}^{n'} \gamma''_{i,j} = \sigma^2(m). \]

The invariance of the blocking transformation implies that no information is lost in considering the blocked data set. From Eq. (9.9) we know that
\[ \sigma^2(m) \geq \frac{\gamma_0}{n}. \]
and from (9.13) and (9.14) it follows that \( \gamma_0/n \) increases every time the "blocking" transformation is applied, unless \( \gamma_1 = 0 \). In the latter case \( \gamma_0/n \) is invariant. The idea of the blocking method is very simple. If the computed \( \sigma^2(m') \) is not the same as the original \( \sigma(m) \) we have to apply again the blocking transformation to the data set until the estimated \( \sigma^2(m') \) is approximately the same as that calculated from the previous data set.

It is easy to show that

\[
(\frac{\gamma_i}{n})_{i=0,1,\ldots} \approx (\delta_i,0)_{i=0,1,\ldots}
\]

is a fixed point of the linear blocking transformation. At the fixed point \( \gamma_i = 0 \) for \( t > 0 \) and hence

\[
\sigma^2(m) = \gamma_0/n.
\]

In order to estimate \( \sigma^2(m) \) we have to replace \( \gamma_0 \) in the above expression by its estimate \( c_0 \) which is defined in (9.11). Because of Eq. (9.11) and of \( \Delta_0 = 0 \) solving for \( \sigma(m) \) we obtain

\[
\sigma^2(m) \geq \frac{\langle c_0 \rangle}{n-1}.
\]

Of course, the identity is satisfied at the fixed point.

In practice one proceeds as follows. Starting from the original data set \( x_1, \ldots, x_n \) one estimates \( \sigma^2(m) \) using \( c_t \) as an estimate for \( \gamma_t \), i.e. one calculates

\[
\frac{c_0}{n-1}
\]

as an estimate for

\[
\frac{\langle c_0 \rangle}{n-1} \leq \sigma^2(m).
\]

Then, the blocking transformation is applied to the data set and

\[
\frac{c'_0}{(n-1)}
\]

is computed. The process is repeated until \( n' = 2 \). The sequence of values obtained for \( c_0/(n-1) \) will increase until it remains constant within the fluctuations. This constant value is the estimate for \( \sigma^2(m) \).

Figure 9.7: Estimates for \( \sigma^2(m) \) obtained with the blocking method.

### 9.4.2 Finite Size Effects


Another point which he have to consider in Monte Carlo simulations of phase transitions is that the correlation length \( \xi \) can not diverge near the critical point. similarly, also the divergencies in other quantities are rounded and shifted. This fact is illustrated in Fig. (9.4.2) where we plot the specific heat of the 2d Ising model as a function of \( \beta \) for various lattice sizes.

Because of the finiteness of the system near \( T_c \) the role of the correlation length \( \xi \) is taken over by the linear size of the system

\[
\xi(T) \approx L \approx |T - T_c|^{-\nu}.
\]
Figure 9.8: Finite size scaling behaviour of the two dimensional Ising model on $L \times L$ square lattices. (Exact solutions of Ferdinand and Fisher?).

Hence we can write

$$|T - T_c| \approx L^{-1/\nu}$$

and we see that near $T_c$ the usual scaling laws are replaced by

$$m(T) \approx (T_c - T)^\beta \to L^{-\beta/\nu},$$

$$C(T) \approx |T - T_c|^{-\alpha} \to L^{\alpha/\nu},$$

$$\chi(T) \approx |T - T_c|^{-\gamma} \to L^{\gamma/\nu}.$$

In order to estimate the critical exponents one usually fits the above finite-size scaling laws.

### 9.5 The Cluster Algorithm

The simulations of the Ising model have shown that as the temperature is decreased, in the absence of an external field, clusters of spins of both signs tend to expand as we approach the transition. In this situation the generation of new configurations with the help of local algorithms is a difficult task since the system is almost trapped in a subset of phase space. To overcome this difficulty cluster algorithms have been proposed Swendsen und Wang [1987]. The basic idea is to rearrange large blocks of spins instead of individual spins. We will describe here the Swendsen–Wang algorithm and will follow the presentation found in Schnakenberg [1995].

The prescription for the cluster update algorithms is motivated by an appropriate equivalent representation of the state function of the Ising model. As we know

$$Z = \sum_{\{\sigma\}} \exp \left( \beta \sum_{\langle ij\rangle} \sigma_i \sigma_j \right).$$

Using the fact that the product $\sigma_i \sigma_j$ of two Ising spins can assume only the two values $\pm 1$, we find that

$$\exp(\beta \sigma_i \sigma_j) = \exp(\beta) \{1 - p\delta_{\sigma_i,\sigma_j}\},$$

where we have introduced $p = 1 - \exp(-2\beta)$. Thus above sum can be rewritten as Kasteleyn und Fortuin [1969]; Fortuin und Kasteleyn [1972]

$$Z = \sum_{\{\sigma\}} \prod_{\langle ij\rangle} \exp(\beta) \{1 - p\delta_{\sigma_i,\sigma_j}\}.$$

With the help of the trivial identity

$$a + b = \sum_{n=0}^{1} (a\delta_{n,0} + b\delta_{n,1}),$$

we can write

$$Z = \sum_{\{\sigma\}} \sum_{n_{ij}} \prod_{\langle ij\rangle} \exp(\beta) \{1 - p\delta_{n_{ij},0} + p\delta_{\sigma_i,\sigma_j}\delta_{n_{ij},1}\},$$

where we have introduced the bond variable $n_{ij}$ which can take the values $n_{ij} = 0$ or 1, corresponding to deleted or active bonds. The idea of the cluster algorithm consists in identifying clusters of spins with parallel spins that are connected by active bonds and by updating all the spins in cluster at one time.
As we just mentioned each spin configuration contains connected domains of parallel spins, i.e., nearest neighbour spins show in the same direction. These domains play a central role in the Swendsen–Wang algorithm:

(i) The starting point is a configuration, say \( K \). For each nearest neighbour pair within a domain of parallel spins in \( K \) we establish with probability

\[
p^b = 1 - \exp(-2\beta J)
\]

a bond. A cluster is a region of the lattice where all lattice sites are connected by bonds. In this way the spin configuration is converted into a cluster configuration.

(ii) Each cluster is assigned a new spin value, \( s = 1 \) with probability 1/2 or \( s = -1 \) with probability 1/2.

(iii) The new configuration \( K' \) is obtained by reassigning the value of the spins at each site in a cluster. All sites in a given cluster assume the same spin assigned in 2. to the given cluster.

It is clear that if the original configuration contains sizeable cluster the new configuration \( K' \) will be very different from it. With the help of a local algorithm the new configuration would have been obtained only after a very long time. We now have to check that the above algorithm leads to the correct simulation of the equilibrium properties.

To this end we consider the transitions between two spin configurations \( K \) and \( K' \) over a cluster configuration \( C \). It follows from the algorithm that if the transition \( K \rightarrow K' \) is possible also the transition \( K' \rightarrow K \) is allowed, and hence

\[ K \leftrightarrow C \leftrightarrow K'. \]

Since step (i) in the above algorithm and step (ii) and (iii) are independent we have

\[
\begin{align*}
W(K'|K) & = W(K'|C)W(C|K), \\
W(K|K') & = W(K|C)W(C|K').
\end{align*}
\]  

(9.15)

The spins \( s = \pm 1 \) are assigned with equal probability to the clusters, so it is obvious that we have

\[ W(K|C) = W(K'|C), \]

and hence it follows from Eq. (9.15)

\[
\frac{W(K'|K)}{W(K|K')} = \frac{W(C|K)}{W(C|K')}.
\]

Let us now \( r \) be the number of nearest neighbours pairs, which do have in \( K \) parallel spins, but which in \( C \) belong to different clusters. In the step 1, \( K \rightarrow C \) bond have not been established between these pairs with the probability \( 1 - p_b \). Correspondingly, \( r' \) denotes the same number in \( K' \). Then we have

\[
\frac{\text{frac} W(C|K)W(C|K')}{W(C|K)} = (1 - p_b)^{r'-r} = \exp[-2\beta(r - r')],
\]

(9.16)

Let \( q \) denote the number of nearest neighbours pairs in \( K \) having different spins, correspondingly \( q' \) in \( K' \), and \( c \) the number of nearest neighbour pairs, which are separated in \( C \) by cluster borders. Then we have

\[ r + q = r' + q' = c \]

and hence

\[ r - r' = -(q - q'). \]

Thus Eq. (9.16) may be written as

\[
\frac{\text{frac} W(C|K)W(C|K')}{W(C|K)} = \exp[2\beta(q - q')].
\]

The last expression can be rewritten in an appropriate way by considering that the energies \( E(K) \) and \( E(K') \) of the configurations \( K \) and \( K' \) can be written as

\[
\begin{align*}
E(K) & = -dL^dJ + 2qJ, \\
E(K') & = -dL^dJ + 2q'J.
\end{align*}
\]
9.5. THE CLUSTER ALGORITHM

and hence
\[ E(K) - E(K') = 2(q - q') J, \]

so that finally
\[ \frac{\text{d}W(K'|K)W(K|K')}{(K|K')} = \exp[\beta(E(K) - E(K'))]. \]

We immediately recognise again the detailed balance condition. This completes the proof of the Cluster algorithms.

Prior to presenting the Java code of the Cluster algorithm for the 2 dimensional Ising model it may be helpful to give a more detailed description of the practical implementation of the three steps (i), (ii), and (iii) introduced above. The experience with the Hoshen–Kopelman algorithm for percolation problems will turn out to be very useful. Again we follow Ref. Schnakenberg [1995].

First let us remark that we will make use of helical boundary conditions. The spin configuration will be represented in a linear array denoted by \( s_v \), where the index \( v \) may assume the values \( v = 1, 2, \ldots, N = L^2 \), where \( L \) denotes the number of spins in one dimension. The index \( v \) is easily obtained from the Cartesian coordinates \( i, j \)

\[
\begin{align*}
v &= i + jL; & \text{for } d &= 2, \\
v &= i + jL + kL^2; & \text{for } d &= 3.
\end{align*}
\]

The nearest neighbours of the lattice site \( v \) are easily determined: in 2 dimensions the left and the right neighbour are at \( v \pm 1 \) and the upper and bottom neighbours are at \( v \pm L \); in 3 dimensions, we have furthermore a neighbour in front and on the back at \( v = \pm L^2 \). Counting the lattice sites in this way we have essentially fixed the boundary conditions: If the lattice site \( v \) is on the right border of the lattice its right neighbour will be in the line below on the left border; and so on. We need only a prescription for the case that the numerical value of \( \mu = v \pm 1, \mu = v \pm L, \mu = v \pm L^2 \) is negative or greater then \( N = L^d \):

\[
\begin{align*}
\mu \leq 0 : & \mu \rightarrow \mu + L^d \\
\mu > L^d : & \mu \rightarrow \mu - L^d
\end{align*}
\]

The helical boundary conditions are depicted in Fig. 9.9.

**Figure 9.9:** Helical boundary conditions

The initial configuration of the spins \( s_v \) will be saved in a linear array \( S(v) \). We will need to further arrays, \( A(v) \) with \( 1 \leq v \leq N \) and \( B(c) \) with \( c = 1, 2, \ldots, C \leq N \). For a given lattice site \( v \) we have to distinguish whether a bond to a nearest neighbour \( \mu \) has been established or not. In the first case \( s_v = s_\mu \). In the second case, either \( s_v \neq s_\nu \) or \( s_v = s_\mu \) and the bond was not established with probability \( 1 - p_b \).

**Sweep (i):** for \( v = 1, 2, \ldots, N \)

1. No bonds are established to the nearest neighbours \( \mu = v - 1 \) and \( \mu = v - L \) of lattice site \( v \). With lattice site \( v \) begins a new cluster with name \( c \), i.e., we set \( A(v) = c \). Since no bonds are established starting from \( v \) the cluster \( c \) is unbound and we set \( B(c) = 0 \).

2. A bond is established to exactly one of the two neighbours \( \mu = v - 1 \) and \( \mu = v - L \) of lattice site \( v \). The nearest neighbour spin at \( \mu \) belongs to cluster \( c' \). Hence also cell \( v \) is ascribed to cluster \( c' \), i.e., \( A(v) = c' \); the \( B \) array is unchanged.

3. A bond is established to both nearest neighbours \( \mu_1 = v - 1 \) and \( \mu_2 = v - L \) of lattice site \( v \) and cell \( \mu_1 \) and \( \mu_2 \) belong to the same cluster \( c' \). In this case we proceed as in 2, and set \( A(v) = c' \).

4. A bond is established to both nearest neighbours \( \mu_1 = v - 1 \) and \( \mu_2 = v - L \) of lattice site \( v \) and cell \( \mu_1 \) and \( \mu_2 \) belong to different clusters, say \( A(\mu_1) = c_1 \) and \( A(\mu_2) = c_2 \). We assume that \( c_1 < c_2 \). The clusters \( c_1 \) and \( c_2 \) have to be united. To this end we begin by attributing cell \( v \) to the cluster with the smaller name, \( A(v) = c_1 \). Then cluster \( c_2 \) is attached to cluster \( c_1 \) by putting \( B(c_2) = -c_1 \). If cluster \( c_2 \) was already
attached to another cluster \( c'_2 \), i.e., \( B(c_2) = -c'_2 \) then we have to go back and set \( B(c'_2) = -c_1 \). Negative entries \( B(c') = -c \) indicate that the cluster \( c' \) has been attached to the cluster \( c \).

Because of the helical boundary conditions the following situation may arise. A cell \( v \) in the first row may establish bonds to a cell \( \mu \) in the last row, which has not yet been ascribed to any cluster. In this case we put \( A(v) = A(\mu) = c \).

**Sweep (ii):** for \( c = 1, \ldots, C \), where \( C \) is the largest cluster name.
1. The cluster \( c \) is not attached, \( B(c) = 0 \). Draw a equally distributed random number \( 1 \leq r < 1 \) and put \( B(c) = +1 \) if \( r < 1/2 \), else put \( B(c) = +2 \).
2. The cluster \( c \) is attached, \( B(c) = -c' \neq 0 \). The value \( B(c) = B(c') \) is taken over. If \( c' > c \) as a consequence of further attachments, eventually step 1. has to be repeated for \( B(c') \).

**Sweep (iii):** for \( v = 1, \ldots, N \).
1. A new configuration is generated with the help of the rule \( S(v) = 2B(A(v)) + 3 \).

Figure 9.10: The stages of the Swendsen–Wang algorithm for a 6 \times 6 array with helical boundary conditions. IM WESENЛИЧEN DIE FIGUR AUS MACKEOWN, S.375 MIT ANDEREN RANDBEDINGUNGEN.
Bibliography


Chapter 10

Nonequilibrium Monte-Carlo Methods

10.1 The Description of Irreversible Processes

The statistical mechanics of irreversible processes is concerned with the understanding of the following two observed facts van Kampen [1962]: Consider a collection of similar particles, e.g. atoms or molecules.

(i) On a microscopic level the equations of motion of all individual particles are determined are determined completely by the familiar equations of motions of classical mechanics (Newton’s equations) or of quantum mechanics (Schrödinger’s equation). These equations are symmetric with respect to past and future.

(ii) In a very rough and incomplete way the collection of particles may be described by a small number of macroscopic variables. These variables obey in a selfconsistent way deterministic phenomenological differential equations (the balance equations) which are distinguish between past and future.

The problem is that there seems not to be a rigorous derivation of the macroscopic irreversible equations from the reversible microscopic ones. The task of statistical mechanics of irreversible processes is to build an approximate bridge between the microscopic description and the macroscopic one. In particular, the most interesting question is: Where does the irreversibility get in the description?

It seems to be appropriate to introduce an intermediate level of description between the microscopic and the macroscopic equations. The formal setting for this description is the master equation, which as we already know has the general form

$$\frac{d}{dt} P(J) = \sum_{J'} [W(J|J')P(J') - W(J'|J)P(J)],$$

where $J$ is an index characterizing the different states of the system, $P(J)$ is the probability to find the system in state $J$ at time $t$, and $W(J|J')$ is the conditional probability density per time unit for a transition from state $J'$ to state $J$ to take place. The master equation is an equation for the probability density of the different states. This implies that the evolution of the physical system is described in terms of a Markovian stochastic process. The master equation is a good candidate for the description of irreversible processes because of some of its properties. It is evident that the master equation is not invariant under time reversal. As we will see in the next section the solutions of the master equation tend toward a (fixed) equilibrium distribution.

With the help of the above introduced mesoscopic level of description the transition between the microscopic and the macroscopic level can be performed in two steps:

Step 1: The mesoscopic master equation can be derived from the macroscopic deterministic equations of motions for the constituents of the many-particle system. This is the difficult step since the irreversibility is introduced here.

Step 2: From the master equation for the stochastic process derive the deterministic macroscopic phenomenological equations. In a schematic way we have the situation depicted in Fig. (10.1).

Of course, step 1 is the most difficult one and we will not discuss it further here. There are several excellents books introducing the subject of the derivation of master equations from the reversible micro-
scopical equations of motion and we refer the interested reader to them Prigogine [1962]; Kreuzer [1981];
McLennan [1989]. Here we will only be concerned with the description of irreversible processes with the
help of master equations and in particular with the simulation of master equations describing irreversible
thermodynamic phenomena.
Frage: Habe ich die Bemerkung zum Papier in dem das Wort master Gleichung zum erstem mal
vorkommt schon in Kap, 2 gemacht?

10.2 The Ehrenfest Dog–Flea Model

The fundamental problem of statistical mechanics is to describe how a system reaches equilibrium in an
irreversible way. In this this process the irreversible macroscopic dynamics has to be compatible with the
reversible microscopic one.

In 1907 Paul and Tatiana Ehrenfest have suggested a stochastic model . This so-called "dog–flea" model
(sometimes it is also called the "urn model") is an excellent example for the application of Markov
processes to the investigation of problems in statistical mechanics. The model has been formulated originally
to discuss the meaning of the $H$–theorem in thermodynamics. Here, we follow the discussion in
Jancel [1969]; Kac und Logan [1987].

10.2.1 The Model

We consider $2N$ balls (fleas) numbered from 1 to $2N$. The balls are distributed in two urns (dogs), say
$A$ and $B$. At random we choose an integer between 1 and $2N$ and move the ball whose number has been
drawn from the urn it is in to the other one. The procedure is repeated for an arbitrary number of times $s$.
If initially there are more balls in urn $A$, we expect an approach to a naive equilibrium, in which there are
$N$ balls in each urn. Of course, the situation is more involved because there are fluctuations, i.e., deviations
from the naive equilibrium. These lead to two problems, which can be discussed in this model. Namely,
find the equilibrium distribution of the problem (static problem) and describe the decay of the fluctuations
to the naive equilibrium (dynamic equilibrium).

For notational ease we denote by $n_A(s)$ ($n_B(s)$) the number of balls in urn $A$ ($B$) after $s$ drawings. Of
course we have

$$2N = n_A(s) + n_B(s)$$

and further we introduce

$$2k = n_A(s) - n_B(s)$$

so that

$$n_A = N + k,$$

$$n_B = N - k.$$

Furthermore, it turns out to be useful to introduce $\Delta_s$ as the absolute value of the difference of $n_A$ and $n_B$

$$\Delta_s = |n_A(s) - n_B(s)| = 2|k|.$$

We now deriv the master equation for this model. To this end let us assume that after $s$ drawings (steps) there are $n_A(s) = m$ balls in urn $A$. Afte a further drawing there are only two possibilities. Either
\( n_A(s+1) = m+1 \) or \( n_A(s+1) = m-1 \). Since \( m = N+k \) and according to the nature of the draws we can write for the transition probabilities

\[
W(m+1|m) = \frac{2N-m}{2N} \frac{N-k}{2N},
\]

and correspondingly

\[
W(m-1|m) = \frac{m}{2N} \frac{N+k}{2N}.
\]

In order to make the meaning of the above transitions explicit we consider the special initial condition \( n_A(0) = 2N \). The first draw implies a decrease of the quantity \( \Delta_0 = 2N \) of 2. With the second draw the probability of a further decrease is \( 1 - 1/2N \), whereas the probability of an increase is only \( 1/2N \). For \( N \approx 10^{23} \) the probability for a decrease of \( \Delta_s \) is very large as long as \( \delta_s \) is not very small. In this case the irreversible decrease of \( \Delta_s \) is very probable.

As we have formulated it the model is a special case of a Markov chain. We introduce the conditional transition probability \( T(m,s|n) \) to find \( n_A(s) = m \) after \( s \) draws under the condition that for \( s = 0 \) we had \( n_A(0) = n \). \( T(m,s|n) \) satisfies the Chapman–Kolmogorov equation

\[
T(m,s|n) = \sum_l W(m|l)T(l,s-1|n),
\]

where it follows from (10.1) and (10.2) that

\[
W(m|l) = \frac{l}{2N} \delta(l-1,m) + \frac{2N-l}{2N} \delta(l+1,m).
\]

Explicitly the discrete master equation for the conditional transition probability reads

\[
T(m,s|n) = \frac{m+1}{2N} T(m+1,s+1|n) + \frac{2N-m+1}{2N} T(m-1,s-1|n).
\]

Accordingly, the discrete equation for the probability density for the special initial condition \( P(m,0) = \delta(n,m) \) is

\[
P(k,s) = \frac{N+k+1}{2N} P(k+1,s-1) + \frac{N-k+1}{2N} P(k-1,s-1).
\]

The above equation completely specifies the urn model.

It is now easy to show that the mean number of balls in urn \( A \) decreases exponentially towards its equilibrium value. To this end we calculate

\[
\langle k(s) \rangle = \sum_k kP(k,s)
\]

\[
= \sum_k k \left\{ \frac{N+k+1}{2N} P(k+1,s-1) + \frac{N-k+1}{2N} P(k-1,s-1) \right\}
\]

\[
= \sum_k (k+1) P(k+1,s-1) \left( \frac{1}{2} \frac{k+1}{2N} \right)
\]

\[
+ \sum_k (k-1) P(k-1,s-1) \left( \frac{1}{2} \frac{k-1}{2N} \right)
\]

\[
- \sum_k P(k+1,s-1) \left( \frac{1}{2} \frac{k+1}{2N} \right)
\]

\[
+ \sum_k P(k-1,s-1) \left( \frac{1}{2} \frac{k-1}{2N} \right).
\]
By renaming the summation variables the terms in the above exession can be written in the more concise form

\[ \langle k(s) \rangle = \langle k(s - 1) \rangle - \frac{1}{N} \langle k(s - 1) \rangle = \left( 1 - \frac{1}{N} \right) \langle k(s - 1) \rangle. \]

With the initial condition \( k(0) = n \) the solution to the above discrete equation reads

\[ \langle k(s) \rangle = n \left( 1 - \frac{1}{N} \right)^s. \]

In the limit \( N \to \infty, \tau \to 0, 1/N - \tau \to \gamma \), with \( s \tau = t \) this above expression becomes

\[ \langle k(t) \rangle = n \exp(-\gamma t), \]

which is just the monotonic exponential approach to equilibrium. In the mean \( \langle k(s) \rangle \) starts from \( n \) and approaches 0. Accordingly, it can easily be calculated that

\[ \langle k^2(s) \rangle = n^2 \left( 1 - \frac{2}{N} \right)^s + \frac{N}{2} \left[ 1 - \left( 1 - \frac{2}{N} \right)^s \right], \]

which in the limit \( s \to \infty \) tends to

\[ \langle k^2(s \to \infty) \rangle = \frac{N}{2}. \]

As we mentioned an important problem consists in studying the limit probability \( \lim_{s \to \infty} P(m, s|n) \). In general it is expected that this probability is independent of \( n \), so that we could name it \( W(m) \). However this is not the case for the ehrenfest model. A stationary distribution can be obtained provided that the value of \( n = n_0 \) of \( n_A(0) \) is not fixed initially and that a distribution \( W(n_0) \) of all possible values is taken, with \( \sum_{n=0}^{2N} W(n_0) = 1 \). In practice we replace the system of two urns by an ensemble of such systems with initial conditions distributed according to \( W(n_0) \). The stationary probability distribution satisfies

\[ W(m) = \sum_{n=0}^{2N} W(m|n_0)W(n_0), \]

with \( m \geq 0 \). One can verify that

\[ W(m) = \left( \frac{1}{2} \right)^{2N} \frac{(2N)!}{m!(2N - m)!}, \]

which is simply a binomial distribution. If \( 2N \) is sufficiently large the binomial distribution can be approximated by a gaussian distribution paked around \( \langle k \rangle = 0 \) with variance \( \langle k^2 \rangle = N/2 \),

\[ W(k) = \left( \frac{1}{\pi N} \right)^{1/2} \exp(-k^2/(2N)). \]

10.2.2 The Simulation

Schrödinger erwahnen;

10.2.3 Discussion of the Results

The "Umkehrinwand" of Loschmidt

The "Umkehrinwand" has been formulated in 1876 by Loschmidt as an argument against Boltzmann’s kinetic theory, which is based upon the Boltzmann equation. The starting point of the "Umkehrinwand"
is that fact, that in classical mechanics, which is at the basis of Boltzmann’s theory, all processes are reversible, whereas the Boltzmann equation describes irreversible processes. In particular, the $H$–theorem (as we will see in the next section) selects a particular direction of time, and hence breaks the reversibility. In fact, it can be shown for the Ehrenfest model, that the following conditional probabilities are equal

$$\text{Prob}\{n_A(s-1) = a|n_A(s) = m\} = \text{Prob}\{n_A(s+1) = a|n_A(s) = m\},$$

so that in a certain sense the model is reversible. This means, that the reversibility and the tendency of the $\Delta s$–curve to decrease are compatible.

The ”Wiederkehreinwand” of Zermelo

The ”Wiederkehreinwand” has been formulated in 1896 by Zermelo. It is an argument against Boltzmann’s derivation of the $H$–theorem from classical mechanics. The staring point of the Widerkehreinwand is the Wiederkehr Theorem of Poincaré, which states that each mechanical system composed of a finite number of particles returns arbitrarily near to its initial condition after a finite time, the Poincaré Widerkehrzeit. Obviously, this behaviour is in contrast to monontonous increase of the entropy which is preticted by the $H$–theorem.

Let us denote by $P(n_0|\bar{n}_0\ldots\bar{n}_0n_0)$ the probability that after $s$ draws the system is agian in the initial state $n_0$, where it is intended that all the $s-1$ $\bar{n}_0$ are different from $n_0$. The mean Widerkehrzeit” $\bar{T}$ is found to be

$$\bar{T} = \sum_{s=1}^{\infty} sP(n_0|\bar{n}_0\ldots\bar{n}_0n_0)$$

$$= \frac{1}{W(n_0)}$$

$$= \frac{2^N n_0!(2N-n_0)!}{(2N)!}.$$ 

As soon as $N$ is large and $n_0$ is significantly different from $N m$ $\bar{T}$ is a very large number!

### 10.3 Parallel Progamming with Java - Introduction

At this point it certainly makes sense to stop for a second with the discussion of physics and enter an exciting and meanwhile important part of program development: writing parallel programs. The necessity for this section will be clear, when we do the simulations of the nonequilibrium systems considered in the following sections, where the time for a simulation on one processor is too long. Of course it is much easier to study new things on a simple (toy) model, like the Ehrenfest model just discussed. So our aim will be to write a parallel version of the Ehrenfest model and to execute it in parallel on many computers at the same time.

In the first section we will discuss general issues connected to parallel programming, which not only applies to Java, but also to FORTRAN and other programming languages. In the second section we take a closer a look at the harware used for parallel computing and the third section will be about the software being applied to parallel algorithms. The fourth section will then finally show you an example of parallel programming in Java using the Ehrenfest model as an example problem.

#### 10.3.1 What is Parallel/Concurrent Programming?

**Why do we have to do it?**

These are probably the most often asked questions of the late 90s concerning high performance computing. Before we want to explain the terms in detail, let us look at the hardware development in the last 10 years to get an impression, why there is so much hype about parallel programming.

Almost every month the CPUs get cheaper and cheaper and additionally the CPUs get faster too. To be honest, todays CPUs in usual PCs are much too powerful to be used as simple terminals for word processing.
and some internet surfing. So the situation is that there are many “powerful” CPUs connected mostly by some kind of networking cable to the internet\(^1\) or the intranet\(^2\).

In contrary on the high performance computer market the CPUs were getting more expensive compared to the performance gained by using these high-speed processors instead of mass market consumer PC processors, most notably the Intel processors. So starting around 1990 the supercomputer companies started to use off the shelf processors, which are much cheaper and put them together with special (expensive) hardware, which connect these creating a very high performance connection between the CPUs. Although there are still companies producing specialized expensive processors e.g. for vector computing machines, most of the modern supercomputers are made out of “standard” processors, also found in small Workstations and PCs.\(^3\) Examples are the Cray T3E, which uses standard DEC Alpha Chips (21164 or 21264), the IBM SP2, which uses Power PC processors, the SGI Origin, which uses the MIPS R10000 and now also uses the IA64/IA32 architecture also “codenamed Merced”\(^4\) like HP\(^5\), and of course one of the fastest computers in the world: the “ASCII Red”, with about 8000 Intel Pentium Pro processors connected together. All these processors are also available in single machines, e.g. COMPAQ PCs use DEC Alpha chips, Apple and IBM sell Power PC computers and of course you will get the Intel Merced in a PC in the near future.

This leads to two important points to make: we now have a lot of computers all over the world connected by some sort of cable, having enough resources to do interesting calculations, while somebody is still working on the machine. We should also mention that these machines are mostly used during the day and are switched off during the night and weekends or run idle. Second the supercomputers of the late 90s are “just” a collection of single standard processors connected via a high performance network, which costs most of the money.

Then in the scientific and industrial areas there are many difficult problems, which can only be solved using very fast computers. This just means that the time to do the calculation on one processor takes months or even years and more. A natural step is therefore to ask: How can we use the given hardware consisting of many CPUs maybe even different ones connected by a network or a high performance network to get our calculations done in a shorter time?

The answer of course is to use all the CPUs available for one big problem. So it is like inviting some friends to help moving from one apartment to another. Without your friends it would take much longer, but now you have to manage for example ten people running around and have to tell them what to do, otherwise it would end up in a complete mess. This already leads us to the problem of how to write parallel programs?

Before diving into the details, let us define, what we mean by parallel and concurrent: The task of the programmer is to write code, which can then be executed somehow by many CPUs in PARALLEL. Writing this kind of code is called writing a CONCURRENT program. The computer is later taking care of running your code in parallel, you just have to write the concurrent code for this to happen. Because this difference is somewhat cumbersome to keep up with, we will use both terms interchangeably.

### 10.3.2 The Hardware

Basically there are certain problems to be aware of, when writing concurrent programs:

- What kind of the many different hardwares available, do you want to use as a parallel machine? *the hardware question*

- How are the machines connected? *the network question*

- Is your problem at hand capable of being run in parallel, or differently stated: is it possible to get an algorithm for solving your problem in parallel? *the algorithm question*

---

\(^1\) The name for the network of all computers in the world connected together.

\(^2\) The computers connected e.g. in a company (not necessarily connected to the outer world), the internet.

\(^3\) Actually Workstations just refer to the machines used for some way of high performance in graphics, number crunching or somewhere else. The difference to the standard PC has almost vanished and so just a price difference referring to the more expensive parts used for assembling the Workstations remained.

\(^4\) See [http://www.sgi.com/vision/tech.html](http://www.sgi.com/vision/tech.html)

\(^5\) See [http://www.bussinessservers.hp.com/toolbar/5\(\_\)10/csy\_i64support\_index.html](http://www.bussinessservers.hp.com/toolbar/5\(\_\)10/csy\_i64support\_index.html)
• How to tell the processors, how to cooperate with each other? the software question

Unfortunately there is still no possible way of avoiding to ask these questions before writing your code (at least you have to keep them in mind before you start parallel programs and pollute the network and keep the processors busy.)

<table>
<thead>
<tr>
<th>&lt;10Mbps</th>
<th>10Mbps</th>
<th>100Mbps</th>
<th>155Mbps</th>
<th>1000Mbps</th>
<th>&gt;1000Mbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPP / Slip / ADSL</td>
<td>Ethernet</td>
<td>Fast Ethernet</td>
<td>ATM</td>
<td>Gigabit Ethernet</td>
<td>Special (e.g. Crossbars)</td>
</tr>
<tr>
<td>Telephone Cable/ Modem/ TV Cable</td>
<td>copper cables</td>
<td>optic fibres/ special cables</td>
<td>(short distances)</td>
<td>1995</td>
<td>1999</td>
</tr>
</tbody>
</table>

slow —————————————————— fast
cheap —————————————————— expensive

Figure 10.2: The different networking models and cables used to connect the CPUs in a “Parallel Computer”. The years just represent the “standard” network used for most permanently connected systems. This is by no means a complete overview, but it should give an impression of what can be expected. The speed denoted above the network protocol is the theoretical maximum value. In reality you can be lucky, if you accomplish 1/10th of this value.

Table 10.1 Abbreviations used in the area of networking models.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mbps</td>
<td>Million Bits Per Second</td>
</tr>
<tr>
<td>ATM</td>
<td>Asynchronous Transfer Mode</td>
</tr>
<tr>
<td>SCI</td>
<td>Scalable Coherent Interface</td>
</tr>
<tr>
<td>PPP</td>
<td>Point to Point Protocol</td>
</tr>
<tr>
<td>SLIP</td>
<td>Serial Line Internet Protocol</td>
</tr>
<tr>
<td>ADSL</td>
<td>Asymmetric Digital Subscriber Line</td>
</tr>
</tbody>
</table>

The first two questions above are mostly interconnected with the third question about the capability of the problem to be parallelized. The network question being often more difficult to answer than the hardware question. A small (and incomplete) overview about possible network technologies is given in figure 10.2.

A very important issue for writing concurrent programs and using a parallel machine is addressed by the memory model of the machine in question. There are two different models implemented by the hardware: the shared memory model (SM) and the distributed memory model (DM). These refer to the access of the CPU to the memory available (not the disk space, this is a different concern).

Let’s look at two examples:

• SM: a 2 processor (SMP) PC machine.
  If you have one machine (say a PC), which has two CPUs, each one a Intel Pentium II with 350 MHz and a main memory of 128 MB RAM, then this is a shared memory machine having two processors. This of course is all put on one board (the motherboard). Both machines can access the memory at the same speed and they have equal priority. Of course if one of the processors wants to write or read the same address they have to be careful and probably have to wait for the other processor.

• DM: five connected PCs each with 1 processor.
  Now we have a so called cluster of networked workstations or PCs. These are for example connected
Table 10.2 Abbreviations used for describing computer hardware.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi Processor</td>
</tr>
<tr>
<td>MPP</td>
<td>Massively Parallel Processing</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non Uniform Memory Access</td>
</tr>
<tr>
<td>SM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>DM</td>
<td>Distributed Memory</td>
</tr>
<tr>
<td>DSM</td>
<td>Distributed Shared Memory</td>
</tr>
</tbody>
</table>

by an Ethernet, so each having a network card (PCI or other) and a cable (e.g. twisted pair copper cable) connecting them. Each of the machines have 32 MB RAM, so altogether there are five times 32 MB RAM, which adds up to 160 MB RAM. But this memory is of course no longer available to all the processors at the same speed. If one processor wants to read/write a value from/to an address in one of the other machines, it has to communicate using the network. So in this case we have a distributed memory.

There are two special cases, which are very common and have to be mentioned separately: The SMP and the NUMA architectures. The SMP machine as already mentioned in the first example above is a computer/mainboard, which has two or more processors directly on one board. All the processors together have one main memory, which they have to share. But for each CPU the computer looks exactly the same, therefore the name SMP (Symmetric Multi Processor).

The NUMA architecture refers to a special kind of shared memory architecture. For example the SGI Origin 2000 is an example thereof. In this model you take SMP boards (you can take single CPU boards too) each with a certain amount of onboard main memory and put them together in a rack. The boards get connected by special networking hardware for example a crossbar. The racks can then also be connected again with specialized hardware. This hardware/network has to make sure that the total memory is available to all CPUs at (almost) the same speed and that the caches used for the memory is always coherent (not out of date). Each processor can therefore access all the memory directly without any special considerations, but actually the time (called latency) to access different addresses in memory can be different depending on where the processor is and which memory you want to access (the name NUMA refers to this: Non Uniform Memory Access). But the whole memory management is implemented in hardware and therefore hidden from the user.

10.3.3 The Software

Now that we have decided or know about the hardware to use for the problem we want to solve, we have to decide what software model to use to get all CPUs working for us.

First of all you have to think about an algorithm for the parallel implementation of the problem to be solved. This is actually a difficult and time consuming part and you probably have many different strategies/algorithms for one single problem. You also do not know beforehand, how each of the algorithms will compare to each other on the same platform and how they will perform on different hardware.

If you have an idea, how to use many CPUs for your problem, then you can start thinking about the software model to be used for your available hardware (see figure 10.3). This decision is the most difficult one, because it is not very easy to go back and use a different model. Writing the code for a given software model might be very easy and fast or you might have to spend a few months to get an efficient code. Then you can run your program on different platforms and use the fastest available for your problem.

6Until 1999 the crossbar technology was basically not available for Intel platforms, but now COMPAQ even proposes servers with up to eight Intel Xeon processors using crossbar technology called “Profusion Eight-Way Architecture” (see http://www.compaq.com/products/servers/technology/eightway-pd.html). This again will lead to a decrease in hardware prices for the high performance shared memory (NUMA) architectures.
10.3. PARALLEL PROGRAMMING WITH JAVA - INTRODUCTION

Figure 10.3: Possible software models to be used for parallel programming. For a description of the used abbreviations see table 10.3. Difficult/Easy programming refers to the time needed to implement parallel algorithms using a certain model.

### Table 10.3 Abbreviations used for the models in parallel programming.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSM</td>
<td>Distributed Shared Memory</td>
</tr>
<tr>
<td>HPF</td>
<td>High Performance Fortran</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>PVM</td>
<td>Parallel Virtual Machine</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multiple Processes</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>COBRA</td>
<td>Common Object Request Broker Adapter</td>
</tr>
</tbody>
</table>

But what are the different software models?

**RMI and COBRA**

COBRA and RMI are basically an open standard for a service provided by many computers to distribute objects (in the sense of objects the oriented paradigm) across heterogeneous networks. It is kind of a network service to supply programs with certain objects not available on the computer running the program. What you need for this to work are so-called ORB (Object Request Brokers), which are servers for objects and their names. You can call it the “object oriented version of MPI”.

Both can of course be used for object oriented parallel computing, but still they have not gained a lot of attraction in the scientific community. Not because they are inferior to the other models, but there is a much higher hill to be climbed in order to write programs using RMI or COBRA. Of course this is a very subjective opinion, but all the other models are very widespread for high performance computing. RMI and COBRA are still in their beginnings.

This ansatz to distributed (parallel) computing is mostly used for large databases, network applications, etc. RMI is included in standard Java, but COBRA is coming separately and you need additional software.

But you should keep in mind, that RMI and COBRA are actually “extensions” to using threads, because threads are only possible for shared memory systems. For distributed memory systems, like large intranets or even the whole internet, you have to use RMI for parallel applications in Java or choose COBRA. The other choices are the subject of the next few sections. A nice introduction (based on Java 1.1) to RMI and COBRA in Java is [Farley, 1998].

Since Java 2 all the COBRA functionality is included in the standard and there are 7 standard API packages, which implement all necessary stuff to use COBRA in Java. Take a look at the JDK 1.2 API documentation.
OpenMP

If you want to spend less time for coding, you should use OpenMP, an approach based on directives. You just have to insert comments (FORTRAN) or pragmas (C/C++) before the code you want to be executed in parallel. The DoAccross directives and other directives based models were the forerunners of OpenMP and should not be used anymore, because all companies have already or are going to switch to the OpenMP standard sooner or later. The advantage of OpenMP is, that you do not have to change your serial program, you just add comments (or pragmas in C/C++) and most of the work is done by an appropriate compiler. The drawbacks are the restriction to shared memory machines \(^7\) and the worse performance compared to good MPI programs.

?? Example OpenMP program

Symphony

An alternative to OpenMP would be to use a kind of server, which distributes small pieces (jobs) of your problem to many machines available and collects the results to be processed later on, therefore exploiting a kind of distributed memory model, but with no communication between the different jobs. An example of this philosophy realized in Java is a project developed at the University of Freiburg called Symphony (see later on).

HPF

If you want to spend more time thinking about how the data and the work can be distributed to the available processors, then you can go with HPF. HPF employs a data distributed model and is of course only available to Fortran 77/90/95 programmers and has not found a very widespread use. But still it is a viable tool for many problems and it is available on almost any computer architecture. An interesting solution is the (free) ADAPTOR software, which converts HPF programs to Fortran 90 with MPI calls or OpenMP directives.

MPI / PVM

Even more work and time has to be spent to write a MPI/PVM (see table 10.3 program. PVM was the first available message passing interface, but has meanwhile been abandoned by most people in favour of MPI, which is today the standard to use for writing message passing programs.

In this model you have to take care of everything yourself. You have to tell the computer, when, what and with who you want to exchange messages (data). This works by writing a single program, which consists of several calls to MPI functions\(^8\). There are functions for sending messages from one CPU to another or you can send data from one CPU to all other CPUs working on the same problem (a so called communicator). The compiler (or to be exact the library) just translates the MPI commands to a lower level of networking commands, depending on the available library and network hardware used.

For example if you have an ethernet network with several PCs running Linux/Windows, you can install a free version of MPI (e.g. MPICH or LAM) and compile the MPI programs. The code will then be run on all requested CPUs using the ethernet protocol to communicate and exchange data. On a shared memory machine (including SMP machines) you can also use MPI, but this time the communication could be done directly, so using MPI for an SMP machine is usually a big overhead and should be avoided. The biggest advantage of MPI is, that it is available for all platforms and hardware architectures, so it is completely portable.

(Java) Threads

The last alternative for writing parallel programs is using threads. Threads are small parts of a single program, which are to be executed by the same CPU using time sharing or by a separate CPU. So it is kind

\(^7\)There are meanwhile software packages available, which convert OpenMP programs into usual Fortran/C/C++ code using MPI calls and therefore running on distributed memory machines, e.g. OdinMP, etc.

\(^8\)You just have to add a library to your compiler commands and you have to use a special program to run the parallel MPI program (mpirun).
of using MPI calls, but this time you are not calling high level library functions, but low-level functions. These are implemented on the level of the underlying operating system. This means your program is very fast in starting a job on a different CPU and it is very easy to have global variables, but you are confined to shared memory machines and to a certain operating system. So this is usually not the model of choice for a scientific problem.

But with Java this is not (always) correct anymore. In Java it is very easy to use threads and because Java is portable across all platforms, almost all of the restrictions just mentioned no longer apply. But of course you still have to work a bit more to get a threaded program running correctly compared to an MPI or OpenMP program. But it definitely has become an interesting alternative, if Java is becoming much faster in the next few years (and we are sure it will).

### 10.3.4 Amdahl’s Law

At last we want to mention an important point, which has already been made by G. Amdahl in 1967: If you denote the part of a program to be run sequentially by $f$, then the “speedup” $S_N$ can be calculated as

$$ S_N = \frac{T_1}{T_N} = \frac{N}{1 + f(N-1)}. $$

Where the speedup is defined as the ratio of the time to complete the serial program and the time needed by using $N$ processors simultaneously. This dramatically restricts the performance gained by using many CPUs. So the sequential parts of a parallel program should be kept as small as possible. Fortunately the speedup depends on the problem size and will increase for bigger problems.

![Figure 10.4: Amdahls law with some examples for the value of the serial part $f$ of a program.](image-url)

CPUs. So the sequential parts of a parallel program should be kept as small as possible. Fortunately the speedup depends on the problem size and will increase for bigger problems.
10.3.5 The Ehrenfest Model using a Parallel Program

Here we want to demonstrate, how to write and use a parallel program employing the Ehrenfest model as an example problem.

Symphony

Using Symphony is very easy in this case. The idea is to compute the same model (number of time steps) with different number of fleas on dog one. Then we can generate very easily the statistics for the Ehrenfest model discussed in section 10.2.

```java
/**
 * DogFlea.java
 */

/*
 * Created: Thu Jan 21 11:30:31 1999
 * @author Peter Biechele
 * @version 10
 */

import java.awt.*;
import java.awt.event. *
import java.util.*;
import java.applet.*;
import ptolemy.plot.*;
import simulation.*;

public class DogFlea extends Applet {

    // Input values for Symphony
    public double Nd = 1000;
    public double n0d = 10;

    public int N = (int)Nd; // number of fleas
    public int n0 = (int)n0d; // initial number of fleas on dog 1

    public int steps = 10000; // How many time steps to be done
    int [] flea; // to which dog belongs the flea ? (1 or 0)

    // Output value for Symphony
    public int na, nb;

    // Parallel FLAG: 0 No parallel job, 1 use Symphony
    int SYMPHONY=1;

    Random rand = new Random();
    Plot plot;

    public void run () {
        int choose;
        boolean connect=false;

        instVars();
        initial ();
        na = calck ();
```
10.3. PARALLEL PROGRAMMING WITH JAVA - INTRODUCTION

```java
if (SYMPHONY==0) {
    System.out.println("s=0":na);
    plot = new Plot();
    plot.setTitle("Difference of fleas versus time");
    this.add(plot);
    this.repaint();
}

for (int s=1; s<steps; s++) {
    choose = Distribution.nextInteger(rand,N-1);
    flea[choose]=flea[choose];
    na = calcK();

    if (SYMPHONY==0) {
        plot.addPoint(0,s,na,connect);
        if (connect==false) connect=true;
        if (s%100==0) plot.repaint();
    }
}

nb=N-na;
if (SYMPHONY==0) plot.repaint();

int calcK() {
    int sum=0;
    for (int i=0; i<N; i++) {
        sum+=flea[i];
    }
}

void initial() {
    for (int i=0; i<N; i++) {
        if (i<n0) flea[i]=1;
    }
}

void instVars() {
    flea = new int[N];
}

public DogFlea() {
}

public static void main(String[] args) {
    Frame f = new Frame("Dog-Flea Model");
    f.setSize(600,600);

    DogFlea prg=new DogFlea();
    f.add(prg);
    f.show();

    // Close Window event
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    prg.init();
```

```java
prg.start();

public void init() {
}

public void start() {
    run();
}

public void stop() {
}

} // DogFlea
```

First you have to compile the java program and copy the class file to the directory, where a browser can access it (e.g. /home/user/public_html/Symphony/SymphonyTasks for a unix machine). Then on the Symphony server machine, which has to have a WWW server running you have to edit the file ..../Symphony/SymphonyServer/tasks.sym to contain the parameters you want to compute, e.g. it could look like this:

```java
process=DogFlea;
entryPoint=run;
param(n0d),min=10,max=1000,incr=10;
param(Nd),const=1000;
return (na,nb) as {EVENT @DogFlea @run0};
timeout=1;
```

Now you start the Symphony server with

```shell
$ ./Symphony/SymphonyServer/start_server
```

and on the clients (the computers you want to have to participate on the computation) you start the appletviewer with

```shell
appletviewer http://SymphonyServer/~user/Symphony/applet.html ,
```

where you have to substitute your Symphony server WWW address and the name of the user. This assumes you have a unix system, for a Windows system just change the path accordingly.

The clients are only computing, if the appletviewer window is visible, so the user calling the page to start the Symphony applet can decide if she/he wants to participate or not. On the clients you get the messages:

```plaintext
{DogFlea.59}
...got to process DogFlea.59
Results of run;
class: DogFlea;
entry point: run;
Input Field: n0d=810.0; Input Field: Nd=1000.0;
Output Field: na=479; Output Field: nb=521;
Time to process: 3.0 seconds;
** End of Symphony run **
```

This shows one job, which was run using 810 fleas initially on dog 1 and 1000 fleas altogether. The results get saved in a file called event.log in a directory

```shell
.../Symphony/SymphonyServer/data/DogFlea/run0/
```

The second step is to extract the results from the logfile, e.g. use the following program:

---

3For example the Apache web server. Apache is freely available for all unixes and even for Win32 since a few months ago, although the performance is much better on unix.
/**
 * ConvertSymphony.java
 *
 * Created: Tue Jan 26 10:39:01 1999
 * @author Peter Biechele
 * @version
 */

import java.io.*;

public class ConvertSymphony {

    // Files for read and write
    File filein = new File("event.log");
    File fileout = new File("gnu.out");

    // Parameters to read out: also the output order in the new file
    String [] p = { "nd", "Nd", "na", "nb" };

    public ConvertSymphony () {
    }

    public static void main(String [] args) throws IOException {
        ConvertSymphony a = new ConvertSymphony ();
        a.run ( ) ;
    }

    public void run () throws IOException {
        FileReader fin = new FileReader ( filein ) ;
        FileWriter fout = new FileWriter ( fileout ) ;
        BufferedReader in = new BufferedReader ( fin ) ;
        BufferedWriter out = new BufferedWriter ( fout ) ;

        String line , lineout , sdummy ;
        char schar ;
        int pos , start , end , flag ;

        out . write ( "##" ) ;
        for ( int i = 0 ; i < p . length ; i ++) {
            out . write ( "_" + p [ i ] ) ;
            p [ i ] = p [ i ] + "=";
        }
        out . newLine ( ) ;

        while ( ( line = in . readLine ( ) ) != null ) {
            // System . out . println ( line ) ;
            flag = 0 ;
            lineout = new String ( ) ;
            for ( int j = 0 ; j < p . length ; j ++) {
                pos = line . indexOf ( p [ j ] , 0 ) ;
                start = pos + p [ j ] . length () ;

                end = start ;
                schar = line . charAt ( end ) ;
        }
\textbf{CHAPTER 10. NONEQUILIBRIUM MONTE-CARLO METHODS}

while ( schar != ';' ) {
    end++; \\
    schar = line . charAt ( end );
}

sdummy=","+line . substring ( start , end );
// System . out . println ( j+" : "+p[j]+ " : : "+sdummy );
if ( j != 1 ) {
    lineout = lineout . concat ( sdummy );
}
else if ( Double . valueOf ( sdummy ) . intValue ( ) == 1000 ) {
    flag = 1;
}

if ( flag == 1 ) {
    out . write ( lineout );
    out . newLine ( );
}

in . close ( );
fin . close ( );
out . close ( );
fout . close ( );
}

} // ConvertSymphony

This produces a data file containing the variables (n0d Nd na nb) and can be plotted using PtpLOT or by any plotting program e.g. gnuplot (after starting gnuplot use for example gnuplot "gnu.out" using 1:3).

?? Discuss sprintf() and StringBuffer ????? Other method (String, arrays) ?? new version (ints)

\textbf{Java Threads}

Now we take a look at Java Threads, the details can be found in [Oaks und Wong, 1999].

A thread is a shorthand for “thread of control” and stands for a section of code executed independently of other portions of the code in a single program. You are actually already familiar with this concept, because every operating system today uses multitasking to run multiple programs at the same time sharing one CPU. If you view these “tasks” as single threads, we only have to extend to multiple threads in a single program.

In Java the JVM is responsible of starting multiple threads in your code. That is also where the restrictions come in: most of the JVMs did not support multiple threads running in parallel on SMP machines so far, meaning that you do not have real parallel programs working with all available CPUs. Nevertheless the concept of threads is useful for single processor machines. For example we have already met this case in the last chapters, where we did extensive calculations, while still the user had the opportunity to interrupt the calculation by hitting a mouse button or moving the mouse. This is only possible, if during the simulation a thread is always asking for user intervention.

The JVM thread implementation of multiple threads, with only one thread being active at a time, is called green threads\$^{10}\$. All JDK implementations employ these green threads for Java threads.

Here we want to concentrate on running different threads on several CPUs. To that end we need a JVM, which supports “real” threads or so-called native threads\$^{11}\$. If the JVM you are using supports native threads, should be stated in the JVM documentation. For the JDK, all the 1.2 versions on all platforms support and use native threads without further notification. For JDK 1.1.7 running Linux/Solaris you can get an additional package, which supports native threads. To call the native threads version you have to use java -native Program.

\$^{10}\$ These threads are user-level threads. It means that the first thread started by the operating system starts multiple threads itself and the operating system does not interfere. These are also called lightweight processes, because the kernel of the operating system does not have to do much work for switching from one user-level thread to another.

\$^{11}\$ These are so-called kernel-level threads. This time the kernel of the operating system has to do some context switching and therefore there is a little overhead compared to user-level threads.
Native threads are only useful for threads, which do some heavy calculations. For the case where you only want to have user interaction and calculation running in “parallel”, like in the molecular dynamics simulation or the Ising model, we are better off with the green threads.

Applets also employ threading, but here we also have to check if the browser or appletviewer running the applet “only” have green threads or even support native threads. Actually I do not know of any JVM implemented in a web browser capable of using native threads yet. But the need for threads (green threads) in applets is even more important than for applications, because if you have an applet on top of a web page and the applet has to do some time consuming initializations, it will stop showing the rest of the page until the applet finishes. If the browser itself (not the Java applet) uses a seperate thread for the applet, this is not a concern, but still you should use your own Java threads for time consuming operations in an applet.

Now let us take a look at our example and how we can use Java threads to speed up our calculation of the Ehrenfest model. The Thread class in Java is located in the java.lang package. We basically have two ways of defining a thread before we can use it.

- Extending the thread class in a separate file. Example: First the thread:

```java
public class OurOwnThread extends Thread {
    public void run() {
        // here we can do computations
    }
}
```

and second the calling class:

```java
public class OurOwnThread1 {
    public static void main(String[] args) {
        // main method of application
        OurOwnThread own = new OurOwnThread();
        own.start(); // this calls the run() method of the thread
    }
}
```

- Implementing a runnable interface. Implementing an interface means to supply all necessary methods needed for an interface (here the thread) to be build up. For a thread you only have to supply the `run()` method, nothing else. Example:
public class OurOwnThread2 implements Runnable {
    public static void main(String[] args) {
        // main method of application
        OurOwnThread2 own = new OurOwnThread2();
        Thread th = new Thread();
        th.start(); // this calls the run() method of the thread
    }
    public void run() {
        // the computation method of the thread
    }
}

Which way you use does not matter and there is no preferred way, it just depends on your personal preference.

Now there might be a little confusion about the start() method. There is a start() method in the Applet class and one in the Thread class. The same applies to the stop() method. So you have to be a little bit careful to which method you refer, but it will always be clear from the context. And a second remark is about the run() method. You should never call directly, but always call the start() method of the thread, which in turn starts the thread and calls the run() method. This is a common mistake, because the program starts and works correct, but it does start the threads sequentially and not in parallel.

Now finally let us see how we can apply this to our problem. First we need the thread:

```java
import java.util.*;
import simulation.*;

/**
  * DogFleaCalc.java
  *
  * @author Peter Biechele
  * @version 1.0
  */
 public class DogFleaCalc extends Thread {
    int na, nb; // Output values
    int n0, N, steps;

    private int[] flea; // to which dog belongs the flea? (1 or 0)
    private Random rand = new Random();

    DogFleaCalc(int N, int n0, int steps) {
        this.flea = new int[N];
        this.n0 = n0;
        this.N = N;
        this.steps = steps;
    }

    public void run() {
        int choose;
        boolean connect=false;

        initial(n0);
        na = calc(k);

        System.out.println("at time s=0:");
        for (int s=1; s<steps; s++) {
            choose=Distribution.nextInteger(rand,N-1);
            ...
10.3. PARALLEL PROGRAMMING WITH JAVA - INTRODUCTION

```java
flea[choose]=1-flea[choose];
na = calck();
}

nb=N-na;
System.out.println("at time s=\"+steps+\";\"+na);
}

int calc() {
    int sum=0;
    for (int i=0; i<N; i++) {
        sum+=flea[i];
    }
    return sum;
}

void initial(int n0) {
    for (int i=0; i<N; i++) {
        if (i<n0) flea[i]=1;
    }
}

} // DogFleaCalc

And second we need a "server", which starts the threads and collects the answers; this is very similar to the Symphony example and will be the same in the MPI problem:

```
f. setSize (600,600);
DogFleaThreads prg=new DogFleaThreads ();
f. add (prg);
f. add (new Label("...I am calculating...") );
f. show ();
// Close Window event
f. addWindowListener ( new WindowAdapter() {
    public void windowClosing (WindowEvent e)
    {
        System . exit (0);
    }
});
prg . init ();
prg . start ();

public void init () {
    if ( prg == null ) {
        prg = new DogFleaThreads ();
    }

current = Thread . currentThread ();
current . setPriority (Thread . MAX_PRIORITY);

    // Ensure Round Robin Scheduling
    // SimpleScheduler ss = new SimpleScheduler (100);
    // ss . start ();
}

public void start () {
    int n0; // initial number of fleas on dog 1
    int stepsze = N/( Nthreads -1);
    Thread [ ] job = new Thread [NCPUS];

    // start exactly NCPUS threads, then wait for them to finish
    // and start again NCPUS threads
    int prio=Thread . currentThread (). getPriority ()-1;
    int count =0;
    for ( n0 =0; n0<N; n0 += stepsze ) {
        job[ count ] = new Thread ( new DogFleaCalc (N, n0 , stepsze ) ) ;
        job[ count ]. setPriority (prio);
        job[ count ]. start ();
        count ++;
        if ( count ==NCPUS ) {
            // start all threads
            for ( int j =0; j<NCPUS; j++) {
                // job[ j ]. start ();
            }
            System . out . println ("...Waiting...");
            // wait for all threads to be done
            for ( int j =0; j<NCPUS; j++) {
                try {
                    job[ j ]. join ();
                } catch ( InterruptedException e ) {} 
            }
            System . out . println ("...Ready...");
            count =0;
        }
    }
System . exit (0);
10.3. PARALLEL PROGRAMMING WITH JAVA - INTRODUCTION

```java
} public void stop () { }
} // DogFlea
```

The `void join()` method returns only if the thread is marked as no longer being alive and therefore being completed. The `join` method also accepts an argument, which is the timeout (a long variable) in milliseconds. If you give a timeout, `join` returns at the latest after the timeout time.

If you compile both classes and start the `DogFleaThreads.java` class you will see the threads starting and stopping, here we always start two threads at a time and wait for both to finish.

Try the two different executions:

- `java -green DogFleaThreads`
- `java -native DogFleaThreads`

On UNIX you can use the `time` command to check the execution times. Here we run both commands (`-green` on the left and `-native` on the right) on a Linux machine using the JDK1.2 on a single and a two processor machine.

On a single processor machine you get:

```text
Waiting !
at time s=0:  0
at time s=100000:  499
at time s=0:  333
at time s=100000:  496
Ready !
Waiting !
at time s=0:  666
at time s=100000:  475
at time s=0:  999
at time s=100000:  530
Ready !

real   4m32.880s
user   4m1.290s <--------
sys    0m4.750s <--------
```

And on a 2 processor machine you will get:

```text
Waiting !
at time s=0:  0
at time s=100000:  491
at time s=0:  333
at time s=100000:  496
Ready !
Waiting !
at time s=0:  666
at time s=100000:  473
at time s=0:  999
at time s=100000:  478
Ready !

real   0m14.397s <--------
user   0m12.610s
sys    0m0.180s
```

And on a 2 processor machine you will get:

```text
Waiting !
at time s=0:  0
at time s=100000:  491
at time s=0:  333
at time s=100000:  496
Ready !
Waiting !
at time s=0:  666
at time s=100000:  473
at time s=0:  999
at time s=100000:  478
Ready !

real   0m8.216s  
user   0m12.970s
sys    0m0.170s
```

Neglect the absolute times given, because we used two totally different machines for the tests. But look at the relative timings comparing the native and the green threads on single and multiple processor machines.
By using a 2 processor machine we almost get a speedup of 2 for this problem (look at the real time), and on the single processor machine the performance gets worse by using native threads (look at the user and system time), which is due to the nature of kernel-level threads and user-level threads. The use of JDK1.2 is just due to the fact that a JIT is already supplied with the 1.2 version and we wanted to speed up the program. You could also use another JIT with JDK1.1.

You should also note that the messages appear randomly and not in any order. So if you use output in threads you should always remember, that there is no given order for the output to be send to the screen. This will be also true for MPI programs.

The reason for the randomness is the way the JVM executes the threads. This is called the “scheduling of threads”. Like the operating system has to tell, at what time and how long the processes are to be executed, the JVM has to use a scheduling algorithm to tell each thread, when it has to go on and when it has to wait. Because this is a difficult issue, we do not go into details. The scheduling used in the JDK is not written in detail in the JVM specifications and therefore there are differences on different platforms. It only states that it has to be a pre-emptive, priority based scheduler, where the priority (greater equal to zero) is a natural number and the highest number refers to higher priority. The case of threads with equal priority creates most of the problems. For a more detailed discussion and examples of e.g. a Round-Robin Scheduler written in Java, you should take a look at a book about threading (like [Oaks und Wong, 1999]).

An important issue concerning threads is of course a race condition. This is the problem of ensuring that a variable (e.g. a global one) is accessed by only one thread at a time and if a variable is changed, all the other threads have to know about it. To ensure these there are certain constructions to be used, like the synchronized keyword. Because this is a rather complex topic, you should consult a comprehensive discussion, e.g. in [Oaks und Wong, 1999].

Just to complete this section, we want to mention additional methods of the Thread class.

**static void sleep(long t)** This method stops the execution of a thread for t milliseconds.

**boolean isAlive()** Another useful method is the isAlive() method, which returns true, if the thread is still “running”. Do not confuse this with the boolean isActive() method of the Applet class, which returns true, if the applet is somewhere still executing (executing code between the start() and the stop() method.).

**static Thread currentThread()** This returns the thread object of the current thread.

**static int activeCount()** Returns how many threads are running.

There are many other methods, but some of them are already depreciated as of Java 2 (e.g. suspend(), stop(), resume(), etc.), because they introduce serious problems. Deprecated means, you should avoid using these functions, because they might not be available in a new Java version anymore.

There is one last remark you should keep in mind: Never restart a thread!! The reason for this is a bit involved and the interested reader should refer to [Oaks und Wong, 1999].

**MPI and Java**

??????????? which interface MpiJava ???

### 10.4 Master Equations, Entropy, and the \( H \)–Theorem

In this section we are going to discuss in more detail the connections between the master equation and the entropy. In particular we want to derive the \( H \)–theorem, which we have already mentioned in the previous section. The starting point of our discussion will be a master equation of the general form

\[
\frac{\partial}{\partial t} P(x,t) = \sum_{x'} [w(x'|x)P(x',t) - w(x|x')P(x,t)]
\] (10.3)
and we want to discuss separately two cases: (i) the transition matrix is symmetric, i.e.,

\[ w(x|x') = w(x'|x); \]

(ii) the condition of detailed balance is satisfied

\[ w(x|x')P_{eq}(x',t) = w(x'|x)P_{eq}(x,t). \tag{10.4} \]

**Case (i): The Symmetric Transition Matrix**

It is evident that for these case the master equation (10.3) can be written in the simplified form

\[ \frac{\partial}{\partial t}P(x,t) = \sum_{x'} w(x|x')\left[P(x',t) - P(x,t)\right]. \tag{10.5} \]

Thus, at equilibrium we have

\[ \frac{d}{dt}P = 0, \]

since \( w(x|x') \geq 0 \). Hence,

\[ P_{eq}(x,t) = \text{const} = \frac{1}{\Omega}, \tag{10.6} \]

where

\[ \Omega = \sum_x 1. \]

In order to demonstrate that equilibrium is approached in a unique way we choose the following \( H \)-function

\[ H(t) = \sum_x P(x,t) \ln P(x,t). \]

We now calculate explicitly the time derivative of the above defined function. For notational ease we replace the sum by an integral. We find

\[ \frac{d}{dt}H(t) = \int dx \left[ \frac{\partial}{\partial t}P(x,t) \right] \ln P(x,t) + \frac{\partial}{\partial t} \ln P(x,t). \]

Because of the conservation of the norm of the probability density

\[ \int dx P(x,t) = 1, \]

the time derivative of the \( H \)-function is

\[ \frac{d}{dt}H(t) = \int dx \left( \frac{\partial}{\partial t}P(x,t) \right) \ln P(x,t). \]

Now, we insert the master equation (10.5) into the above expression and obtain

\[ \frac{d}{dt}H(t) = \int dx \int dx' w(x|x') (P(x',t) - P(x,t)) \ln P(x,t), \]

\[ = \int dx \int dx' w(x'|x) (P(x,t) - P(x',t)) \ln P(x',t). \]
The second line in the above expression has been obtained by changing the names of the integration variables. Exploiting the symmetry of the transition matrix, the time derivative of the $H$–function can be written in the symmetrized form

$$
\frac{d}{dt} H(i) = \frac{1}{2} \int dx \int dx' w(x'|x) \left( P(x',t) - P(x,t) \right) \left( \ln P(x,t) - \ln P(x',t) \right).
$$

Recalling that

$$(a - b)(\ln a + \ln b) \geq 0$$

for $a, b > 0$ it follows for the time derivative of the $H$–function that

$$
\frac{d}{dt} H(i) \leq 0,
$$

The $H$–function decreases monotonically as a function of time during the time evolution of the Markov process. The $H$–function gets stationary when the condition

$$
P(x, t) \to P_{eq}(x); \quad P_{eq}(x) = P_{eq}(x')
$$

is satisfied for each pair of states $x$ and $x'$, which are connected by nonvanishing transition matrix elements. If the set of the states $x$ does not decay into two or more independent subsets, the equilibrium is uniquely characterized by the form (10.6).

**Case (ii): Detailed Balance**

Case (ii) is of course more general than case (i). In order to investigate this second case we choose the following $H$–function

$$
H(i) = \int dx P(x,t) \ln \frac{P(x,t)}{P_{eq}(x)}
$$

and show that again the $H$–theorem holds, i.e.,

$$
\frac{d}{dt} H(i) \leq 0.
$$

To this end we define a new matrix, say

$$
\tilde{w}(x|x') = w(x'|x) P_{eq}(x'),
$$

which because of the property of detailed balance (10.4) is symmetric. With the help of the definition (10.8) we can write the master equation in the form

$$
\frac{\partial}{\partial t} P(x,t) = \int dx' \tilde{w}(x'|x) \left[ \frac{P(x',t)}{P_{eq}(x')} - \frac{P(x,t)}{P_{eq}(x)} \right],
$$

Inserting this form of the master equation in the definition of the $H$–function (10.7) we can show in a form analogous to the previous subsection

$$
\frac{d}{dt} H(i) = \frac{1}{2} \int dx \int dx' \tilde{w}(x'|x) \left[ \frac{P(x',t)}{P_{eq}(x')} - \frac{P(x,t)}{P_{eq}(x)} \right] \left( \ln \frac{P(x,t)}{P_{eq}(x)} - \ln \frac{P(x',t)}{P_{eq}(x')} \right) \leq 0,
$$

If the system is ergodic, after some time it reaches the equilibrium

$$
P(x,t) \longrightarrow P_{eq}(x)$$
and the $H$–function assumes its minimal value. The $H$–function (10.7) has two important properties. The special form of the $H$–function allows the proof of a $H$–theorem also for the Boltzmann equation. The interesting point is that the Boltzmann equation itself being a nonlinear equation for the distribution in a 6 dimensional one–partice phase space is not a master equation. However, the linearised Boltzmann equation is a master equation.

The function $H$ is an additive quantity. Consider two independent systems with states $x$ and $y$ and probability densities $p(x)$ and $q(y)$. We regard the two systems as a combined system with states $(x, y)$ and probability density $p(x)q(y)$. Then the $H$–functional of the combined system is the sum of the $H$–functionals of the two subsystems

$$\int dx \int dy P(x)Q(y) \ln \frac{P(x)Q(y)}{P_{eq}(x)Q_{eq}(y)} = \int dx P(x) \ln \frac{P(x)}{P_{eq}(x)} + \int dx Q(x) \ln \frac{Q(x)}{Q_{eq}(x)}.$$  

Thus, $H$ is an extensive quantity.

Because of these two important properties one could like to identify $H$ with the entropy of the second Hauptsatz of thermodynamics. However, it is important to keep in mind, that the $H$–functional we introduced is a functional of a nonequilibrium distribution, whereas the thermodynamic entropy is defined for a system at equilibrium. The nonequilibrium $H$ functional allows therefore a generalization of the thermodynamic entropy

$$S = -kH + S^{eq},$$

where $k$ is Boltzmann’s constant and $S^{eq}$ is a constant term, which is independent of $p(x)$. At equilibrium $H = 0$, so that $S^{eq}$ is the thermodynamic entropy.

### 10.5 Nonequilibrium Thermodynamics

#### 10.5.1 Balance Equations of Fluid Dynamics

In classical theoretical physics the dynamics of fluids is described with the help of the balance equations of nonequilibrium thermodynamics for the mass, the momentum, and the energy. These balance equations are derived under the assumption, that in a small volume element of the bulk of the fluid mass, momentum, and energy are in a local thermal equilibrium. Evidently, if one considers the molecular structure of a fluid, one has to regard the mass, the momentum, and the energy density as mean value of strongly fluctuating microscopic quantities. These fluctuations are neglected in the purely macroscopic classical description.

However, as we have seen in the preceding sections, fluctuations play a fundamental role in systems out or approaching equilibrium. Typical situations in which fluctuations are of central importance in the description of fluids occur near equilibrium in the form of small Gaussian fluctuations in the macroscopic variables, these are the so-called hydrodynamic fluctuations, and the large non–Gaussian fluctuations in the velocity filed of turbulent fluids.

For these reasons it seems natural to look for a stochastic, mesoscopic formulation of fluid dynamics. On this macroscopic level we will have the same number of variables as in the macroscopic level of description. The dynamics of these variables will no longer be deterministic but stochastic. The mesoscopic approach should satisfy four conditions:

(i) In the limit of vanishing fluctuations the equation of motion for the mean values of the stochastic variables should be identical with the balance equations of fluid dynamics.

(ii) In a linear noise approximation we should obtain from the master equation defining the stochastic process the linear Langevin equation of fluctuating hydrodynamics. In this way nonequilibrium thermodynamics should be contained in the approach.

(iii) In the continuum limit the characteristic function of the stochastic process should generate the complete hierarchy of coupled equations for the moments of the velocity fields, which are known from the theory of turbulence. In this way the master equation should describe correctly the turbulent, non–Gaussian fluctuations.

(iv) The mesoscopic formulation should lead to simple and efficient numerical algorithms, which should be vectorizable and parallelizable.
In the following we are going to investigate such a formulation for a very simple balance equation, the one dimensional Burgers equation, which is the equation of motion for a one–dimensional velocity field \( v(x,t) \).

The Burgers equation is a one–dimensional version of the Navier–Stokes equation. It is often used in gas dynamics since it shows shock wave solutions, and as a simple model of turbulence.

### 10.5.2 The Definition of the Phase Space

In this subsection we define the space of states of the fluid. Once the accessible states of the fluid are known it is possible to construct a master equation governing the probabilistic time evolution of the system.

The first step of the construction of an appropriate phase space consists in the division of physical space into a number of cells centered at the points \( x_i \) which we label by an integer–valued index \( \lambda \in \mathbb{Z} \). For the purpose of general (three dimensional) considerations it suffices to use cubic cells of size \( \delta l^3 \); in practice, shape and size of the cells can be adapted to the confining geometry of the flow. Here the cells are just one–dimensional intervals of length \( \delta l \) since we are considering one space dimensional examples only. Depending on the physical situation under study the number of cells may be finite or infinite. In the stochastic simulation this number is, of course, always finite and fluids of (practically) infinite size have to be modelled by imposing suitable boundary conditions.

We now interpret the velocity field \( u(x,t) \) appearing in the Navier–Stokes equation as an expectation value of a discrete stochastic process \( N^\lambda_u(t) \in \mathbb{Z} \), i. e. we define

\[
\langle u(x,t) \rangle = \delta u \langle N^\lambda_u(t) \rangle .
\]

The above equation provides the connection between the macroscopic and the stochastic description of fluid motion. Within the stochastic description the velocity is a time–dependent random variable \( N^\lambda_u \), i. e. a stochastic process governed by a master equation to be defined below. On the other hand, the velocity on a macroscopic scale, that is on the scale accessible to standard experimental observation, is given by an expectation value, and therefore obeys the deterministic Navier–Stokes equation. The velocity unit \( \delta u \) has been introduced in order to obtain a discrete stochastic process \( N^\lambda_u \). Thus it represents the size of the smallest possible changes of the state of the fluid in the discretized phase space. This means that within our description a positive value \( \delta u \cdot N^\lambda_u \) of the random velocity in the cell \( \lambda \) can be interpreted as the presence of \( N^\lambda_u \) velocity particles each carrying the velocity \( \delta u \). Correspondingly, a negative value \( \delta u \cdot N^\lambda_u \) is to be interpreted as the presence of \( N^\lambda_u \) antiparticles of velocity. Defining the positive and negative part of \( N^\lambda_u \) as

\[
N^\lambda_{u+} := \left\{ N^\lambda_u, \quad N^\lambda_u \geq 0 \right\} , \quad N^\lambda_{u-} := \left\{ 0, \quad N^\lambda_u \leq 0 \right\} ,
\]

we write:

\[
N^\lambda_u = N^\lambda_{u+} + N^\lambda_{u-} .
\]

Thus we see that within our description the mesoscopic state of the fluid may be viewed as a many velocity particle state and is completely fixed by specifying the number \( N^\lambda_u \) of velocity particles in each cell \( \lambda \). Formally, the resulting phase space may be written as

\[
\Gamma = \left\{ (N^\lambda_u)_{\lambda \in \mathbb{Z}} \mid N^\lambda_u \in \mathbb{Z} \right\} .
\]

### 10.5.3 The Construction of the Master Equation

In order to obtain the master equation for the Burgers model we proceed as follows: We show explicitly how stochastic processes can be constructed whose expectation values obey the corresponding terms in the Burgers equation equation. These stochastic processes are defined by a master equation which is a
10.5. NONEQUILIBRIUM THERMODYNAMICS

The time evolution equation for the joint probability distribution \( P = P(N^L_u; t) \) in the phase space \( \Gamma \). From this probability distribution the expectation value for an arbitrary observable \( \hat{O} \) may be evaluated according to

\[
\langle \hat{O} \rangle = \sum \limits_{\{N^L_u\}} O \sum \limits_{\{N^L_u\}} P(N^L_u; t)
\]  

(10.13)

Viscosity as a Many–Particle Random Walk

In this section we are going to describe viscous fluids within our stochastic approach. In the simplest cases, the Navier–Stokes equation containing only viscous forces reads

\[
\frac{\partial u}{\partial t} = \frac{1}{R} \frac{\partial^2 u}{\partial x^2} .
\]  

(10.14)

As in the preceding section we want to define a stochastic process \( N^L_u(t) \) in such a way that its expectation value

\[
u(x,t) = \delta u \langle N^L_u(t) \rangle
\]  

(10.15)

obeys the diffusion–like equation (10.14).

It is well–known that the probability distribution \( \phi(\lambda, t) \in \mathbb{Z} \), of a one particle random walk fulfills in the continuous limit a diffusion equation. More precisely, let us consider a continuous time random walk in one dimension. The probability distribution \( \phi(\lambda, t) \) obeys the following master equation

\[
\frac{\partial \phi}{\partial t} = d \left( \phi_{\lambda+1} - 2\phi_{\lambda} + \phi_{\lambda-1} \right) , \quad d = \text{const} .
\]  

(10.16)

It is easy to see from the above equation that in the limit of infinitesimal small random steps,

\[
\phi(\lambda, t) \rightarrow \phi(x = \delta l \cdot \lambda, t) ,
\]  

(10.17)

the probability distribution \( \phi(x,t) \) is a solution of the diffusion equation

\[
\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2} , \quad D := \lim_{\delta l \to 0} \left( \delta l^2 \frac{d}{dt} \right) .
\]  

(10.18)

The formal analogy of the diffusion equation (10.18) with the one dimensional Navier–Stokes equation (10.14) is obvious. However, there is a fundamental difference between these two equations. The diffusion equation (10.18) describes the time evolution of a probability distribution which is normalized and positive definite. In contrast, the one dimensional Navier–Stokes equation (10.14) is an equation for an expectation value which might be negative and not normalizable. Thus, the stochastic process underlying (10.14) can not be described by a one particle random walk. Therefore, one has to leave the one particle picture. To this end, we consider a collection of independent velocity particles each of which is governed by Eq. (10.16). The state of the resulting collective system Kampen [1992] is characterized by the set of numbers \( N^L_u \) of velocity particles in all cells \( \lambda \). Thus, in this many–particle picture one disregards the identity of the individual particles and is merely interested in the occupation numbers \( N^L_u \) of particles in each cell. (Note, that in the one–particle picture the state of the system is completely specified by giving the location \( \lambda \) of the particle.)

In the framework of the many particle picture the one–particle density \( \phi(\lambda, t) \) is replaced by the many particle probability distribution \( P(\lambda; t) \). The master equation defining the time evolution of \( P(\lambda; t) \) can be obtained from the one–particle master equation by multiplying the one–particle transition rates with the occupation number \( N^L_u \) of cell \( \lambda \),

\[
\frac{\partial P}{\partial t} = \frac{1}{R \delta l^2} \sum \limits_{\lambda} \left( (N^L_{\lambda} + 1)P(\ldots, N^L_{\lambda+1} - 1, N^L_{\lambda}, 1, \ldots) + (N^L_{\lambda} + 1)P(\ldots, N^L_{\lambda}, 1, N^L_{\lambda+1} - 1, \ldots)ight) .
\]
Introducing the shift–operators $E_\lambda$, which operate on an arbitrary function $F$ of the random variables as

$$E_\lambda F(\cdots, N^\lambda_u, \cdots) = F(\cdots, N^\lambda_u + 1, \cdots)$$
$$E_\lambda^{-1} F(\cdots, N^\lambda_u, \cdots) = F(\cdots, N^\lambda_u - 1, \cdots),$$

we can write the master equation as

$$\frac{\partial P}{\partial t} = \frac{1}{R \delta l^2} \sum_\lambda \left[ \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) N^\lambda_u + \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) N^\lambda_{u+} \right] P,$$

where $\delta l$ denotes the cell size. Since the expectation value of $N^\lambda_u$ is a statistical guess of the one–particle density at the point $\lambda$, the velocity field (10.15) obeys the diffusion equation (10.14) as required. Until now we implicitly restricted our attention to the collective random walk of positive velocity particles. Obviously, to be consistent with our general picture of particles and antiparticles of velocity we have to generalize the master equation describing viscous fluids in order to allow for the correct treatment of antiparticles of velocity. Since a one–particle jump to the right is equivalent to a one–antiparticle jump to the left and vice versa, the master equation (10.20) has to be modified in the following way: In order to describe the diffusion of antiparticles the numbers $N^\lambda_u = N^\lambda_{\bar{u}}$ are replaced by their absolute values $|N^\lambda_u| = -N^\lambda_{\bar{u}}$ and the shift operators $E_{\lambda+1} E_\lambda$ by $E_{\lambda+1} E_\lambda^{-1}$. Consequently, if both particles and antiparticles of velocity are present the master equation can be written:

$$\frac{\partial P}{\partial t} = \frac{1}{R \delta l^2} \sum_\lambda \left[ \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) N^\lambda_u + \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) N^\lambda_{\bar{u}} \right] P \tag{10.20}$$

$$- \frac{1}{R \delta l^2} \sum_\lambda \left[ (E_{\lambda-1} E_\lambda^{-1} - 1) N^\lambda_{u+} + (E_{\lambda-1} E_\lambda^{-1} - 1) N^\lambda_{\bar{u}+} \right] P$$

$$=: A_d P \tag{10.21}$$

where we introduced the “diffusion operator” $A_d$. Having presented the master equation defining the time evolution of the stochastic process $\{N_\lambda\}$ we now demonstrate that our requirement explained in subsection $x$ is satisfied, i.e., we show that the expectation value of the stochastic process $\delta u\langle N_\lambda \rangle$ indeed obeys a discrete form of the Burgers equation containing only viscous forces.

More generally, let us first derive an appropriate form for the time evolution equation of the expectation value of an arbitrary function $F(\{N_\lambda\})$ of the stochastic variables. To this end, we consider an expression of the form $\langle E_{\lambda+1}^{-1} E_\lambda F \rangle$. Recalling that the expectation value involves a multiple sum over all integers $N_\lambda$, it is easy to show with the help of the definition of the operators $E_{\lambda}^{\pm}$, Eq. (10.19), and by shifting the summation indices appropriately that the following equation holds

$$\langle E_{\lambda+1}^{-1} E_\lambda F \rangle = \langle F \rangle \tag{10.22}$$

This equation is true, of course, for any other product of shift operators appearing in our master equation (10.21). Thus, these products of shift operators when acting on a function of the stochastic variables have no influence on its expectation value and can, within the angular brackets, always be replaced by the identity. Since the time evolution operator $A$ contains the above products of shift operators only in the form $(EE - 1)$, Eq. (10.22) implies that

$$\langle A F \rangle = \sum_{\{N_\lambda\}} A F P = 0 \tag{10.23}$$

Introducing the commutator (the function $F$ being regarded as a multiplication operator)

$$[F, A] := FA - AF \tag{10.24}$$
and exploiting relation (10.23), equation (10.25) takes the form

$$\frac{\partial}{\partial t} \langle F \rangle = \langle [F, A] \rangle .$$

(10.25)

Thus, the time derivative of the expectation value of a function $F$ of the stochastic variables is equal to the expectation value of the commutator $[F, A]$. For the special case of the first moments of the stochastic process $\delta u N$, we have

$$\delta u \frac{\partial}{\partial t} \langle N_s \rangle = \delta u \langle [N_s, A] \rangle .$$

(10.26)

In order to determine the commutator $[N_s, A]$ we proceed by evaluating some useful the commutators. Let us begin with the calculation of the commutator $[N_s, E_{s'}]$ by looking at how it operators on an arbitrary function of the stochastic variables. Let us consider first the case that $s = s'$. We have

$$[N_s, E_{s'}] F(\ldots, N_s, \ldots) = N_s E_{s'} F(\ldots, N_s, \ldots) - E_{s'} N_s F(\ldots, N_s, \ldots) = N_s F(\ldots, N_s + 1, \ldots) - (N_s + 1) F(\ldots, N_s + 1, \ldots) = -E_{s'} F(\ldots, N_s, \ldots).$$

For the case that $s \neq s'$ the calculation is similar

$$[N_s, E_{s'}] F(\ldots, N_s, \ldots) = N_s E_{s'} F(\ldots, N_s, \ldots) - E_{s'} N_s F(\ldots, N_s, \ldots) = N_s F(\ldots, N_s, N_{s'} + 1, \ldots) - N_s F(\ldots, N_s, N_{s'} + 1, \ldots) = 0.$$

Summarizing, we can write

$$[N_s, E_{s'}] = -E_{s'} \delta_{s,s'}.$$

Similarly, we can show that

$$[N_s, E_{s'}^{-1}] = E_{s'}^{-1} \delta_{s,s'}.$$ Another useful commutator is

$$[N_s, E_{s'+1}^{-1} E_{s'}] = [N_s, E_{s'+1}^{-1}] E_{s'} + E_{s'+1}^{-1} [N_s, E_{s'}] = E_{s'+1}^{-1} E_{s'} \delta_{s'+1,s} - E_{s'+1}^{-1} E_{s'} \delta_{s',s} = E_{s'+1}^{-1} E_{s'} (\delta_{s'+1,s} - \delta_{s',s}).$$

In order to determine the equation of motion for the expectation value of random velocity variable we thus have to compute first the commutator $[N_s, A_s]$. We obtain

$$[N_s, A_s] = \frac{1}{\mathcal{R} \mathcal{S}^2} \sum_{s'} \left[ [N_s, E_{s'+1}^{-1}] E_{s'} (N_s^+ - N_{s-1}^-) + [N_s, E_{s'+1}^{-1}] (N_s^+ - N_{s-1}^-) \right]$$

$$= \frac{1}{\mathcal{R} \mathcal{S}^2} \sum_{s'} \left[ E_{s'+1}^{-1} E_{s'} (\delta_{s'+1,s} - \delta_{s',s}) (N_s^+ - N_{s-1}^-) + E_{s'+1}^{-1} E_{s'} (\delta_{s'+1,s} - \delta_{s',s}) (N_s^+ - N_{s-1}^-) \right]$$

$$= \frac{1}{\mathcal{R} \mathcal{S}^2} \left\{ E_{s'}^{-1} E_{s'+1} (N_s^+ - N_{s-1}^-) - E_{s'-1}^{-1} E_{s} (N_s^+ - N_{s-1}^-) + E_{s'}^{-1} E_{s'-1} (N_s^+ - N_{s-1}^-) - E_{s'-1}^{-1} E_{s} (N_s^+ - N_{s-1}^-) \right\}. $$

Before taking the expectation value of the above expression we have to notice that whenever in the sum $\sum_{\{N_s\}}$ the shift operators appear on the extreme left they can be replaced by 1. This is easy to show with
the help of the definition of the shift operators and by shifting appropriately the summation indices. With the help of this trick it is evident that we have

\[
\delta u \frac{\partial}{\partial t} \langle N_\lambda \rangle = v \delta u \frac{\partial}{\partial t} (N_{\lambda+1}^+-2N_{\lambda+1}^-+N_{\lambda-1}^+ + N_{\lambda-1}^- - 2N_{\lambda}^- + N_{\lambda-1}^-).
\]  

(10.27)

Recalling that \( N_\lambda = N_{\lambda}^+ + N_{\lambda}^- \) the time evolution equation of the first moment of the stochastic process \( N_\lambda \) can be written as

\[
\delta u \frac{\partial}{\partial t} \langle N_\lambda \rangle = v \delta u \frac{\partial}{\partial t} (N_{\lambda+1} - 2N_{\lambda} + N_{\lambda-1}) .
\]

(10.28)

Since, the equation is linear in \( N_\lambda \) we have, of course, \( v_\lambda = \delta u \langle N_\lambda \rangle \) and hence we immediately obtain a discretized version of the Burgers equation without convective terms.

\[
\frac{\partial}{\partial t} v_\lambda (t) = v \frac{v_{\lambda+1} - 2v_\lambda + v_{\lambda-1}}{\delta t^2} - \frac{1}{2} \frac{v_{\lambda+1}^2 - v_{\lambda-1}^2}{2 \delta t} ,
\]

(10.29)

which, in turn, leads to Burgers’ equation (??) in the continuum limit \( \delta t \rightarrow 0 \).

The Nonlinear Convection Term

Up to now, we have shown how to deal with viscous diffusion. However, it is of great importance to include within the stochastic theory nonlinear interactions which, for example, enter the Navier–Stokes equation through the nonlinear inertial term \((\vec{u} \cdot \nabla) \vec{u}\). It is a remarkable fact that interpreting the inertial term as a nonlinear convection term a stochastic interpretation in terms of one–particle jumps can be found. As will be shown here, again, the many–particle picture is absolutely necessary.

As before we are looking for the stochastic process \( N_u^\lambda (t) \) the expectation value of which obeys Burgers’ equation. Obviously, the stochastic process underlying the nonlinear convection term is fundamentally different from the previously introduced stochastic processes: It is not a random walk of a collection of independent particles. The construction of the Hydrostochastic form of the convection term will now be given heuristically within the many particle picture.

Let us consider what happens in a specific cell \( \lambda \) occupied by \( N_u^\lambda \geq 0 \) velocity particles. Within the stochastic approach convection may be modelled as a jump process from cell \( \lambda \) to cell \( (\lambda + 1) \). Thus, the stochastic process is completely specified by giving the transition rates for these elementary jumps. The probability for the jump of a specific particle situated in cell \( \lambda \) is proportional to the velocity at \( \lambda \) and, hence, proportional to the number of velocity particles in cell \( \lambda \). Consequently, the total transition rate is proportional to the number of pairs of velocity particles in the cell \( \lambda \), i. e. to \( N_u^\lambda \cdot (N_u^\lambda - 1)/2 \). On dimensional grounds the proportionality factor is found to be \( \delta u / \delta l \) (note that transition rates have the dimension of an inverse time) and we obtain the following master equation for the convection of velocity particles:

\[
\frac{\partial P}{\partial t} = \frac{\delta u}{\delta l} \sum_\lambda \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) \frac{1}{2} N_u^\lambda (N_u^\lambda - 1) P .
\]

(10.30)

It can be shown that the above form of the master equation for the convection term leads to a discretization of the differential operator \( \delta u / \delta x \) which is of order \( \delta l \). In order to obtain a discretization of this differential operator which is of order \( \delta l^2 \) the following symmetrized form of the master equation will be used:

\[
\frac{\partial P}{\partial t} = \frac{\delta u}{2 \delta l} \left\{ \sum_\lambda \left( E_{\lambda+1}^{-1} E_\lambda - 1 \right) \frac{1}{2} N_u^\lambda (N_u^\lambda - 1) P \\
+ \sum_\lambda \left( E_{\lambda}^{-1} E_{\lambda-1} - 1 \right) \frac{1}{2} N_u^\lambda (N_u^\lambda - 1) P \right\} =: A \ P ,
\]

(10.31)
where we abbreviated the effect of the right hand side by defining the “convection operator” $A_c$.

Constructing Eq. (10.31) we assumed that $N^h_\lambda \geq 0$. The general master equation describing both the presence of velocity particles and antiparticles can now easily be derived. Let us assume, that cell $\lambda$ contains $N^h_\lambda = -N^{\bar{h}}_\lambda$ antiparticles of velocity. The elementary jumps of the antiparticles can be obtained from those of the particles of velocity by simply reversing the directions of the jumps, i.e. all antiparticles jump to the left. However, since the process of an antiparticle jumping to the left is identical to the process of a particle jumping to the right, the above master equation describes the combined convection of particles and antiparticles of velocity if $N^h_\lambda$ is replaced by $|N^h_\lambda|$.

Summarizing the master equation of the Burgers model reads

$$\frac{\partial P}{\partial t} = \{A_d + A_c\} P$$

$$= \frac{1}{R \delta t^2} \sum_\lambda \left[ \left( E_{\lambda-1}^e E_{\lambda+1} - 1 \right) N^h_{\lambda+1} + \left( E_{\lambda+1}^e E_{\lambda-1} - 1 \right) N^{\bar{h}}_{\lambda+1} \right] P$$

$$- \frac{1}{R \delta t^2} \sum_\lambda \left[ \left( E_{\lambda-1}^e E_{\lambda+1} - 1 \right) N^{\bar{h}}_{\lambda-1} + \left( E_{\lambda+1}^e E_{\lambda-1} - 1 \right) N^h_{\lambda-1} \right] P$$

$$+ \frac{\delta u}{2 \delta t} \left\{ \sum_\lambda \left( E_{\lambda+1}^e E_{\lambda-1} - 1 \right) \frac{1}{2} \left| N^h_\lambda \right| \left| \left( N^h_\lambda - 1 \right) P \right| \right\}.$$  \hspace{1cm} (10.32)

Let us now demonstrate how the macroscopic equation may be derived from the above equation. To this end we have only to compute the commutator $[N_{\lambda}, A_c]$ with the help of (??)

$$[N_{\lambda}, A_c] = \frac{\delta u}{4 \delta t} \sum_{i,j} [N_{\lambda}, E_{\lambda+1}^e E_{\lambda-1}^e] (N^2_{\lambda} + N^{2}_{\lambda+1})$$

$$= \frac{\delta u}{4 \delta t} \sum_{i,j} E_{\lambda+1}^e E_{\lambda-1}^e \left( \delta_{\lambda+1,i} - \delta_{\lambda-1,i} \right) (N^2_{\lambda} + N^{2}_{\lambda+1})$$

$$= \frac{\delta u}{4 \delta t} \left\{ E_{\lambda+1}^e E_{\lambda-1}^e \left( N^2_{\lambda} + N^{2}_{\lambda+1} \right) + E_{\lambda+1}^e E_{\lambda-1}^e \left( N^2_{\lambda} + N^{2}_{\lambda+1} \right) \right\}.$$  \hspace{1cm} for sufficiently small $\delta t$ the number of velocity particles $N^h_\lambda$ becomes large one expects that fluctuations are small and, therefore, that the approximation

$$\langle \left( \frac{N^h_\lambda}{\delta t} \right)^2 \rangle \approx \langle N^2_{\lambda} \rangle ^2$$  \hspace{1cm} (10.33)

holds to a sufficient degree of accuracy. Eq. (??) may then be written as an equation for $u_\lambda$,

$$\frac{\partial u_\lambda}{\partial t} + \frac{1}{2} \frac{u^2_{\lambda+1} - u^2_{\lambda-1}}{2 \delta t} = \frac{1}{R} \frac{u_{\lambda+1} - 2u_\lambda + u_{\lambda-1}}{\delta t^2}.$$  \hspace{1cm} (10.34)

Obviously, this equation is nothing but the discretized version of the Burgers equation (??) which emerges in the continuum limit $\delta t \rightarrow 0$. In deriving the above macroscopic equation for the expectation value $u_\lambda$ we neglected, of course, all higher moments of the stochastic process $N^h_\lambda$. As is well–known the above master equation, defining a nonlinear stochastic process, leads to an infinite dimensional system of coupled differential equations for the moments. Thus, in order to derive more rigorously from our master equation the macroscopic equation and to investigate the dynamics and influence of fluctuations one has to employ a more systematic method. Such a method is provided by the well–known $\Omega$–expansion Kampen [1992]. Applying this expansion to the master equations of Hydrostochastics reveals that, in fact, the macroscopic equations are equivalent to the equations of fluid dynamics. Furthermore, it can be shown that within the linear noise approximation the fluctuations superimposed on the macroscopic dynamics can be identified with those fluctuations derived from the theory of fluctuating hydrodynamics. Thus, the stochastic dynamics implied by our master equation formulation can, indeed, be given a clear physical interpretation as we will see in one of the next sections.
10.5.4 The Stochastic Simulation Algorithm

It is the aim of this section to show that the master equation formulation of the Burgers equation naturally leads to very transparent numerical simulation algorithms. Basically, the stochastic simulation method generates an ensemble of realizations of the stochastic process \( \{N_s\} \). From this ensemble the physical quantities of interest, e.g. the mean velocity field and correlation functions, can then be evaluated as ensemble averages. The stochastic simulation algorithm consists of 3 basic steps:

1. Let us assume that at time \( t \) the state of the system is given by \( \{N_s(t)\} \). In the first step, the time \( t + \tau \) of the next transition is determined. Our algorithm uses a stochastic time step \( \tau \) to be evaluated as follows. Obviously, the total transition rate \( W(\{N_s(t)\}) \) is given by

\[
W(\{N_s\}) = \sum_{j=0}^{M} w_j = \sum_{j=0}^{M} \left( \frac{2v}{\delta t^2} |N_j| + \frac{1}{2} \frac{\delta u}{\delta t} N_j^2 \right),
\]

where \( w_j \) denotes the rate for those transitions in which \( N_j \) is changed. Consequently, the probability for the next transition to occur somewhere in the system within the infinitesimal time step \( d\tau \) is \( W(\{N_s(t)\})d\tau \). The total transition rate \( W \) determines the waiting time distribution, i.e. the probability distribution of the time \( \tau \) the system remains in the state \( \{N_s(t)\} \). In the stochastic simulation the random number \( \tau \) which determines the time for the next transition to occur can be obtained by the inversion method with the help of the following formula

\[
\tau = -\frac{1}{W(N^\lambda(i))} \ln \eta,
\]

where \( \eta \) is a uniformly distributed random number on the interval \([0, 1]\).

2. Having determined the transition time we have to perform a specific transition, i.e. we have to determine the new state \( \{N_s(t+\tau)\} \) of the system. To this end, one chooses according to the relative probabilities \( w_j/W \) a certain cell. Once a definite cell \( \lambda \), say, has been chosen the new state of the system is to be selected from the following possibilities:

1. Diffusive transitions:

\[
\begin{align*}
N_k &\rightarrow N_k - s \\
N_{k+1} &\rightarrow N_{k+1} + s \\
&\text{probability } \frac{v}{\delta t^2} \frac{|N_k|}{w_k}
\end{align*}
\]

\[
\begin{align*}
N_k &\rightarrow N_k - s \\
N_{k-1} &\rightarrow N_{k-1} + s \\
&\text{probability } \frac{v}{\delta t^2} \frac{|N_k|}{w_k}
\end{align*}
\]

where

\[
s = \begin{cases} 
+1 & \text{for } N_k \geq 0 \\
-1 & \text{for } N_k < 0 
\end{cases}
\]

2. Convective transition:

\[
\begin{align*}
N_k &\rightarrow N_k - 1 \\
N_{k+1} &\rightarrow N_{k+1} + 1 \\
&\text{probability } \frac{\delta u}{4\delta t} \frac{N_k^2 + N_{k+1}^2}{w_k}
\end{align*}
\]

Note that each of these transitions corresponds to one of the 5 terms in our master equation (Equations 10.36) and that the transition probabilities given above add up to 1. Performing one of these transitions yields the new state \( \{N_s(t+\tau)\} \).

3. The complete trajectory of the stochastic process can be determined by repeating the above scheme until a desired final time is reached. Finally, by generating a large number \( S \) of realizations of the stochastic process \( \{N_s(t)\}_j \), \( j = 1, \ldots, S \), one can evaluate the interesting quantities as ensemble averages.

3. The complete trajectory of the stochastic process can be determined by repeating the above scheme until a desired final time is reached. Finally, by generating a large number \( S \) of realizations of the stochastic process \( \{N_s(t)\}_j \), \( j = 1, \ldots, S \), one can evaluate the interesting quantities as ensemble averages.
The simplicity of the above scheme makes clear that the numerical simulation of a stochastic process defined by our multivariate master equation is straightforward. In the following we shall exemplify this by showing some stochastic simulations of shock wave solutions of Burgers equation.

Originally Burgers proposed the equation (10.36) as a simple one–dimensional model of homogeneous turbulence. The main features of the Navier–Stokes equations are retained in the above equation. The nonlinearity has the same structure as in the Navier–Stokes equation, and the dissipative term is also of the same type. Only the pressure term is missing, so that one has to expect a relaxation of turbulence with time. The model also lacks an equation of continuity, so that it describes in practice a one–dimensional compressible flow. The Burgers equation is particularly appealing because the analytical solutions of the initial value problem is known Whitham [1974]. These solutions represent, for example, nonlinear wave solutions like shock waves and “humps”. Thus, Burgers’ equation is interesting as it makes possible the study of the interplay of nonlinear propagation and viscous diffusion.

**Shock Waves**

The first example we are going to treat is the shock wave solution of Burgers’ equation. The shock wave solution is obtained for the following initial condition

\[ u(x,0) = u_0(x) = \begin{cases} 
  1 & , \ x < 0 \\
  0 & , \ x > 0 
\end{cases}. \tag{10.37} \]

The diffusion and the convection of this initial step are described by the following time–dependent solution

\[ u(x,t) = \frac{1}{1 + h(x,t) \exp[R(x-t/2)/2]} , \tag{10.38} \]

where the function \( h(x,t) \) is defined by

\[ h(x,t) = \frac{\text{erfc}(-\sqrt{R/4t}x)}{\text{erfc}(-\sqrt{R/4t} (x-t))} . \tag{10.39} \]

The function \( \text{erfc}(x) \) is known as the conjugated error function, and is given by

\[ \text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} dy \exp(-y^2) . \tag{10.40} \]

As a first example, we depict in Fig. 1 a stochastic simulation of a shock wave for the Reynolds number \( R \equiv \nu^{-1} = 100 \) which develops from the step function initial condition

\[ v(x,0) = \begin{cases} 
  1 & , \ x < 0 \\
  0 & , \ x > 0 
\end{cases} . \tag{10.41} \]

This initial condition corresponds to the initial configuration of the stochastic variables \( \{N_\lambda\} \) given by

\[ N_\lambda(0) = \begin{cases} 
  \text{int}(1/\delta u) & , \ \lambda < 0 \\
  0 & , \ \lambda > 0 
\end{cases} . \tag{10.42} \]

In Fig. 1 the exact analytical solution of Burgers’ equation is represented by the lines, whereas the symbols mark the results of the simulation. Fig. 2 shows a simulation obtained for the same initial condition and the same numerical parameters; however, the Reynolds number was set equal to infinity, i. e. \( \nu = 0 \). This figure clearly demonstrates the great stability of the stochastic simulation algorithm.

As a final example we depict in Fig. 3 the simulation of the confluence of two shocks propagating with different velocities to the right \( (R = 100) \). Again the analytical solution for this problem Whitham [1974] is indicated by the lines.
10.6 Hydrodynamic Fluctuations

In section 2 we proposed a master equation for the stochastic description of Burgers’ equation. This master equation has been constructed in such a way that the expectation value of the stochastic process (10.9) obeys in the limit $\delta u \to 0$, i.e., in the limit of large numbers $N_\lambda$, a discretized version of Burgers’ equation. By some stochastic simulations we demonstrated in Section 3 that this yields a simple numerical approach to fluid dynamical computations. Of course, the stochastic description by a master equation implies fluctuations around the macroscopic value of the velocity field governed by Burgers’ equation.

In this section we present a systematic analysis of the stochastic process $N_\lambda$. This analysis proves that, in fact, the macroscopic equation pertaining to $N_\lambda$ is nothing but Burgers’ equation. Furthermore, it is shown that the fluctuations around the macroscopic velocity which are predicted by our master equation, are those expected from the theory of hydrodynamic fluctuations. The basic tool of the following analysis will be the well-known van Kampen’s $\Omega$–expansion [1992]. Basically, this method is a consistent expansion of the master equation in powers of a small quantity $\Omega^{-1}$ which has to be identified from the physical parameters of the model under study. The first step in our analysis is, thus, to identify the appropriate small quantity in our master equation which allows for an $\Omega$–expansion. Recall that the construction of the master equation was based on the equation

$$v(x,t) = \delta u \langle N_\lambda \rangle \ .$$

(10.43)

This equation connects the stochastic process $N_\lambda$ to the mean velocity $v(x,t)$ which obeys Burgers’ equation. It follows from this equation that $\delta u^{-1}$ is approximately proportional to $N_\lambda$. Since in the limit of infinitely large numbers $N_\lambda$ one expects a well-defined macroscopic law to emerge from the mesoscopic formulation the quantity $\delta u$ turns out to be an appropriate small parameter for the $\Omega$–expansion, i.e., in the following we study the limit

$$\Omega^{-1} = \delta u \to 0 \ .$$

(10.44)

Assuming that the fluctuations of $N_\lambda$ are small it is natural to split the stochastic process $\delta u N_\lambda$ into two parts: the first part is a macroscopic variable $v_\lambda$ which satisfies a deterministic evolution equation, whereas the second part $w_\lambda = \delta u^{1/2} \eta_\lambda$ describes the small stochastic deviations from this deterministic time evolution:

$$\delta u N_\lambda = v_\lambda + \delta u^{1/2} \eta_\lambda \equiv v_\lambda + w_\lambda \ .$$

(10.45)

In this equation $v_\lambda$ is introduced as an intensive quantity which is independent of $\delta u$. Also, the new stochastic variable $\eta_\lambda$ is assumed to be independent of $\delta u$ to leading order. This means that equation (10.45) expresses the expectation that the fluctuations $w_\lambda$ around the macroscopic values are proportional to $\delta u^{1/2}$.

We are now in the position to perform the $\Omega$–expansion along the lines given by van Kampen. Since the two stochastic variables $\eta_\lambda$ we first introduce the transformed probability distribution $\Pi$

$$P\left(\{N_\lambda\}; t\right) = \delta u^{-M+1/2} \Pi\left(\{\eta_\lambda\}; t\right)$$

(10.46)

for which we have

$$\frac{\partial P}{\partial t} = \frac{\partial \Pi}{\partial t} + \sum_\lambda \frac{\partial \Pi}{\partial \eta_\lambda} \frac{\partial \eta_\lambda}{\partial t} = \frac{\partial \Pi}{\partial \eta_\lambda} \frac{\partial v_\lambda}{\partial t} = \delta u^{-1/2} \sum_\lambda \frac{\partial \Pi}{\partial \eta_\lambda} \frac{\partial v_\lambda}{\partial \eta_\lambda} \ .$$

(10.47)

We also expand the shift operators in powers of $\delta u$:

$$E_\lambda = 1 + \delta u^{1/2} \frac{\partial}{\partial \eta_\lambda} + \frac{1}{2} \delta u \frac{\partial^2}{\partial \eta_\lambda^2} + \cdots \ .$$

(10.48)

Inserting these expressions into the master equation for $\Pi$ and collecting terms of the same order in $\delta u$ we
finally obtain (including terms of order $\delta u^0$):

\[
\frac{\partial \Pi}{\partial t} = \sqrt{\frac{1}{\delta u}} \sum_\lambda \left\{ \frac{\partial v_\lambda}{\partial t} + \frac{v_{\lambda+1}^2 - v_{\lambda-1}^2}{2\delta l} - \frac{v_{\lambda+1} + v_{\lambda-1} - 2v_\lambda}{\delta l^2} \right\} \frac{\partial \Pi}{\partial \eta_\lambda} 
+ \sum_\lambda \frac{\partial}{\partial \eta_\lambda} \left\{ \frac{\eta_{\lambda+1} + \eta_{\lambda-1} - 2\eta_\lambda}{\delta l^2} - \frac{\eta_{\lambda+1} \eta_{\lambda+1} - \eta_{\lambda-1} \eta_{\lambda-1}}{2\delta l} \right\} \Pi 
+ \frac{1}{2} \sum_{\lambda \mu} \left\{ D_{\lambda \mu} \frac{\partial^2}{\partial \eta_\lambda \partial \eta_\mu} \right\} \Pi .
\]

(10.49)

Let us explain the meaning of the various terms in this equation. The first sum is of order $\delta u^{-1/2}$ and, therefore, diverges in the limit $\delta u \to 0$ unless one imposes the condition that the expressions within the curly brackets of the first sum vanishes for each $\lambda$. In fact, until now the dynamics of the macroscopic variables $v_\lambda$ has not been specified. Thus, in order to obtain a well-defined $\delta u$-expansion we now require that the curly brackets of the first sum vanishes identically. This condition is easily seen to be equivalent to the requirement that $v_\lambda$ obeys the (discretized) Burgers equation (10.50).

The remaining terms of the master equation for $\Pi$ constitute a linear Fokker–Planck equation which has an obvious physical interpretation. The drift term of this Fokker–Planck equation is obtained by linearizing Burgers’ equation around the macroscopic solution $v_\lambda$. Thus, the drift term governs the time evolution of small perturbations of the macroscopic variable.

The last term of equation (10.49) describes a multivariate diffusion process with diffusion matrix

\[
D_{\lambda \mu} := (d_{\lambda} + d_{\lambda-1}) \delta_{\lambda \mu} - d_{\lambda} d_{\lambda+1, \mu} - d_{\mu} d_{\lambda, \mu+1} 
\]

(10.50)

where

\[
d_{\lambda} := \sqrt{\tilde{v}_{\lambda} + \tilde{v}_{\lambda+1}} + \frac{\tilde{v}_{\lambda}^2 + \tilde{v}_{\lambda+1}^2}{4\delta l} , \quad \tilde{v}_{\lambda} := \delta u \langle N_{\lambda} \rangle .
\]

(10.51)

Since the above Fokker–Planck equation is linear its general solution represents a nonstationary multivariate Gaussian process which is completely characterized by its mean values and variances. The latter are defined by

\[
\langle \eta_\lambda(t) \eta_\mu(t) \rangle = \int D\eta \eta_\lambda \eta_\mu \Pi(\{\eta_\lambda\}; t) ,
\]

(10.52)

where

\[
D\eta := d\eta_0 d\eta_1 d\eta_2 \ldots d\eta_M .
\]

(10.53)

It is now straightforward to write down the dynamic equations for the correlation function $\langle \eta_\lambda \eta_\mu \rangle$. This will be done for the stochastic variables $w_\lambda = \delta u^{1/2} \eta_\lambda$ which represent the random velocity fluctuations. Since the equations of fluctuating hydrodynamics are usually formulated in a continuous notation we write in the continuum limit $\delta l \to 0$:

\[
\langle w_\lambda(t) w_\mu(t) \rangle \to \langle w(y, t) w(x, t) \rangle .
\]

(10.54)

We then obtain from the Fokker–Planck equation (10.49)

\[
\frac{\partial}{\partial t} \langle w(y, t) w(x, t) \rangle = \left( \nabla_y \cdot \frac{\partial}{\partial y} v(y, t) \right) \langle w(y, t) w(x, t) \rangle 
+ \left( \nabla_x \cdot \frac{\partial}{\partial x} v(x, t) \right) \langle w(y, t) w(x, t) \rangle 
+ \delta u D_{xy} ,
\]

(10.55)
where we introduced the continuum limit of the diffusion matrix $D_{\delta}$ as
\[
D_{\delta} := -2v \frac{\partial}{\partial x} \frac{\partial}{\partial y} \delta(y-x).
\] (10.56)

It should be noted that the $\Omega$–expansion as presented above is based on the assumption that the stochastic process $\delta u_N$ can be decomposed into a macroscopic part of order 1 and a fluctuating part of order $\delta u^{1/2}$. This implies, of course, that we assume that the solutions of the macroscopic equation are stable. This condition guarantees that the fluctuations around the macroscopic variables are bounded Kampen [1992].

Let us make an important remark. Our master equation (44) describes both the viscous friction and the nonlinear inertial term of Burgers’ equation by stochastic processes. Thus, our stochastic formulation treats the nonlinear convection term and the dissipative viscosity term on an equal footing. However, it is important to note that in the continuum limit the nonlinear term does not contribute to the fluctuations. This fact can be clearly seen from the continuum limit (10.56) of the diffusion matrix (10.50).

The above equation for the correlation function $\langle w w \rangle$ now allows to identify the physical meaning of the mesoscopic velocity and length scales $\delta u$ and $\delta l$ which have been introduced within our stochastic description. Let us consider the stationary and spatially homogeneous velocity fluctuations around a state with constant macroscopic velocity $v$ and mass density $\rho$. With these assumptions we obtain the following stationary solution (neglecting boundary effects):
\[
\langle w(y,t)w(x,t) \rangle = \bar{v} \delta u \delta l \delta(y-x).
\] (10.57)

This explicit expression for the velocity fluctuations enables us to relate the mesoscopic quantities $\delta u$ and $\delta l$ to thermodynamic state variables. Recall, that the stochastic process $\delta u N_{\delta}(t)$ represents, from a microscopic viewpoint, the average velocity of the real fluid particles of mass $m$ in cell $\delta l$. We now assume the fluid particles in each cell to be in a local thermodynamic equilibrium state. It is then natural to identify the magnitude of the variance of the random velocity $w$ with the thermodynamic fluctuations $(\Delta u)^2$ of this average velocity.

From the Maxwell distribution at temperature $T$ we obtain
\[
(\Delta u)^2 = \frac{kT}{\rho \delta l},
\] (10.58)
where $k$ denotes Boltzmann’s constant. Thus we have
\[
\bar{v} \delta u = \frac{kT}{\rho \delta l}.
\] (10.59)

Equation (10.59) represents a type of fluctuation–dissipation relation of our mesoscopic theory. Note that both mesoscopic parameters, $\delta u$ and $\delta l$, enter the equation (10.59). Thus, $\delta u$ is fixed once a length scale $\delta l$ has been chosen. Of course, the magnitude of $\delta l$ is determined by the very assumptions of local thermodynamic equilibrium which usually enter non–equilibrium thermodynamics: On the one hand, $\delta l$ has to be chosen in such a way that the number of fluid particles contained in a cell is large enough to allow for a reasonable thermodynamic description. On the other hand, the size of $\delta l$ is limited by the requirement that the variation of the various quantities over the cells is small compared to the magnitude of these quantities.

We have seen that the dynamics of the velocity fluctuations is governed by a linear Fokker–Planck equation. Of course, the dynamics of the fluctuations can be described as well by an equivalent Langevin equation. Invoking our fluctuation–dissipation relation (10.59) the Langevin equation for the stochastic variable $w(x,t)$ reads
\[
\frac{\partial}{\partial t} w(x,t) = \left( v \frac{\partial^2}{\partial x^2} - v \frac{\partial}{\partial x} \right) w(x,t) - \frac{\partial}{\partial x} \xi(x,t).
\] (10.60)

Interpreting $\xi(x,t)$ as the random momentum flux density, equation (10.60) is precisely of the form of the corresponding equation in fluctuating hydrodynamics (see, for example, [22]). The correlation function of
the random momentum flux can be determined from the diffusion matrix and our fluctuation–dissipation relation. One finds

$$\langle \xi(y,t)\xi(x,t') \rangle = \frac{2kT\nu}{\rho} \delta(y-x)\delta(t-t') .$$  \hspace{1cm} (10.61)

Let us sum up what has been achieved in this section. By means of the simple example of Burgers’ equation we have established the connection of our multivariate master equation description of fluid dynamics to the theory of fluctuations in fluids. In other words, the master equation describes both the macroscopic dynamics and the dynamics of the fluctuations.

### 10.6.1 Couette Flow and Poiseuille Flow

### 10.7 Homogeneous Turbulence: The Burgers Equation

#### 10.7.1 Homogeneous Turbulence

A turbulent fluid is usually characterized by irregular and disordered temporal variations of the velocity field at each point of the fluid and conversely the variations of the velocity vary irregularly from point to point at a fixed time. Up to now there is no complete quantitative theory of turbulence, but there are well accepted qualitative results.

**The energy cascade** The main aspect of the turbulence problem is the presence in a turbulent fluid of different length scales. Looking at a turbulent fluid you will notice vortices with small and large dimensions. For very large Reynolds number there are very turbulent elements on very large and on very small length scales. The main role is played by the large vortices, which are of the order of magnitude of the exterior dimensions of the system under consideration. Let's call this length scale $l$. The order of magnitude of the velocity is of the order of the variation of the mean velocity $\Delta u$ on the length scale $l$. The large turbulent elements contain most of the kinetic energy of the turbulent fluid. Small vortices contain only a small part of the kinetic energy.

The general properties of a given flow are characterized by the Reynolds number

$$R = \frac{ul}{\nu} .$$

A better characterization is found by introducing Reynolds numbers for turbulent elements of different size. Denoting by $\lambda$ the size of the turbulent element, by $\nu_{\lambda}$ the order of magnitude of the velocity in that turbulent element we can define

$$R_{\lambda} \approx \frac{\nu_{\lambda} \lambda}{\nu} .$$

For large values of $R$ also the Reynolds numbers $R_{\lambda}$ of the large vortices ($\lambda \approx l$) are large. Since a large $R$ implies a small $\nu$, we may follow that the viscosity does not play a great role for the large turbulent elements. Hence it is not to be expected that dissipation will play a great role for these vortices. Viscosity will affect only the smallest vortices ($\lambda_0$), for which $R_{\lambda} \approx 1$. And it is here that the dissipation takes place.

The physical picture of what is going on in a turbulent fluid is due to L. Richardson (1922). The energy flows from the large vortices into the small ones. This process occurs without dissipation. The energy flow is dissipated in the smallest vortices, where kinetic energy is converted into heat. To make this process “stationary” it is of course necessary to add energy in the large length scales. This picture is called the energy cascade. As we will see, a major aspect of this picture is that all quantities involved which belong to turbulent elements with $\lambda > \lambda_0$ do not depend upon the viscosity.

**The orders of magnitude of the energy dissipation** The energy dissipation is defined as the mean energy which is dissipated pro time unit and pro mass unit in a turbulent fluid. The order of magnitude of the energy dissipation may be determined with the help of quantities which are characteristic of the large
vortices. These quantities are the density of the fluid $\rho$, the linear dimension $l$, and the velocity $\Delta u$. Only a combination of these 3 quantities has the dimension of a dissipation $([\varepsilon] = J/(kg \cdot s) = m^2/s^3)$

$$\varepsilon \approx \frac{(\Delta u)^3}{l}.$$ 

A turbulent flow can be described qualitative as a fluid with a turbulent viscosity $\nu_{\text{turb}} \neq \nu$. The only dimensionless quantity with the dimension of a viscosity which may be constructed with the help of $\rho$, $\Delta u$, and $l$ is $\nu_{\text{turb}} \approx l\Delta u$.

We have

$$\frac{\nu_{\text{turb}}}{\nu} \approx R_c.$$

Note that $\nu_{\text{turb}}$ gets larger for increasing Reynolds number $R$. In other words, we can write the energy dissipation also in the form

$$\varepsilon \approx \left(\frac{\Delta u}{l}\right)^2$$

in accordance with the usual definition of viscosity.

**The law of Kolmogorov** Now we want to look at the properties of the turbulence on the length scale $\lambda$, which is assumed to be small with respect to $l$. We look at the bulk of the fluid and neglect boundary effects. In these domains it is safe to assume that the turbulence is homogeneous and isotropic. We will now deduce with the help of similarity arguments some fundamental results (Kolmogorov, 1941; Obuchow, 1941). The local properties of the turbulence are determined for $l \gg \lambda \gg \lambda_0$ by $\lambda$, the density $\rho$ and the energy dissipation $\varepsilon$. The local properties do not depend upon $l$ and $\Delta u$, which are relevant for the flow as a whole. Furthermore, they do not depend upon $\nu$.

The variation of the velocity $v_{\lambda}$ on a length scale $\lambda$ is easily guessed by considering that the only quantity with the dimension of a velocity which can be formed with the help of $\rho$, $\varepsilon$ and $\lambda$ is

$$v_{\lambda} \approx (\varepsilon \cdot \lambda)^{1/3}. \quad (10.62)$$

This is the famous law of Kolmogorow and Obuchow, which states that the variation of the velocity on a distance of the order $\lambda$ is proportional to the cubic root of the distance $\lambda$. The velocity $v_{\lambda}$ may be regarded as the velocity of a turbulent element of dimension $\lambda$.

Similarly, the variation of the velocity at a fixed point during the time $\tau$, which is smaller than characteristic time scale $T \approx l/\Delta u$, may be estimated by inserting $\Delta t$ for $\lambda$ in the above equation to be

$$v_{\tau} \approx (\varepsilon \cdot \tau \cdot \Delta u)^{1/3}. \quad (10.63)$$

With the help of

$$\varepsilon \approx \frac{(\Delta u)^3}{l},$$

Eqs. (10.62) and (10.63) can be brought in the following suggestive form

$$\frac{v_{\lambda}}{\Delta u} \approx \left(\frac{\lambda}{T}\right)^{1/3}$$

and, respectively,

$$\frac{v_{\tau}}{\Delta u} \approx \left(\frac{\tau}{T}\right)^{1/3}.$$ 

The two above expressions make evident the self–similarity property of local turbulence. The characteristic quantities differ only through the scales in which the length and the times are measured.
Estimation of the dimensions \( \lambda_0 \) of the smallest turbulent elements  On the smallest "inner" length scale \( \lambda_0 \) of turbulence the effects of dissipation are important. The local Reynolds number we find

\[
R_k \approx \frac{v \cdot \lambda}{\nu} = \frac{\rho \cdot \lambda^{4/3}}{\nu^{1/3}} \approx R \left( \frac{\lambda}{l} \right)^{4/3}.
\]

The smallest length scale \( \lambda_0 \) can be found from the condition that \( R_k \approx 1 \). We find

\[
\lambda_0 \approx \frac{l}{R^{1/2}}.
\]

It is important to notice, that the inner dimensions of the turbulence gets rapidly smaller for increasing Reynolds number. This point has to be kept in mind when performing numerical simulations of the turbulence. For large values of the Reynolds number a very large number of degrees of freedom is necessary. This makes the direct simulation of turbulence one of the hardest numerical problems.

The spectral form of the law of Kolmogorow and Obuchow  Summarizing, we have seen that the turbulence allows for the distinction of three well-defined regions. The energy range, ist the region for which \( \lambda \approx l \). In this region the greatest part of the kinetic energy is stored. In the dissipation range, \( \lambda \leq \lambda_0 \), the kinetic energy is dissipated. In between for \( \lambda_0 \ll \lambda \ll l \) we have the inertial range. For large Reynolds numbers the energy and the dissipation range are clearly separate. The results of these subsection apply essentially to the inertial range.

It is customary to look at results obtained in this subsection in the Fourier space. To this end we replace the dimension of the fluctuation \( \lambda \) by the wave number \( k \)

\[
k \approx \frac{1}{\lambda}.
\]

By \( E(k)dk \) we denote the kinetic energy per unit mass contained in the interval \( (k,k+dk) \). The dimension of \( E(k) \) is

\[
[E(k)] = \frac{m^3}{s^2}.
\]

The only combination of \( \varepsilon \) and \( k \) with this dimension is

\[
E(k) \approx \varepsilon^{2/3} k^{-5/3}.
\]

The above expression is the famous spectral form of the law of Kolmogorov. The relation to the law we already know is made evident by integration

\[
\int_{k} E(k)dk \approx \varepsilon^{2/3} k^{-2/3} \approx (\varepsilon \lambda)^{2/3} \approx \nu^2 \lambda^{14/3}.
\]

### 10.7.2 Burgerlence

As a one-dimensional model of turbulence we want to consider now the Burgers equation. There is a formal similarity between the statistical equations for the Navier–Stokes equations and the Burgers equation. To establish this relation we consider random solutions \( u(x,t) \), which are stationary random functions of \( x \). The mean energy \( \langle u(x)^2 \rangle/2 \) decays according to the expression

\[
\varepsilon = \frac{1}{\partial t} \frac{\partial}{\partial x} \langle u^2 \rangle = \nu \left( \frac{\partial u}{\partial x} \right)^2 = \nu \frac{\langle u^2 \rangle}{\lambda^2} = \nu \langle \omega^2 \rangle.
\]

The quantity \( \omega = \partial u / \partial x \) is analogous to the vorticity. The arguments put forward by Kolmogorov which are at the basis of the cascade process do not depend upon the dimension of the system under investigation.
and hence they should also apply to Burgerslence. Unfortunately, this is not the case. This can be seen by expoliting the fact that the Burgers equation has a well-known exact solution

\[ u(x,t) = -2\nu \frac{\partial}{\partial x} \log \theta \]

where

\[ \frac{\partial \theta}{\partial t} = \nu \nabla^2 \theta, \]

So it can be shown, that the spectru of Burgerslence is given by the expression

\[ E(k) = \frac{2\nu^2 \pi}{4l} \cosech^2 \left( \frac{\pi \nu k}{2u} \right). \]

For \( k \ll u/\nu \), we have

\[ E(k) = \frac{2u^2}{\pi l} k^{-2} \]

and for \( k \gg u/\nu \)

\[ E(k) = \frac{2\nu \pi}{l} \exp(-\nu k/u), \]

which shows the exponential cut-off due to the viscosity. For Burgerslence the dissipation spectra \( k^2 E(k) \) is constant until the cut-off by viscous decay.

### 10.7.3 The Master Equation Formulation

We now turn to the master equation formulation of the problem. The stochastic process \( \delta u N_z(t) \) obviously represents a homogeneous Markov process. Therefore, it is characterized by the probability distribution \( P(N_z,t) \) and a transition probability \( P_2 \),

\[ P_2 = P_2 \left( \{N_z^{(2)}\}, t_2 \mid \{N_z^{(1)}\}, t_1 \right) \]

which only depends on the difference \( t_2 - t_1 \) of the time arguments. The consistency condition connecting the probability distribution and the transition probability reads

\[ P(N_z, t) = \sum_{N_z^{(0)}} P_2 \left( \{N_z\}, t \mid \{N_z^{(0)}\}, 0 \right) P_0 \left( \{N_z^{(0)}\} \right) \]

where \( P_0 \left( \{N_z^{(0)}\} \right) \) denotes the initial probability distribution.

In a first step, we derive the equation of motion for the characteristic functional \( M(\{z_{\lambda}\}, t) \) pertaining to the stochastic process \( \delta u N_{\lambda}(t) \),

\[ M(\{z_{\lambda}\}, t) := \left\langle \exp \left\{ i \delta u \delta \lambda \sum_{\lambda} z_{\lambda} N_{\lambda} \right\} \right\rangle \equiv \left\langle F(\{N_{\lambda}\}) \right\rangle \]

where we have introduced the function

\[ F(\{N_{\lambda}\}) := \exp \left\{ i \delta u \delta \lambda \sum_{\lambda} z_{\lambda} N_{\lambda} \right\}. \]

The equation of motion for \( M \) contains, of course, the small mesoscopic scales \( \delta u \) and \( \delta \lambda \) introduced within our discrete master equation formulation. This is due to the fact that the phase space \( \Gamma \) underlying our
master equation formulation is discrete in space as well as velocity space. We therefore assume that in the limit of small \( \delta u \) the stochastic process

\[
u(t) := \delta u N(t)
\]

represents a process which, to leading order, does not depend on \( \delta u \). Thus, the second step consists of an expansion of the equation of motion for \( M \) with respect to this mesoscopic velocity scale \( \delta u \). In a third step we then perform the continuum limit \( \delta t \to 0 \). The leading order term of this expansion turns out to be identical to the functional Hopf equation. Furthermore, the physical origin of the next to leading order term is discussed and the mesoscopic scales are related to physical quantities.

The equation for \( M \) can be written in the following form which is appropriate to infer the structure of the continuum limit:

\[
\frac{\partial M}{\partial t} = \nu \sum_{\lambda} \delta l \left( \frac{z_{\lambda+1} - z_{\lambda} - 2z_{\lambda}}{\delta l^2} + \frac{i \partial u \delta l}{4} \left( \frac{z_{\lambda+1} - z_{\lambda}}{\delta l} + \frac{z_{\lambda} - z_{\lambda-1}}{\delta l} \right)^2 \right) \frac{1}{\delta l^2} \frac{\partial^2 M}{\partial z_{\lambda}^2} - \left( \frac{z_{\lambda+1} - z_{\lambda}}{\delta l^2} + \frac{z_{\lambda} - z_{\lambda-1}}{\delta l^2} \right) \langle |u_\lambda| F \rangle .
\]

This is now easy to perform, at least formally, the continuum limit of this equation. In the limit \( \delta t \to 0 \) the set of numbers \( z_\lambda \) turns into a function \( z(x) \). Correspondingly, \( M(\{z_\lambda\}, t) \) becomes a functional \( M[z(x), t] \) and ordinary derivatives with respect to \( z_\lambda \) translate into functional derivatives:

\[
\frac{1}{\delta l} \frac{\partial}{\partial z_\lambda} \to \frac{\delta}{\delta z(x)}.
\]

Moreover, we assume that the process \( u_\lambda(t) \) converges in the continuum limit to a well-defined stochastic process \( u(x, t) \). Hence, we obtain to leading order in \( \delta l \):

\[
\frac{\partial}{\partial t} M[z(x), t] = \nu \int dx \frac{\partial^2 z}{\partial x^2} \frac{\partial M}{\partial z(x)} - \frac{i}{2} \int dx \frac{\partial}{\partial z(x)} \frac{\partial^2 M}{\partial z(x)^2} - \nu \partial u \delta l \int dx \left( \frac{\partial z}{\partial x} \right)^2 \left( \langle |u(x)| e^{i f dy(z(x))} \rangle \right) .
\]

Performing an integration by parts finally yields

\[
\frac{\partial}{\partial t} M[z(x), t] = \int dx z(x) \left( \frac{i}{2} \frac{\partial}{\partial x} \left( \frac{\partial^2 M}{\partial z(x)^2} \right) + \nu \frac{\partial^2}{\partial x^2} \left( \frac{\partial M}{\partial z(x)} \right) \right) - \nu \partial u \delta l \int dx \left( \frac{\partial z}{\partial x} \right)^2 \left( \langle |u(x)| e^{i f dy(z(x))} \rangle \right) .
\]

Eq. (10.70) represents, including terms of order \( \delta u \), the equation of motion for the characteristic functional \( M \) of the stochastic process defined by our multivariate master equation. Comparing Eq. (10.70) with (???), we conclude that the leading order terms in Eq. (10.70) are identical to the Hopf functional equation. The next to leading order term which may formally be written as

\[
c[z(x), t] = -\nu \partial u \delta l \int dx \left( \frac{\partial z}{\partial x} \right)^2 \left( \frac{1}{i} \frac{\delta}{\delta z(x)} M[z(x), t] \right) .
\]

is of order \( \delta u \) and, therefore, vanishes as \( \delta u \to 0 \). Note that including the case of external random stirring forces Novikov derived a generalization of Hopf’s functional equation in which an additional term appears which is of similar structure as \( c[z(x), t] \). However, in our formulation the functional \( c[z(x), t] \) does not represent the effect of external random forces. We shall see below that it has a different physical interpretation.
Eq. (10.70) constitutes the central result of this section. It leads to the conclusion that in the limit \( \delta u \longrightarrow 0 \) the stochastic process \( \delta u N_N(t) \) underlying our master equation is equivalent to the stochastic process \( v(x,t) \) of the Hopf formulation.

From a mathematical point of view this conclusion has to be taken with caution since we did not actually prove that the probability distribution \( P \) converges to \( P^H \) in a rigorous mathematical sense. In order to give such a proof one would first have to construct an embedding of the phase space of smooth velocity fields into the discrete phase space \( \Gamma \) underlying our master equation. Secondly, a measure in the space of probability distributions is required which allows for a precise definition of convergence. A mathematical investigation along these lines must show, in particular, how the two limiting procedures \( \delta u \rightarrow 0 \) and \( \delta l \rightarrow 0 \) have to be performed in order to guarantee certain smoothness properties of the stochastic process \( u(x,t) \).

However, in spite of these mathematical questions we can give the above statement concerning the relation between the random processes \( \delta u N_N(t) \) and \( v(x,t) \) the following meaning. As we will now show interpreting this term as the continuum limit of \( \delta u N_N \) does not lead to a smooth stochastic process. We will see below that this is to be expected physically.

As an example, we derive the equation for the 2–point correlation function

\[
\langle u(y,t)\, u(x,t) \rangle = \frac{1}{l^2} \frac{\delta^2}{\delta z(y) \delta z(x)} \langle f[z,t]\rangle \bigg|_{z=0} .
\]

Following the usual procedure we obtain

\[
\begin{align*}
\frac{\partial}{\partial t} \langle u(y,t)\, u(x,t) \rangle + \frac{1}{2} \langle u(y,t) \frac{\partial}{\partial x} u^2(x,t) \rangle + \frac{1}{2} \langle u(x,t) \frac{\partial}{\partial y} u^2(y,t) \rangle &= \nu \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \langle u(y,t)\, u(x,t) \rangle - \frac{\delta^2}{\delta z(y) \delta z(x)} \langle f[z,t]\rangle \bigg|_{z=0} .
\end{align*}
\]

(10.72)

Apart from the last term the above equation is identical to that one which is obtained from the functional Hopf equation, of course. The functional derivatives of \( f[z,t] \) yield the expression \( \tilde{v} = \langle \{ u \} \rangle \) see Eq. (10.20)

\[
\frac{\delta^2}{\delta z(x) \delta z(y)} \langle f[z,t]\rangle \bigg|_{z=0} = -2\nu \delta u \delta z \bigg( \frac{\partial}{\partial x} \tilde{v} \bigg) \delta(x-y) \quad (10.73)
\]

which is obviously of order \( \delta u \).

From a physical point of view there is another reason why mathematical considerations concerning the continuum limit of \( \delta u N_N(t) \) are only of secondary importance. The reason is that the functional \( f[z(x),t] \) can be given a clear physical meaning. As we will now show interpreting this term as the influence of thermal fluctuations one is forced to fix the product \( \delta u \delta l \) to a finite value. Thus, questions about the existence of the continuum limit are only of mathematical interest.

We now demonstrate that the term (10.73) can be interpreted as a random stress induced by thermal fluctuations. To this end, we assume in the following that the stochastic process is spatially homogeneous; in particular this implies \( \tilde{v} = \text{const} \). Introducing the Fourier transformation of the velocity,

\[
u(x,t) = \frac{1}{L} \sum_k e^{i k x} u_k(t)
\]

(10.74)
where $k = 2\pi n/L$ and $n \in \mathbb{Z}$, we obtain from Eq. (10.72)

$$
\frac{\partial}{\partial t} \langle u^2_k \rangle + \sum_q iq \langle u^2_{k+q} u_q \rangle + c.c. = -2\nu k^2 \langle u^2_k \rangle + 2\nu \delta u \delta l L k^2.
$$

(10.75)

Recall that $\langle u^2_k \rangle$ is proportional to the kinetic energy $E_k$ (per unit mass) pertaining to the mode $k$,

$$
E_k = \frac{1}{2L} \langle u^2_k \rangle.
$$

(10.76)

The convolution sum on the left hand side of equation (10.75) results from the inertial term of Burgers’ equation and obviously couples the different modes. The first term on the right hand side represents the decay of the hydrodynamic modes due to viscous friction whereas the last term is the Fourier transform of (10.73). At small scales, that is for large $k$, the influence of the inertial term may be neglected. It is then easy to see that the effect of the last term in (10.75) is to slow down the exponential decay of the energy of the mode $k$ until, finally, the stationary value

$$
E_k^s = \frac{1}{2} \nu \delta u \delta l
$$

(10.77)

is reached. One can go one step further by requiring that the kinetic energy $E_k^s$ of each stationary mode is equal to its thermodynamic equilibrium value at temperature $T$. This requirement leads to the relation $E_k^s = \frac{1}{2} k_B T/\rho$ ($\rho$ denotes the fluid density and $k_B$ the Boltzmann constant) which implies:

$$
\nu \delta u \delta l = \frac{k_B T}{\rho}.
$$

(10.78)

The fluctuation–dissipation type relation (10.78) fixes $\delta u \delta l$ to a finite value. This means that from a physical viewpoint the continuum limit makes no sense. This fact should have been clear from the beginning since below a certain length scale the assumption of thermodynamic equilibrium and, thus, a description by macroscopic variables only is no longer possible.

It should be clear that performing stochastic simulations of our master equation (see the next section) it is by no means necessary to choose parameters in such a way that the fluctuation–dissipation relation (10.78) is satisfied. In other words, it is not necessary to take into account all scales ranging from the hydrodynamic to the thermodynamic degrees of freedom. On the contrary, in view of practical applications another interpretation of the finite mesoscopic scales $\delta l$ and $\delta u$ is possible. This interpretation is based on the fact that any experimental measurement is characterized by some finite resolution in velocity space as well as position space. It is therefore natural to assume that

$$
\delta l = \Delta l, \quad \delta u = \sigma^2,
$$

where $\Delta l$ denotes the spatial resolution and $\sigma^2$ the variance of the error of velocity measurements. Fixing the mesoscopic parameters in this way, it might be possible to simulate directly that velocity field which is actually measured.

### 10.7.4 The Stochastic Simulation

In this subsection we will apply the stochastic simulation method described above to Burgers’ model of homogeneous turbulence. Let us first precisely define the initial conditions and the physical quantities which have been used in our calculations.

Within our discrete description the 2–point correlation function is defined by

$$
Q(\mu, t) := \frac{1}{L} \sum_{\lambda=0}^{M} \delta l \delta u^2 \langle N_{\mu+\lambda} N_\lambda \rangle, \quad \mu = 0, 1, 2, \ldots, M,
$$

(10.79)
where, assuming spatial homogeneity, a space average has been taken. The expectation value is, as explained in the preceding subsection, evaluated by averaging over the ensemble which is generated by the stochastic simulation. Computing the Fourier transform of the correlation function we obtain the energy spectrum $E_k$ defined in Eq. (10.76). The total kinetic energy $E$ is evaluated by means of the expression

$$E(t) = \frac{1}{2L} \sum_k \delta k \, \delta w^2 \langle N_k^2 \rangle ,$$

(10.80)

which leads to the energy dissipation rate $\epsilon(t) = -\dot{E}$. The initial random velocity is given by a superposition of $N = 1000$ randomly chosen modes,

$$u_0(x) = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} (A_i \cos k_i x + B_i \sin k_i x) + w(x) .$$

(10.81)

Here, $k_i = 2\pi n_i$ (we choose $L = 1$) and the $n_i$ denote identically distributed random integers with distribution function $P(n) = P(-n)$. The amplitudes $A_i$ and $B_i$ are also identically distributed and independent real random numbers with zero mean and with variance

$$\langle A_i A_j \rangle = \langle B_i B_j \rangle = A^2 \delta_{ij} .$$

Furthermore, we add a small fluctuating field $w(x)$ with zero mean and correlation function $\langle w(y) w(x) \rangle = w^2 \delta(y-x)$, where $w^2 = \langle \delta w \rangle$. It follows from these conditions that the initial field (10.81) is homogeneous and that the initial energy spectrum is given by $E_k(0) = A^2 P(k/2\pi)/2 + \delta w \delta k/2$. In our calculations we used $A^2 = 1/3$ and a Poisson–distributed initial spectrum

$$P(n+1) = \frac{1}{2} \frac{\mu^n}{n!} e^{-\mu} ,$$

(10.82)

where $n = 0, 1, 2, 3, \ldots, (M+1)/2$ and $P(0) = 0$. We shall characterize this initial condition by two Reynolds numbers: First, we define an integral Reynolds number based on the total length $L = 1$ and rms–velocity $U := \langle u^2 \rangle^{1/2}$,

$$R_L := \frac{U \cdot L}{\nu} .$$

(10.83)

Second, we introduce a Taylor–Reynolds number by

$$R_\Lambda := \frac{U \cdot \Lambda}{\nu} ,$$

(10.84)

where the Taylor microscale $\Lambda$ is defined by

$$\left( \frac{2\pi}{\Lambda} \right)^2 := \frac{\epsilon(0)}{2 \nu E(0)} = \frac{k^2 E_k(0)}{\sum_k E_k(0)} = \left( \frac{2\pi}{\Lambda} \right)^2 \left[ (\mu + 1)^2 + \mu \right] .$$

(10.85)

The following simulations have been performed along the lines described in subsection A employing the parameters $\delta u = 5 \cdot 10^{-4}$ and $M + 1 = 1024$. The random initial configuration is given by

$$N_k(0) = \int \left( \frac{u_0(s_k)}{\delta u} \right) .$$

(10.86)

In Fig. 1 we depict one realization of the stochastic process $u_\lambda = \delta u N_\lambda$ for three different times and $\nu = 5000^{-1}$. As can be seen from the figure, the initial field develops a typical sawtooth structure consisting of smooth increasing ramps followed by sharp shocks.

In Fig. 2 we show the energy dissipation rate $\epsilon$ as function of time for $\nu = 3000^{-1}$ ($R_L = 1733$, $R_\Lambda = 156$), and $\nu = 5000^{-1}$ ($R_L = 2821$, $R_\Lambda = 253$). In both cases the results have been obtained by averaging over 100 realizations. Fig. 2 clearly demonstrates the characteristic features of the dissipation:
During a short initial period in which the dissipation rate is small, shocks form and the resulting steep velocity gradients lead to a strong enhancement of the dissipation rate. The latter reaches a maximum at a characteristic time $t \approx 0.02$. During the post shock period the dissipation rate can be clearly seen to become independent of the viscosity as is predicted by analytical considerations\(^{22}\).

Finally, we shall discuss the energy spectrum. Fig. 3 shows $E_k$ as obtained from the Fourier transform of the correlation function $Q(\mu,t)$. Again, the expectation value has been evaluated by averaging over 100 realizations of the stochastic process. We show the energy spectrum for three different times and for the viscosity $\nu = 5000^{-1}$ ($R_L = 2821$, $R_A = 253$). As can be seen, a $k^{-2}$ power law behaviour appears which represents the universal inertial range of Burgers' model. In order to demonstrate the accuracy of the stochastic simulation method over the whole range of wavenumbers we compare the spectrum at time $t = 0.06$ with the analytical result given by Saffman\(^{23}\):

$$E_k = W \sinh^{-2}(dk/2\pi), \quad (10.87)$$

Saffman derived this expression for the Burgers model by assuming that the small scale structure is given by periodic trains of shocks. Note that $E_k$ as given by (10.87) approaches for $dk/(2\pi) \ll 1$ the form $E_k \approx 4\pi^2 W/(dk)^2$. The energy scale $W$ and the dissipation length $d$ which depend on the large scale properties of the initial condition have been determined from our data by comparison with the limiting behaviour of the Saffman result for large wavenumbers ($E_k \approx 4W \exp(-dk/\pi)$). We find $W = 0.85 \cdot 10^{-5}$ and $d = 0.0078$. As demonstrated in Fig. 3 the agreement between the analytical result (10.87) with our stochastic simulation is excellent over the whole range of wavenumbers.
Bibliography


Part V

Applications
Chapter 11

Quantum Mechanics Simulations

???
Chapter 12

Risc Management

???
Appendix A

Summary of Java

A.1 Basic Syntax

Documentation
/** Javadoc documentation */

Comments
/* Multi Line Comments */
// Single Line Comments

Constants
final int i = 10;

Logical Operators
!, &, |, ˜, &&, ||

Integer data types
byte, short, int, long

Floating point data types
float, double

Arithmetic Operators
+, -, *, /, %, ++, --

Bitwise Operators
˜, &, |, ˜, <<, >>, >>>

Comparison Operators
<, <=, >, >=, ==, !=

Flow Control
if () ... else ...
... ? ... : ...
switch() { case ... : ...; break; default: ... }
for (.. ; .. ; ..) { ...; }
while () { ...;}
do {...; } while ();

A.2 Structure of a Java program

The general structure of a Java program is:

- package

- import

- Classes

- Interfaces
A.3 The java.lang.System class

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintWriter</td>
<td>err</td>
<td>standard error output stream</td>
</tr>
<tr>
<td>InputStream</td>
<td>in</td>
<td>standard input stream</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>out</td>
<td>standard output stream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>modifier</th>
<th>method</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void</td>
<td>arraycopy(Object src, int cpos, Object dst, int dstpos, int length)</td>
<td>Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.</td>
</tr>
<tr>
<td>static void</td>
<td>exit(int status)</td>
<td>Terminates the currently running Java Virtual Machine.</td>
</tr>
<tr>
<td>static void</td>
<td>setErr(PrintStream err)</td>
<td>Reassigns the &quot;standard&quot; error output stream.</td>
</tr>
<tr>
<td>static void</td>
<td>setIn(InputStream err)</td>
<td>Reassigns the &quot;standard&quot; input output stream.</td>
</tr>
<tr>
<td>static void</td>
<td>setOut(PrintStream err)</td>
<td>Reassigns the &quot;standard&quot; output output stream.</td>
</tr>
<tr>
<td>static String</td>
<td>getProperty(String key)</td>
<td>get the system property defined by the given key.</td>
</tr>
<tr>
<td>static String</td>
<td>setProperty(String key)</td>
<td>set the system property defined by the given key.</td>
</tr>
</tbody>
</table>

Figure A.1: The class structure of a Java program, either application or an applet.
A.4 Mathematics

A.4.1 The java.lang.Math class
A.4.2 JNL
A.4.3 JavaSci
A.4.4 Others

A.5 Random Numbers

A.6 Keyboard input and Screen Output

A.7 File I/O

A.8 Ptplot

A.9 AWT

A.10 Conversions and Casting

A.11 Threads

A.12 Printing

A.13 Modifiers

abstract method: A method having no body of code. The code can be implemented in a subclass.
abstract class: A class containing at least one abstract method.
iinterface: A class, where all defined methods are abstract by default. No variables only constants are possible in interface.

A.14 Debugger

A.15 JDE and Emacs
Appendix B

Listings and Tables

B.1 Listing of the ShowTrace.java Program

```java
/**
 * ShowTrace.java
 */
5 /* <HTML> <applet code="ShowTrace.class" width=200 height=200>
 </applet> </HTML> */
import java.applet.*;

public class ShowTrace extends Applet{
10   public ShowTrace() {
       System.out.println("Constructor! ");
       }
   public static void main(String [] args) {
       System.out.println("Main_Method!");
15     }
   public void init() {
       System.out.println("Init_Method!");
     }
   public void start() {
       System.out.println("Applet_Start_Method!");
     }
20   public void stop() {
       System.out.println("Applet_Stop_Method!");
     }
25  }
```
Appendix C

Listings for the Exercises

C.1 Listings for Chapter 1

C.1.1 Calculation of \( \pi \), Exercise 1.1

The plain program without graphical display:

```java
/** A Monte Carlo method to estimate PI (Hit or Miss) 

we draw random numbers in a square and check how many 
fall into a circle of radius 1. 

Specify the number of points on the command line */

import java.util.Random;

/* if you want to calculate the error of the sample, you 
have to save the estimates in an array. */

public class Pi_Calc_plain {
    // initialize the generator
    public static Random rand = new Random();
    public static long num;
    public static double inside = 0;

    public static void main (String[] args) {
        // check for command line arguments
        if (args.length != 1) {
            System.err.println("Error: no or wrong number of "+args.length+"arguments specified!");
            System.exit(1);
        }
        num = Integer.parseInt(args[0]);

        // Calculation
        inside = 0;
        for (int i = 0; i < num; i++) {
            double x = rand.nextDouble();
            double y = rand.nextDouble();
            double r = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
            if (r < 1) {
                inside++;
            }
        }
    }
}
```

381
// output of results in shell
System.out.println("Calculated Pi using "+num+" points!");
System.out.println("The exact value is "+Math.PI);
System.out.println("The estimate for Pi is "+inside);
System.out.println("The exact error is "+Math.abs(Math.PI-inside));
}

Now the graphics version of the above program:

/** A Monte Carlo method to estimate PI (Hit or Miss)
we draw random numbers in a square and check how many fall into a circle of radius 1.
Specify the number of points on the command line */

import java.util.Random;
import java.awt.*;
import java.awt.event.*;

/* if you want to calculate the error of the sample, you have to save the estimates in an array. */

public class Pi_Calc extends Frame {
    // initialize the generator
    public static Random rand = new Random();
    public static long num;
    // set layout of window
    public static BorderLayout mylayout = new BorderLayout(10,10);
    public static int width = 600;
    public static int height = 600;
    public static double inside = 0;

    public static void main (String [] args) {
        // check for command line arguments
        if (args.length != 1) {
            System.err.println("Error: no or wrong number of arguments specified!");
        }
        Pi_Calc.num = Integer.parseInt(args[0]);
    }

    // create a window and plot points
    Frame f = new Frame("CalcPi using Monte Carlo");
    // Event to close the window and exit
    f.addWindowListener(new WindowAdapter() {
        // Handle window close
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    // size of window
    f.setSize(width, height);
    // set a layout manager
    f.setLayout(mylayout);
```java
// Plot a text on the top bar of the window
f.add(new Label("Calculate PI using Hit or Miss Method. " + "To quit, close the window!"), "North");

// Create a drawing area for the points
Drawing draw = new Drawing(f, width, height);
// add it to window in the middle
f.add(draw, "Center");
// set the background color of the drawing panel
draw.setBackground(Color.white);
// display it all
f.show();
}

static class Drawing extends Canvas {
    protected Frame frame;
    protected int width;
    protected int height;

    public Drawing(Frame frame, int width, int height) {
        this.frame = frame;
        this.width = width;
        this.height = height;
    }

    public void paint(Graphics g) {
        // Calculation and plot points:
        // plot the quarter of a circle
        g.setColor(Color.red);
        g.drawArc(-width, 0, 2 * width, 2 * height, 0, 90);
        g.drawArc(-width - 1, 1, 2 * width, 2 * height, 0, 90);
        g.drawArc(width - 2, 2, 2 * width, 2 * height, 0, 90);
        g.setColor(Color.black);

        // Calculation
        PiCalc.inside = 0;
        for (int i = 0; i < PiCalc.num; i++) {
            double x = rand.nextDouble();
            double y = rand.nextDouble();
            double r = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
            // plot the point
            g.drawLine((int)x, (int)y, (int)x, (int)y);
            if (r < 1) {
                PiCalc.inside += 1;
            }
        }
        PiCalc.inside /= PiCalc.num;
        PiCalc.inside *= 4;

        // plot the quarter of a circle again
        g.setColor(Color.red);
        g.drawArc(-width, 0, 2 * width, 2 * height, 0, 90);
        g.drawArc(-width - 1, 1, 2 * width, 2 * height, 0, 90);
        g.drawArc(width - 2, 2, 2 * width, 2 * height, 0, 90);
        g.setColor(Color.black);

        // output of results in shell
    }
}
```
C.1.2 Photoabsorption, Exercise 4.1

%%% Program Photoabsorption
%%
clear;

%%% Number of slabs
N= input(' Number of slabs (100) ? ');

%%% Transmission probability
p= input(' Transmission probability (0.5) ? ');

%%% Number of particles incident at the beginning
Npart= input(' Initial number of particles incident (10000) ? ');

t=cputime;
Particles (1)= Npart;
for i=2:N+1,
    % Slow Version
    random= rand(Npart, 1);
    for j=1:Npart,
        if ( random(j) > p)
            Npart=Npart - 1;
        end
    end
    % Fast Version
    Npart= Npart * sum(rand(Npart, 1) > p);
    Particles(i) = Npart;
end
disp(sprintf(' Elapsed CPU-Time : %f ', cputime-t))

C.1.3 Monte-Carlo-Integration

Standard Routine

%%% Program Standard Monte-Carlo Integration
%%
clear;

ensembles= input(' How many ensembles ? ');
sampling= input(' Sampling step size ? ');
N=input(’How many x values?’);

10 %% Parameters for the function to be integrated
% exact result, if known
exact=pi/4;
% x interval
xmin=0;xmax=1;
deltax=xmax-xmin;

error(1:ensembles)=0;
t(1:ensembles+1)=0;
t(1)=cputime;

20 for j=1:ensembles,
    sum_func=0;
    x=(rand(N,1)*deltax)+xmin;
    for i=1:N,
        sum_func=sum_func+func(x(i));
    end
    integral(j)=sum_func/N;
    if (rem(j-1,sampling)==0)
        estimate(j)=mean(integral(1:j));
        error(j)=std(integral(1:j))/sqrt(j);
    end
end

t(ensembles+1)=cputime;
estimate(ensembles+1)=mean(integral);
error(j+1)=std(integral)/sqrt(j+1);

disp(sprintf(’nEstimated Integral is: %f +-%f’, estimate(ensembles+1), std(integral)/sqrt(ensembles+1))

disp(sprintf(’nDistance to exact result: %f’, exact-estimate(ensembles+1)))

% plot the result versus ensemble size
subplot(2,1,1);
plot(1:sampling:ensembles+1,4*estimate(1:sampling:ensembles+1));
title(’Estimates for the integral using Standard MCI’);
xlabel(’ensemble size’);
ylabel(’Estimation of PI’);
hold on;
plot([1 ensembles+1],[4*exact 4*exact ],’r’);
hold off;

50 % CPU Time Plot
subplot(2,1,2);
plot(1:sampling:ensembles+1,t(1:sampling:ensembles+1)-t(1));
title(’CPU time estimate’);
xlabel(’ensemble size’);
ylabel(’CPU time in sec.’);

% Error Plots
figure;
subplot(2,1,1);
plot(1:sampling:ensembles+1,error(1:sampling:ensembles+1));
hold on; plot([0 ensembles+1],[0 0],’r’); hold off;
title(’Standard Deviation against ensemble size’);

60 % distance to exact result
```
subplot (2,1,2);
plot (1: sampling : ensembles +1, exact–estimate (1: sampling : ensembles +1));
hold on; plot ([0 ensembles +1],[0 0],’r’); hold off;
title (’distance to exact result versus ensemble size’);

% Error against CPU time
figure;
plot (abs (exact–estimate (1: sampling : ensembles +1)), t (1: sampling : ensembles+1));
title (’Accuracy versus CPU time’);
ylabel (’CPU Time’);
xlabel (’Accuracy’);

Hit and Miss Method

%% Program Hit and Miss

 clear;

 ensembles = input (’How many ensembles ? ’);
 N = input (’How many numbers ? ’);

%% Parameters for the function to be integrated

% exact result if known
exact = pi / 4;
% x interval
xmin = 0; xmax = 1;
deltax = xmax – xmin;
% y interval
fmax = 1; fmin = 0;
delf = fmax – fmin;

delta = [deltax delf ];

for j = 1: ensembles ,
    Hits = 0;
    for i = 1:N,
        y = rand (1,2) .* delta ;
        if (y (2) <= func (y (1)))
            Hits = Hits +1;
        end
    end
    integral (j) = Hits / N;
end
estimate = mean (integral );

disp (sprintf (’\n Estimated Integral is : % f+-% f’, estimate , std (integral )/ sqrt (ensembles )))
disp (sprintf (’\n Exact result : % f’, exact ))
disp (sprintf (’\n Distance to exact result : % f’, exact–estimate ))

%% Program Hit and Miss 2

...
C.1. LISTINGS FOR CHAPTER 1

% clear;
% Parameters for the function to be integrated
% exact result if known
exact=pi/4;

%%% Parameters for the function to be integrated
% exact result if known
10 exact=pi/4;
% x interval
xmin=0; xmax=1; deltax=xmax-xmin;
% y interval
fmax=1; fmin=0; delty=fmax-fmin;

15 N=input('Up to how many points (10000)?');
sampling=input('Sampling size (50)?');

integrall=0; t1=0; tstart=0; Hits=0;
for i=1:sampling:N+1,
    tstart=cputime;
    Hits=sum(rand(i,1)*deltax <= (1./((1+((deltaf*rand(i,1))^2)))));
    integral(i)=Hits/i;
25 i(i)=cputime-tstart;
end
clf;

subplot(3,1,1);

30 plot(1:sampling:N+1,4.*integral(1:sampling:N+1));
hold on; plot([1,N+1],[4*exact 4*exact],'r'); hold off;
xlabel('number of points');
ylabel('Estimate of Pi');
title('Hit and Miss Method for Pi using 1 realization');

35 subplot(3,1,2);

plot(1:sampling:N+1,exact-integral(1:sampling:N+1));
hold on; plot([1,N+1],[0 0],'r'); hold off;
xlabel('number of points');
ylabel('Error of the Estimate');

40 subplot(3,1,3);

plot(1:sampling:N+1,t(1:sampling:N+1));
xlabel('number of points');
ylabel('CPU time in seconds');

disp(sprintf('\nEstimated Integral is: %f',integral(N+1)));
disp(sprintf('\nExact result: %f',exact))
disp(sprintf('\nDistance to exact result: %f',exact-integral(N+1)));

50
%% let the user zoom into the plots
zoom on;
disp('Press a key when finished zooming!');
pause;

55 zoom out; zoom off;

%
% Calculate for different number of ensembles
%
60 clear; figure;
% Parameters for the function to be integrated
% exact result if known
exact=pi/4;
% x interval
65 xmin=0;xmax=1; deltax=xmax-xmin;
% y interval
fmax=1;fmin=0; deltaf=fmax-fmin;

ensembles=input(' How many realizations (1000) ? ');
70 sampling=input(' Sampling size (25) ? ');
N=input(' How many numbers (30) ? ');

t(1:ensembles +1)=0;
estimate (1:ensembles +1)=0;
75 error (1:ensembles +1)=0;

for j_max =1:sampling : ensembles +1,
t_start=cputime;
 for j =1: j_max,
 80 % Hit and Miss !
  Hits =sum(rand(N,1)* deltax <= (1./(1+(deltaf*rand(N,1)).^2)));
  integral (j)= Hits / N;
 end
  estimate (j_max)=mean(integral);
85 error ( j_max)=std (integral) / sqrt ( j_max);
 t ( j_max) = cputime - t_start;
end

disp (sprintf ('\n Estimated Integral is : %f +-- %f', estimate (ensembles +1), std (integral) / sqrt (ensembles +1)));
disp (sprintf (' Exact result : %f', exact))
disp (sprintf ('\n Distance to exact result : %f', exact - estimate (ensembles +1)))

% plot results
 subplot (3,1,1);
95 plot (1:sampling : ensembles +1,4.* estimate ( 1: sampling : ensembles +1));
 hold on;
 plot ([1 ensembles +1],[4*exact 4*exact ],'r');
 hold off;
 xlabel ('number of realizations ');
ylabel ('estimation of Pi ');
title ('Estimating Pi using Hit and Miss MCI ');
subplot (3,1,2);
 plot (1: sampling : ensembles +1,4.* error (1: sampling : ensembles +1));
105 xlabel ('number of realizations ');
ylabel ('standard deviation ');
subplot (3,1,3);
 plot (1: sampling : ensembles +1,t (1: sampling : ensembles +1));
ylabel ('cputime in seconds ');
xlabel ('number of realizations ');

  disp ('Press a key when finished zooming !');
 pause;
 zoom out; zoom off;

C.1.4 Euler Constant
C.1. LISTINGS FOR CHAPTER 1

```matlab
%% Darts Method for calculating e

clear;

N=input(' How many regions / throws ? ');
board(1:N)=0;
region=round(rand(N,1)*(N-1))+1;
for i=1:N,
    board(region(i))=board(region(i))+1;
end
N0=0;N1=0;
for i=1:N,
    if (board(i)==0)
        N0=N0+1;
    elseif (board(i)==1)
        N1=N1+1;
    end
end

disp(sprintf('
Estimation for e : %f',N/N0));
disp(sprintf(' or using N1 : %f',N/N1));
disp(sprintf('
Exact solution : %f',exp(1)));
```

C.1.5 The Standard Deviation

```matlab
%% Compare the different methods for the variance

clear; help variance;

N=input(' number of random numbers (15000) : ');
mean_sample=input(' mean of the sample (100) ? ');
slow=input(' Slow version = 1 (0) ? ');
x=rand(N,1)*2*mean_sample;

xm=mean(x);
xs=std(x)^2;
std1=sum((x-xm).^2)/(N-1);
std2=sum(x.^2)/N-xm^2;
std3 =0;
if (slow == 1)
    for j=1:N
        std3 =std3 +sum((x(j+1:N)-x(j)).^2);
    end
    std3 =std3 /N^2;
end
std4 =((N-1)*std1 -(sum(x-xm)^2)/N)/(N-1);

disp(sprintf(' mean value : %f ',xm));
disp(sprintf(' median : %f ',median(x)));
disp(sprintf('n std ( ) function : %2.0f ',xs));
disp(sprintf('l definition : %2.0f ',std1));
```
APPENDIX C. LISTINGS FOR THE EXERCISES

C.2 Listings for Chapter 2

C.2.1 Random number generator check

```matlab
%% Program Check random number generator rand()
%%
%% generate moments of uniform distribution
%%
clear;

% input parameters
num_moments=input('How many moments of the distribution (10)?');
N=input('How many random numbers (5000)?');

% create random numbers
random=rand(N,1);

moment(1:num_moments)=0;
error(1:num_moments)=0;

% calc the moments and the standard deviation of moments
for i=1:num_moments,
    moment(i)=mean(random(1:N).^i);
    error(i)=std(random(1:N).^i)/sqrt(N);
end

% plot the calculated moments and their exact results
subplot(3,1,1);
plot(moment,'b+');
hold on;
plot(1:num_moments,1./(2:num_moments+1),'r');
hold off;
title('Moments of the uniform distribution');
subplot(3,1,2);
errorbar(moment,error);
title('Standard deviation of the moments');
% Histogram to check for distribution
subplot(3,1,3);
hist(random,floor(N/100));
hold on;
plot([0 1],[100 100],'r');
hold off;
title('Histogram of the distribution');
```

%% Poker Test
%%
%% Test the random number generator using the Poker game

```matlab
clear;help Poker_Test;
```

% input parameters
C.2. LISTINGS FOR CHAPTER 2

N=input(' How many hands (20000) ? ');
10 t=cputime;
hands(1:5)=0;
15 for i=1:N,
    % give a hand
16    random=sort(floor(rand(5,1)*14-eps)+1);
    % check for hands
17    compare=random(1);
18    anz=1;
19    anz_max=1;
20    for k=2:5,
16      if (compare==random(k))
17          anz=anz+1;
        else
18            compare=random(k);
19          if (anz>anz_max)
20              anz_max=anz;
21            end
22            anz=1;
        end
23    end
24    hands(anz_max)=hands(anz_max)+1;
25 end
disp(sprintf(' CPU time: %f',cputime-t));
disp(sprintf(' sum of all hands counted: %i',sum(hands)));
30 % plot bargraph of result and counts
31 figure(1);
32 bar(hands);
33 title(' Poker test for the random number generator ');
34 max_hands=max(hands);
35 for i=1:5
36    text(i-0.3/i,hands(i)+max_hands/6,sprintf('%i',hands(i)));
37 end
38 axis([0 6 0 4/3 max_hands]);
39 xlabel(' of a kind ');
40 ylabel(' number of hands ');
41 % exact probabilities for Poker
42 exact=[0.501177 0.422569 0.02112845 0.000240096 0.0];
43 figure(2);
44 plot(hands/N-exact,'b+');
45 hold on;
46 plot([0 6],[0 0],'r');
47 hold off;
48 xlabel(' of a kind ');
49 ylabel(' error of the generated hands ');

\%\% Program Galton Board
\%\%
clear;
5 % input parameters

C.2.2 Galton Board

%% Program Galton Board
%%
clear;
5 % input parameters
stairs=input('How many stairs?');
balls=input('How many balls?');

10 % probability at the sticks
p=0.5;

position(1:balls)=0;
for i=1:balls
15 % Calculate number of steps to the left
    position(i)=(stairs+1)-sum(rand(stairs,1)>p);
end

%%%% plot the histogram of the results
20 hist(position,1:stairs+1);
histo_max=max(hist(position,1:stairs+1));
hold on;

%%%% plot the theoretical result:
25 % (the normal distribution for number of stairs to infinity)
% we need therefore: the variance and the mean of the sampled data
% the mean is just (stairs +2)/2;
% and the variance is calculated using the sampled data, so that the
% theoretical plot just fits the sampled data
30 variance=1/(2*std(position)^2);
x=0:0.25:stairs+2;
plot(x,histo_max*exp(-variance*(x-(stairs+2)/2).^2),'r');

% annotate the plot
35 axis([0 stairs+2 0 histo_max]);
title('The Galton Board');
xlabel('box number');ylabel('number of balls');
hold off;

C.2.3 Poisson Distribution

% Generate Poisson Random Numbers using
% the transformation method and the exponential
% distribution
5 %
% Factorials can be generated using the prod() function!
% N! = prod(1:N)
%
clear; help Poisson;
10 set(0,'DefaultAxesFontSize',13);

% Parameters of the Poisson Distribution
lambda=input('lambda for the Poisson distribution (10) ?');

15 N=input('How many random numbers (1000) ?');
poiss(1:N)=0;

% Loop over random numbers
for i=1:N,
20 % Algorithm
    A=1;
k=0;
C.3. Listings for Chapter 3

C.3.1 Random number generator

% Program to generate random numbers
% uniform in [0,1)
% using linear congruential method

clear; help linear_con;
set(0,'DefaultAxesFontSize',18);

% Parameters for the generator
parm=input('Which parameter set (1 or 2)?
if (parm==1)
a=16807;
c=0;
M=2^31-1;
else
a=65539;
c=0;
M=2^31-1;
end

A=A*rand(1);
while ( A >= exp(-lambda) )
    A=A*rand(1);
k=k+1;
end

Pois (i)=k;

% display the results
disp(sprintf('Mean value (%f): %f', lambda, mean(Pois)));
disp(sprintf('Variance (%f): %f', lambda, std(Pois)^2));

% plot the results
% the random numbers
figure (1); plot(Pois,'+');
title('the generated sequence of Poisson distributed random numbers');

% the distribution
figure (2); hist(Pois,20);
for j=1:100
    factorial(j)=prod(1:j);
end
Pois_theory = lambda*(1:100)*exp(-lambda)/factorial(1:100);
Pois_theory = Pois_theory * (max(hist(Pois,20))/max(Pois_theory));
hold on; plot(1:50,Pois_theory(1:50),'r');
hold off;
title('Histogram of generated numbers compared with theory');

% 2D Plot
figure (3);
plot(Pois(1:2:N-1),Pois(2:2:N),'+');
title('2D Poisson distribution');
APPENDIX C. LISTINGS FOR THE EXERCISES

% Initial Seed
Seed=input(' Seed for the generator (1) ? ');

N=input(' How many numbers should I generate (10000) ? ');
I(1:N+1)=0;
I(1)=Seed;

% Generate the numbers
for i=2:N+1,
    I(i)=floor(rem(a*I(i-1)+c,M));
end
I=I./M;

% display/plot the results
disp(sprintf(' mean value (0.5): %f',mean(I)));
disp(sprintf(' variance (0.08333): %f',std(I)^2));

% Histogram of distribution
figure(1); hist(I,10);
title('Histogram of the random numbers ');

% random numbers itself in 1D
figure(2); plot(I,'*');
title('generated uniform random numbers ');
xlabel('number'); ylabel('random number ');

% random numbers itself in 2D
figure(3); plot(I(1:2:N),I(2:2:N+1),'*');
title('2D vectors of the random numbers ');

% random numbers itself in 1D
figure(4); plot3(I(1:3:N-1),I(2:3:N),I(3:3:N+1),'*');
title('3D vectors of the random numbers ');
rotate3d;

C.3.2 Acceptance-rejection method

%% Program Acceptance / Rejection Method
%% Calc the volume of a n-dimensional sphere
%% using the von Neumann method

clear; help rejection.m;

dim=input(' Dimension of sphere (3) ? ');
samples=input(' Sample size (10000) ? ');
reals=input(' How many realizations (500) ? ');

%%% Parameters
Radius=1;

% exact result
exact(dim)=Radius^dim*pi^(dim/2)/gamma(dim/2+1);
Radius=Radius^2;
C.3. LISTINGS FOR CHAPTER 3

Hits(1:realis)=0;
20 for j=1:realis,
   % Slow Version
   for i=1:samples,
      % if ( sum(( (2*rand(dim,1)-1).*Radius).^2) < Radius)
         Hits(j)=Hits(j)+1;
   % end
end

% Fast Version
Hits(j)=sum(sum(((2.*rand(dim,samples)-1).*Radius).^2) < Radius);

% Calculate results
Hits(j)=Hits(j)/samples*(2^dim);
end

% calc mean and standard deviation
estimate=mean(Hits);
error=std(Hits)/sqrt(realis);

disp(sprintf('nEstimated volume of sphere is : %f +/- %f', estimate, error));

disp(sprintf('Exact result: %f', exact(dim)));
disp(sprintf('nDistance to exact result: %f', exact(dim)-estimate));

% plot samples, mean and 68 percent area around mean
plot(Hits,'bo'); hold on;
plot([1 realis],[estimate estimate],'b');
error=error*sqrt(realis);
plot([1 realis],[estimate+error estimate+error],'b--');
plot([1 realis],[estimate-error estimate-error],'b--');

% plot exact result
plot([1 realis],[exact(dim) exact(dim)],'r:'); hold off;
title(sprintf('nVolume of the %i dimensional sphere ',dim));
xlabel('realizations'); ylabel('volume');

C.3.3 Importance Sampling

%%
%% Importance Sampling
%%
%% Calculate an Integral using an importance function
%%
5  clear; help importance;

N0=input('How many points (1000)? ');
cmax=input('up to which cut-off value (10)? ');

j=1;
for c=0.1:0.2:cmax,
   % normal distributed random numbers with variance 1/2
   % in the interval [-infinity,infinity)
   xi=randn(N0,1)/sqrt(2);
   % take only the values in the integration interval
   xi=xi.*(abs(xi)<c);
   % use correct N
   N=sum(abs(xi)<c);
20   % correct for the normalization constant and the extended interval
C.3.4 First passage times

%%% First Passage Time (fpt) for 2D random walk
%%% use a circle with radius R for the fpt
%%% clear; help first_passage;

N=input('How many walks (1000)?');
R=input('Radius R of circle (5)?');

% maximum of 10000 steps for the fpt
fpt(1:10000)=0;

j_max=1;
for i=1:N,
    j=1; pos(1:2)=0;
    % do steps until the radius is reached
    xi=x.*sqrt(pi)/2;
    estimate(j)=sum(abs(xi))/N;
    xi2=rand(N,1).*c;
    estimate_standard(j)=c./N*sum(xi2.*exp(-xi2.^2));
end
var_imp=std(estimate(floor(j/2):j-1));
var_standard=std(estimate_standard(floor(j/2):j-1));
disp(sprintf('Results for effective N=%i:
importance sampling: %f
standard sampling: %f
variance importance sampling: %f
variance standard sampling: %f
', N, estimate(j-1), estimate_standard(j-1), var_imp, var_standard));

% plot the integral versus cutoff and the estimates
figure(1); v=0:0.1:cmax;
plot(v,0.5*0.5*exp(-v.^2),'b'); hold on;
plot(0.1:0.2:cmax,estimate(1),':r');
plot(0.1:0.2:cmax,estimate_standard(1),':r');
hold off; title('Value of the integral versus the cutoff value');
xlabel('Cutoff c'); ylabel('integral');

% plot systematic error because of cutoff
figure(2); v=0:0.5:c;
semilogy(v,0.5*0.5*exp(-v.^2));
title('Systematic error because of cutoff');

% plot the function itself
figure(3); v=0:0.1:5;
plot(v,v.*v.*exp(-v.^2));
title('The function v e^(-v^2)');

\textbf{APPENDIX C. LISTINGS FOR THE EXERCISES}
C.3. LISTINGS FOR CHAPTER 3

while ( norm(pos(:))<=R ),
x=3;
while ( x==3 )
    x=floor(1+rand(1)*2);
end
% create a -1 or 1 with prob. 0.5 each
jump=0;
while ( jump == 0 )
    jump=sign(rand(1)-0.5);
end
pos(x)=pos(x)+jump;
j=j+1;
end
fpt(j)=fpt(j)+1;
if ( j>j_max )
j_max=j;
end

% results
mean_fpt=sum(fpt(1:j_max).*fpt(1:j_max))/N;
disp(sprintf('mean first passage time is %f steps.',mean_fpt));

% plot the results
figure(1); plot(fpt(1:j_max));
title(sprintf('first passage times (steps) for radius R=%i',R));
xlabel('first passage time (steps)'); ylabel('counts');
% plot the mean value
hold on; plot([mean_fpt mean_fpt],[0 max(fpt)+2],'r:'); hold off;

% plot of some results from different runs
R_runs=[1 2 2.5 3 3.5 4 4.5 5 6 7 10 20];
fpt_runs=[3.65 6.65 10 12.15 15.8 19.5 26.27 30.03 ... 41.22 53.97 110.48 436.37];
figure(2); plot(R_runs,fpt_runs);
title('First Passage Times for different Radii (1000 walks)');
xlabel('Radius R'); ylabel('mean first passage time');

C.3.5 Scaling Behaviour of Random Walk in 2D and 3D

clear; help rw_scaling;

dimension=input( 'Dimension (2 or 3) ? ' );
realizations=input( 'How many realizations (100) ? ' );
N_min=input( 'How many steps START (10) ? ' );
N_max=input( 'How many steps STOP (1000) ? ' );
N_step=floor((N_max-N_min)/N_step);

% Symmetric Probabilities
P(1)=0.25; P(2)=0.5;
P(3) = 0.75; P(4) = 1.0;
% for 3D
P(5) = 1.25; P(6) = 1.5;

20 \% square lattice possibilities
if (dimension == 2)
  \% for 2D
  walk(1:1:2) = [1 0]; walk(2:1:2) = [-1 0];
  walk(3:1:2) = [0 1]; walk(4:1:2) = [0 -1];
else
  \% for 3D
  walk(1:1:3) = [1 0 0]; walk(2:1:3) = [-1 0 0];
  walk(3:1:3) = [0 1 0]; walk(4:1:3) = [0 -1 0];
  walk(5:1:3) = [0 0 1]; walk(6:1:3) = [0 0 -1];
end

% fixed or random stepsize
step = input('Use fixed stepsize of 1 (otherwise random stepsize in [0,1]) (1) ?');
if (step == 1)
  disp('Stepsize is 1 !');
  step_max = 1;
else
  step_max = input('Maximum stepsize (1) ?');
  disp('Stepsize is random between [0,1]!');
end

% start the walks ...
t(1) = cputime;

50 \% Scaling loop
count = 0;
for N = N_min : N Step : N_max,
  count = count + 1;
  step_size = ones(N,1);
  disp(sprintf('Walks with N = %i ! CPU time for previous walks: %f', N, cputime - t(count)));
  t(count + 1) = cputime;
  \% do the realizations
  for j = 1 : realizations,
    pos(1:dimension) = 0;
    random = rand(N,1) * (dimension / 2);
    if (step == 0)
      step_size = step_max .* rand(N,1);
    end

    \% do the steps
    for i = 1 : N,
      pos = pos + step_size(i) .* ...
      walk(sum((random(i) * ones(1, dimension * 2)) < P(1:2 * dimension))) ;
    end

    \% calc the end-to-end distance
    R_square(count, j) = norm(pos)^2;
  end
end

70 \% plot the elapsed CPU time
  t_total = cputime - t(1);
  disp(sprintf('CPU Time: %f seconds', t_total));
% calc results (mean and error)

75 % matlab
R_sq_mean=mean(R_square(1:count,:),2);
R_sq_stddev=std(R_square(1:count,:),0,2)/sqrt(realizations);
% octave
R_sq_mean=mean(R_square(1:count,:));

80 % Least Square Fit
fit=polyfit(log(N_min:N_step:N_max),log(R_sq_mean),1);
disp(sprintf('Least square fit : b=%f and a=%f !','
85 fit(1),exp(fit(2)/fit(1))));
loglog(N_min:N_step:N_max,fit(2)+fit(1)*(N_min:N_step:N_max),'r-');
axis([N_min N_max min(R_sq_mean) max(R_sq_mean)]);
hold on;

90 % plot the sampled results
errorbar(N_min:N_step:N_max,R_sq_mean,R_sq_stddev);
hold off;
title(sprintf('End-to-End Distance in %i Dimensions',dimension));
xlabel('length of the walk N');
95 ylabel('End-to-End Distance \hat{R}^2');
zoom; disp('You can zoom through the function now !!!

C.3.6 Percolation in 2D

%%
%% Percolation in 2D
%%
%% Simulate a percolation cluster in 2D
5 %% and find the spanning clusters using the
%% Hoshen-Kopelmann algorithm
%%
clear; help percolation;

10 size=input('Size of the cluster (20) ? ');
p_min=input('Occupation probability START (0.2) ? ');
p_max=input('Occupation probability STOP (0.8) ? ');
p_step=input('Stepsize for probability (0.05) ? ');
realizations=input('Number of realizations (100) ? ');

15 %%
%% Probability loop
%%
number=floor((p_max-p_min)/p_step)+1; p_sc(number+1)=0; count=0;
20 for p=p_min:p_step:p_max,
count=count+1;
disp(p)
%
25 % Realization loop
%
span_cluster(1:2)=0;
for real=1:realizations,
30 % Create a percolation cluster of size 'size+1'
% have zeros on the left and upper faces!
cluster(1:size+2,1:size+2)=0;
category(2:size+1,2:size+1)=(rand(size)<p);

35  \% Hoshen–Kopelmann algorithm
L(1:size+2,1:size+2)=0; m=0;
% maximum number of indices for one cluster (used to
% find the proper cluster index)
proper(1:2,1:10)=0;
40  \textbf{for} i=2:size+1,
   \textbf{for} j=2:size +1,
   \textbf{if} (cluster(i,j)==1)
      \textbf{if} (cluster(i-1,j)==0)
         \textbf{if} (cluster(i,j-1)==0)
            \% both unoccupied
            m=m+1; N(m)=m; L(i,j)=m;
         \textbf{else}
            \% one occupied
            L(i,j)=L(i,j-1);
       \textbf{else}
          \textbf{if} (cluster(i,j-1)==0)
             \% one occupied
             L(i,j)=L(i-1,j);
          \textbf{else}
             \% two occupied
             \textbf{if} (L(i-1,j)==L(i,j-1))
                \% same labels
                L(i,j)=L(i-1,j);
             \textbf{else}
                \% different labels
                \% classify the proper cluster numbers of
                \% the two neighbouring sites
                \% first
                t1=-N(L(i-1,j)); tc1=1;
                \textbf{while} (t1>0)
                   proper(1,tc1)=t1;
                   t1=-N(t1);
                   tc1=tc1+1;
                \textbf{end}
                \textbf{if} (tc1<2)
                   N(L(i-1,j))=proper(1,tc1-1);
             \textbf{end}
       \textbf{else}
          \textbf{if} (cluster(i,j-1)==0)
             \% one occupied
             L(i,j)=L(i-1,j);
          \textbf{else}
             \% two occupied
             \textbf{if} (L(i-1,j)==L(i,j-1))
                \% same labels
                L(i,j)=L(i-1,j);
             \textbf{else}
                \% different labels
                \% classify the proper cluster numbers of
                \% the two neighbouring sites
                \% first
                t2=-N(L(i,j-1)); tc2=1;
                \textbf{while} (t2>0)
                   proper(2,tc2)=t2;
                   t2=-N(t2);
                   tc2=tc2+1;
                \textbf{end}
                \textbf{if} (tc2<2)
                   N(L(i,j-1))=proper(2,tc2-1);
              \textbf{end}
          \textbf{end}
   \textbf{end}
45  \% get proper index
k_min=min(-t1,-t2);
k_max=max(-t1,-t2);
L(i,j)=k_min;
\textbf{if} (k_min=k_max)
N(k_{\text{max}})=k_{\text{min}}; \\
\text{end} \\
\text{end} \\
\text{end} \\
\% end of if block \\
\text{end} \\
\% redundant index removal from cluster \\
\text{again}=1; \\
\text{while} (\text{again}==1) \\
\text{again}=0; \\
\text{for} i=2: \text{size}+1, \\
\text{for} j=2: \text{size}+1, \\
\text{if} (\text{L}(i,j)>0) \\
\text{if} (N(\text{L}(i,j))<0) \\
\text{L}(i,j)=\text{abs}(N(\text{L}(i,j))); \\
\text{again}=1; \\
\text{end} \\
\text{end} \\
\text{end} \\
\% spanning clusters ?? \\
\% upper and lower faces \\
\text{span}=0; \\
\text{for} i=2: \text{size}+1 \\
\text{if} (\text{L}(2,i)^{\text{\textregistered}}=0) \\
\text{if} (\text{sum(L}(2,i)==\text{L}(\text{size}+1,:))>1) \\
\% There is one \\
\text{span}=1; \text{break}; \\
\text{end} \\
\text{end} \\
\text{end} \\
\% left and right faces \\
\text{if} (\text{span}==0) \\
\text{for} i=2: \text{size}+1 \\
\text{if} (\text{L}(i,2)^{\text{\textregistered}}=0) \\
\text{if} (\text{sum(L}(i,2)==\text{L}(:,\text{size}+1))>1) \\
\% There is one \\
\text{span}=1; \text{break}; \\
\text{end} \\
\text{end} \\
\% disp (’We have found a spanning cluster !!!’); \\
\% disp (’NO spanning cluster !!!’); \\
\% end \\
\% plot using pcolor ONLY for first realization \\
\text{if} (\text{size}<22 & \text{real}==1) \\
\text{figure (1)}; \\
pcolor(L(2: \text{size}+2,2: \text{size}+2)); \text{shading flat};
for i=2:size+1
  for j=2:size+1
    text(j-1+0.25, i-1+0.35, num2str(L(i,j)));
  end
end
pause;
end

% % End of realizations loop
% span_cluster(span+1)=span_cluster(span+1)+1;
end

%% % End of probability loop
%% p_sc(count)=span_cluster(2)/realizations;
end
figure(2);
plot(p_min: p_step: p_max, p_sc(1: number));
title(sprintf('2D percolation with size %i', size));
xlabel('occupation probability');
ylabel('spanning cluster probability');
% disp(sprintf('Critical probability for spanning clusters is %f ... 
% mean(p_sc));

C.3.7 First passage times

% Einstein Solid
% Simulate an Einstein Solid to clarify the importance of the
% Boltzmann distribution
% clear; help einstein_solid;

N=input('Size of the 2D lattice (N=40) ? ');
steps=input('How many steps (10000) ? ');
interval=input('Plot after how many steps each (2000) ? ');
initial=input('Display initial 200 steps (1=yes) ? ');

% initial state
lattice (1:N,1:N)=1;

% transfer of quanta (steps)
plotit=interval;
for i=1:steps,
  flag=1;
  % do as long as the step is disregarded
  while (flag==1)
    % get x, y coordinates of cells
    rnd(1:4)=floor(rnd(1,4)*N-10)+1;
    if (rnd(1:2) == rnd(3:4))
      % is there a quanta in the cell to jump?
      if (lattice(rnd(1),rnd(2))>0)
        lattice(rnd(1),rnd(2))=lattice(rnd(1),rnd(2))-1;
      end
    end
  end
end
C.4. LISTINGS FOR CHAPTER 4

C.4.1 One-Step Processes

```matlab
lattice (rnd(3), rnd(4)) = lattice (rnd(3), rnd(4)) + 1;
flag = 0;
end
% end if
end

% end while
if (i == plotit | (i <= 200 & initial == 1 & mod(i,10) == 0))
    figure (2); contourf (lattice); colorbar; axis square;
title (sprintf ('Lattice after %i steps', i));
pause; plotit = i + interval;
end

% plot the resulting lattice
% numbers
if (N <= 20)
lattice
end
axis auto;
% contour plot
figure (1); contour (lattice);
% contour with patches (filled)
figure (2); contourf (lattice); colorbar;

% surface plot
figure (3); mesh (lattice);
% (filled) surface plot
figure (4); surf (lattice); colorbar;

% distribution function
df (1:N+1) = 0;
for i = 1:N,
df = df + hist (lattice (i,:), 0:N);
end

% fit a boltzmann distribution
% avoid zeros because of logarithm
for i = 1:N+1,
    if (df(i) == 0) break; end
end
a = polyfit (0:2, log (df(1:i-1)), 1);
figure (5); semilogy (0:i-1, df(1:i), 'rd');
title ('distribution of sites containing N quanta');
xlabel ('quanta N'); ylabel ('number of sites');
hold on; semilogy (0:i, exp ((0:i)*a(1))*exp(a(2))); hold off;
disp ('The resulting distribution is:');
disp (df);
disp (sprintf ('Consistency Check for the simulation: ');
disp (sprintf ('Sum of all quanta: %i -- should be %i!', sum(df), NaN));
```

C.4 Listings for Chapter 4
%onestep — Program to simulate a one-step process
clear; help onestep; % Clear memory and print header
set(0,'DefaultAxesFontSize',16);

nstart = input('Enter initial value of N (500) = ');
tend = input('Enter final time in s (30) = ');
nreal = input('Enter number of realizations (10) = ');
Tstart = cputime;

10 tstart = 0;
nmes = zeros(1, tend + 1);
nmes(1) = nstart;
nmes2 = zeros(1, tend + 1);
nmes4 = zeros(1, tend + 1);

15 nmes2(1) = 0;
nmes4(1) = 0;
tmes = [0 : tend + 1];

% realizations loop
for j = 1 : nreal
  20 t = tstart;
n = nstart;
  imes = 2;
  while (t <= tmes(imes))
    % determine one-step jump probabilities per unit time
    % transition rates for radioactive decay
    % [g, r] = decaymaster(n);
    %
    % transition rates for the Poisson process
    % [g, r] = poissonmaster(n);
    %
    30 % transition rates for the continuous time random walk
    % [g, r] = walkmaster(n);
    % evaluate total jump rate
g = 0.4 * n;
    35 r = 0.5 * n;
    lambda = g + r;
yl = r / lambda;
    % draw exponentially distributed random number
    tau = -log(rand(1, 1)) / lambda;
    40 t = t + tau;
    while (t >= tmes(imes))
      nmes(imes) = nmes(imes) + n;
      nmes2(imes) = nmes2(imes) + n * n;
      nmes4(imes) = nmes4(imes) + n * n * n * n;
      imes = imes + 1;
      if imes >= (tend + 2)
        break
      end
    end
  end

50 if rand(1, 1) < yl
    % jump from n to (n-1)
    n = n - 1;
  else
    % jump from n to (n+1)
    n = n + 1;
  end

55 end % end of time integration
end
% end of realization loop
end

% normalize mean values and variance
nmes=nmes/nreal;
nmes(1)=nmes(1)*nreal;
nmes2=nmes2/nreal;
nmes4=nmes4/nreal;
sdev(1)=0;
sdev2(1)=0;
for imes=2:tend+1
    sdev(imes)=sqrt((nmes2(imes)-nmes(imes)*nmes(imes))/nreal);
sdev2(imes)=sqrt((nmes4(imes)-nmes2(imes)*nmes2(imes))/nreal);
end
disp('CPUTIME: ');
cputime=Tstart
if nreal==1
    %plot one realization of the process
    plot(0:tend,nmes,'x');
    %title('Radioactive decay');
    %title('Poisson process');
    title('Continuous time random walk');
    xlabel('time');
ylabel('n');
else
    %plot result of simulation with errorbars
    subplot(2,1,1)
    errorbar(0:tend,nmes,sdev);
    %title('Radioactive decay');
    %title('Poisson process');
    title('Continuous time random walk');
xlabel('time');
ylabel('n');
    subplot(2,1,2)
    errorbar(0:tend,nmes2,sdev2);
    xlabel('time');
ylabel('n^2');
end

%%% Onestep FAST
clear; help onestepfast;

% parameters for correlation function
correlation=input('Calculate correlation function (no=0, yes=1)? ');
if correlation=0
    correlation=1;
anz_corr=input('How many correlation functions should be plotted (5)? ');
end
% After how many steps assume the stationary state?
t_stationary=5;
% input parameters
initial=input('Initial Value? ');
t_end=input('End Time (50)? ');
real=input('Number of Realizations (100)? ');
if (real>1)
    t_sample=input('Sample Time interval (1)? ');
end
% How many steps to save
t_save = floor ( t_end / t_sample ) + 1;
% initialize
N(1: real , 1: t_save ) = 0;
else
25   t_sample = t_end;
   t_save = t_end;
end
29   pn = input ( 'Plot P(n,t) for each t (yes = 1, no = 0)?' );
   if ( pn = 0)
30      pn = 1;
end
%%%%% START of SIMULATION
%%%%%
35   % CPU Time
   t0 = cputime;
   % realization loop
   for i = 1: real ,
      N( i , 1 ) = initial ;
      nn = initial ;
      t = 0; sample = 1; sample2 = 1; lambda = 1;
      % one realization :
      while ( sample < t_save )
         % advance until save of position
         while ( t < t_stop )
            % CHANGE HERE
            [ g, r ] = payrollmaster ( nn );
            %
            lambda = g + r;
            % check if total transition probability is zero (—> stationary )
            if ( lambda > 0)
35      % generate random numbers and advance time
                  random = rand ( 2 , 1 );
                  t = t - log ( random ( 1 ) ) / lambda ;
                  % advance : n = n + 1 or n = n - 1 ??
                  nn = nn + 2 * ( ( random ( 2 ) * lambda ) > r ) - 1;
            end
            if ( lambda > 0 & real == 1)
35         N ( i , sample2 ) = nn ;
                  t_real ( sample2 ) = t ;
                  sample2 = sample2 + 1;
            end
50   end
   if ( lambda > 0 & real > 1)
       % save position
       sample = sample + 1;
       N( i , sample ) = nn;
   else
55       break ;
   end
   if ( lambda > 0 & real > 1)
60       % save whole realization if only one is calculated
       if ( real == 1)
                  N( i , sample2 ) = nn ;
                  t_real ( sample2 ) = t ;
                  sample2 = sample2 + 1;
       end
65   end
   if ( lambda > 0 & real > 1)
   end
   end
end
%%%%%
% END OF SIMULATION

C.4. LISTINGS FOR CHAPTER 4

%%%END OF SIMULATION

% prepare results: first and second moments
if (real > 1)
    \% mean
    N_mean(1 : t_save) = mean(N(:, :), 1);
    N_std(1 : t_save) = std(N(:, :), 0, 1) / sqrt(real);
    \% second moment
    N_moment2(1 : t_save) = mean(N(:, :) .^ 2, 1);
    N_std_moment2(1 : t_save) = std(N(:, :) .^ 2, 0, 1) / sqrt(real);
    \% variance
    N_var(1 : t_save) = mean((N(:, :) - repmat(...
                               N_mean(1 : t_save), real, 1)) .^ 2, 1);
    N_std_var(1 : t_save) = std((N(:, :) - repmat(...
                                     N_mean(1 : t_save), real, 1)) .^ 2, 0, 1) / sqrt(real);
    \% calculate and plot covariance (correlation)
    if (correlation == 1)
        correg = corrcoef(N(:, :), t_save);
        figure(6); pcolor(correg); title('Patch Plot of Correlation matrix');
        xlabel('t'); ylabel('correlation coefficient'); colorbar;
        col = ['y', 'm', 'c', 'r', 'b', 'k', 'g'];
    end
    grid; hold on;
    for j = 1 : anz_correl,
        plot(j * sample : t_sample : t_end, abs(correg(j, j : t_save - 1)), col(j));
    end
end

%%% END OF SIMULATION

% plot final results
if (real > 1)
    figure(1);
    errorbar(0 : t_sample : t_end, N_mean(1 : t_save), N_std(1 : t_save));
    xlabel('t'); ylabel('first moment'); axis tight;
    figure(2);
    errorbar(0 : t_sample : t_end, N_moment2(1 : t_save), N_std_moment2(1 : t_save));
    xlabel('t'); ylabel('second moment'); axis tight;
    figure(3);
    errorbar(0 : t_sample : t_end, N_var(1 : t_save), N_std_var(1 : t_save));
    xlabel('t'); ylabel('variance'); axis tight;
    \% distribution function using histogram estimate
    xmin = min(min(N)); xmax = max(max(N));
    \% use only half of information \rightarrow smoothing of P
    P = mean(hist(N(:, t_stationary : t_save),((xmax - xmin) + 1)/2), 2); P = P / sum(P);
    plot(P); xlabel('n'); ylabel('P(n)');
    title('estimated normalized stationary distribution function');
    \% distribution function with time
    if (pn == 1)
        disp('Plotting P(n,t): Always press a key for the next time t!');
    for i = 2 : t_save,
        xmin = min(N(:, i)); xmax = max(N(:, i));
    end
end
P = hist(N(:, i), (xmax-xmin)+1); P=P/sum(P);
figure(5);
plot(xmin:xmax,P); xlabel('n'); ylabel('P(n,t)');
title(sprintf('distribution function at t=%i',i*t_sample));
pause;
end
end
else
% plot one realization
plot(t_real(1:sample2-1),N(1:1:sample2-1)); axis tight;
end

%%% for payroll process
for i=1:t_save,
% attention: we have to correct for t_sample > 1 !!
S(i)=mean(sum(N(:,1:i),2)*t_sample);
S_std(i)=std(t_sample*sum(N(:,1:i),2));
end
errorbar(0:t_sample:t_end,S(1:t_save),S_std(1:t_save),'r');
axis tight; xlabel('time t'); ylabel('total amount of wages paid');
title('total employee wages');

C.4.2 Quantum Harmonic Oscillator

function [g,r]=qmharmonicmaster(n)
% g=beta*n;
% r=alpha*n;
alpha=0.45;
5 beta=0.4;
g=beta*n;
r=alpha*n;

C.4.3 Growth of competitive population

function [g,r]=nonlineargrowthmaster(n)
% g=beta*n;
% r=alpha*n+gamma*n*(n-1);
alpha=0.1; beta=1.1; gamma=0.01;
5 g=beta*n;
r=alpha*n+gamma*n*(n-1);

C.4.4 Random Telegraph Process

function [g,r]=telegraphmaster(n)
% g=b(1-n);
% r=a*n;
a=0.1; b=0.9;
5 g=b*(1-n);
r=a*n;

C.4.5 Monomolecular Chemical Reaction
function [g, r] = reactionmaster(n)
% g=A
% r=k n
k=1; A=100;
g=A;
r=k*n;

C.4.6 The Payroll Process

function [g, r] = payrollmaster(n)
% g=h
% r=l n
h=0.2; l=0.005;
g=h;
r=l*n;

C.5 Listings for Chapter 5

C.5.1 Johnson Noise

% sdeornstein - Program to generate expectation values of the
% Ornstein-Uhlenbeck process with the help of the stochastic
% Euler algorithm
clear; help sdeornstein; % Clear memory and print header
set(0, 'DefaultAxesFontSize', 16);

xstart = input('Enter initial value of x (0) - '); 
q = input('Enter value of drift q (1) - '); 
D = input('Enter value of diffusion constant D (1) - '); 
tend = input('Enter final time in s (4) - '); 
istep = input('Enter number of time steps (4) - '); 
deltat = input('Enter delta t in s ([0.2 0.15 0.1 0.05]) - '); 
nreal = input('Enter number of realizations (10000) - '); 
tstart =0;

m2=zeros(1, istep);
f2=zeros(1, istep);
t0=cputime;

% loop over different time steps
for is=1:istep

    nstep=tend/deltat(is);
    sigma=sqrt(D*deltat(is));
    muconst = 1 - q*deltat(is);
    
    % realizations loop
    for j = 1:nreal
        t=tstart;
        x=xstart;
        % generate vector of gaussian distributed random numbers
        dw=randn(1, nstep)*sigma;
        for i=1:nstep
            x=x*muconst + dw(i);
        end
        m2(is)=m2(is)+x*x;
        f2(is)=f2(is)+x^4;
    end
end
\begin{verbatim}
% end of realization loop
end
m2(is)=m2(is)/nreal;
f2(is)=f2(is)/nreal;
f2(is)=sqrt((f2(is)-m2(is)*m2(is))/nreal);
fprintf('Time step = %g \n', deltatis);
fprintf('mean value = %g and standard deviation = %g \n', m2(is),f2(is));

% end of loop of time steps
end
t1=cputime-0;
fprintf('CPU time = %g seconds \n', t1);

% plot results
errorbar(deltat,m2,f2);
title('Ornstein-Uhlenbeck process');
xlabel('time step');
ylabel('<x^2>');

subplot(2,1,2)
errorbar(time,xmes2,sdev2);
xlabel('time');
ylabel('<x^2>');

hold on;

p=polyfit(deltat,m2,1);
xp=[0 deltat];
yp=polyval(p,xp);
plot(xp,yp,'-');
hold off;
fprintf('The extrapolated value for deltat=0: <x^2> = %g \n', yp(1));
\end{verbatim}

C.6 additional Listings

C.6.1 Random Walk 1D

%%% Symmetric or asymmetric
%%% ID Random Walk for one particle
%%% 

5 hold off;
clg;
clear;
realizations =10;
10 N=input(' How many steps (1000) ? ');
p=input(' Probability for a step to the right (0.5) ? ');
step=input(' Use fixed stepsize of 1 (otherwise random stepsize in [0,1]) (1) ? ');
step_size=ones(N-1,1);
15 if (step==1)
   disp (' Stepsize is 1 !');
   step_max=1;
else
   step_max=input(' Maximum stepsize (1) ? ');
   disp (' Stepsize is random between [0,1] !');
end
t=cputime;

% let it walk ...  

% do the realizations  
for j=1:realizations,  
  % Start position  
  pos(j,1)=0;  
  
  % draw all random numbers at once  
  random=rand(N-1,1);  
  if ( step ==0)  
    step_size = step_max.*rand(N-1,1);  
  end  
  
  % do the steps  
  for i=2:N,  
    if ( random(i-1)>p)  
      pos(j,i)=pos(j,i-1)+step_size(i-1);  
    else  
      pos(j,i)=pos(j,i-1)-step_size(i-1);  
    end  
  end  

% plot the elapsed CPU time  
t=cputime→t;  
disp ( sprintf ('\n CPU Time : %f seconds', t));  

% plot the paths of the walker  
hold on;  
for j=1:realizations  
  plot ( pos(j,:));  
end  
plot ([1 N],[0 0],'r');

---

C.6.2 Random Walk 2D

%%% Symmetric  
%%% 2D Random Walk for one particle  
%%%  
5 hold off;  
clg;  
clear;  

color =['b', 'k', 'y', 'g', 'm', 'c'];  
10 realizations =10;  

N=input ( ' How many steps (1000) ? ');  
%p= input ( ' Probability for a step to the right (0.5) ? ');  
step = input ( ' Use fixed stepsize of 1 ( otherwise random stepsize in [0,1] (1) ? ');  
15 step_size =ones(N-1,1);  
if ( step ==1)  
  disp (' Stepsize is 1 !');  
  step_max=1;  
else  
20   step_max = input ( ' Maximum stepsize (1) ? ');  
   disp (' Stepsize is random between [0,1] !');
APPENDIX C. LISTINGS FOR THE EXERCISES

C.6.3 Self-Avoiding Random Walk 2D

t=cputime;
25  % let it walk ...

    % Start positions and array definition
    pos(1:realizations,1:N,1:2)=0;

30  % do the realizations
    for j=1:realizations,
        % draw all random numbers at once
        random=floor(rand(N-1,1)*3.999)+1;
35  if  ( step==0)
            step_size=step_max.*rand(N-1,1);
        end

        % do the steps
    for i=2:N,
        % right
        if  ( random(i-1)==1)
            pos(j,i,1)=pos(j,i-1,1)+step_size(i-1);
            pos(j,i,2)=pos(j,i-1,2);
        end
        % left
        elseif ( random(i-1)==2)
            pos(j,i,1)=pos(j,i-1,1)-step_size(i-1);
            pos(j,i,2)=pos(j,i-1,2);
        end
        % up
        elseif ( random(i-1)==3)
            pos(j,i,2)=pos(j,i-1,2)+step_size(i-1);
            pos(j,i,1)=pos(j,i-1,1);
        end
        % down
        elseif ( random(i-1)==4)
            pos(j,i,2)=pos(j,i-1,2)-step_size(i-1);
            pos(j,i,1)=pos(j,i-1,1);
        end
    end

60  end

    % plot the elapsed CPU time
    t=cputime-t;
    disp(sprintf('
CPU Time : %f seconds',t));

70  % plot the paths of the walker
    hold on;
    axis square;
    maximum=max(max(max(abs(max(pos)),abs(min(pos)))))
    axis([-maximum maximum -maximum maximum]);
    for j=1:realizations
        plot(pos(j,:),pos(j,:),color(1+rem(j,6)));
    end

75  end

C.6.3 Self-Avoiding Random Walk 2D
C.6. ADDITIONAL LISTINGS

%%% Self—avoiding Symmetric
%%% 2D Random Walk for one particle
%%% Attraction problem count
5

hold off; clear;
color = ['b', 'k', 'y', 'g', 'm', 'c'];

10 realizations = input('How many realizations (1)? ');
N = input('How many walker steps (200)? ');
len = input('Length of the polymer (200)? ');
%p = input('Probability for a step to the right (0.5)? ');
step_size = 1; disp('Stepsize is 1!'); step_max = 1;
15 t = cputime;
% let it walk ...

%% Start positions and array definitions
20 xpos(1:realizations,1:N+1) = 0; ypos(1:realizations,1:N+1) = 0;
trapped(1:N+1) = 0;
attraction(1:realizations) = 0;

%% do the realizations
25 for j = 1:realizations,
    % do the steps
    trap(1:4) = 1; flag = 0;
    while (sum(trap)==4 & flag == 0)
        attraction(j) = attraction(j) + 1;
        for i = 2:N+1,
            if (random == 1)
                xpos(j,i) = xpos(j,i-1) + step_size;
            elseif (random == 2)
                xpos(j,i) = xpos(j,i-1) - step_size;
            elseif (random == 3)
                ypos(j,i) = ypos(j,i-1) + step_size;
            elseif (random == 4)
                ypos(j,i) = ypos(j,i-1) - step_size;
            end

    end
end
% check self avoiding up to length "len"
flag = 0;
for k=i-1:i:max(1,(i-1)-len),
    if (xpos(j,i)==xpos(j,k) & ypos(j,i)==ypos(j,k))
        flag = 1;
    break;
end
end
%% Check if trapped
if (flag == 1 & sum(trap)==4)
    disp(sprintf('I am trapped and cant escape after %i steps!', i));
    trapped(i)=trapped(i)+1;
    flag = 0;
end
end
if (sum(trap)==4)
    break;
end
end
end
end
distflag = 0;
for j=1:realizations
    plot(xpos(j,:),ypos(j,:),color(1+rem(j,6)));
    % mark the endpoint
    plot(xpos(j,end_point(j)),ypos(j,end_point(j)),'rs');
end
title('Self Avoiding Random Walk in 2 Dimensions');

% plot the number of trapped walks
figure;
plot(1:N+1,trapped(1:N+1),'d');
xlabel('trapped after steps');
ylabel('number of trapped walks');

% Atrition Plots
hist(attrition,20);
title('Attrition problem');
xlabel('number of trials per realization');
disp(sprintf('mean number of walks before no trapping occurs: %f + %f',... 120 mean(attrition), std(attrition)/sqrt(realizations)));
Appendix D

Solutions to exercises

D.1 Solutions for Chapter 1

Solution to exercise 4.1

Photoabsorption

Solution to exercise 4.2

Monte-Carlo Integration – Speed and Accuracy – photoabsorption.m

1. Hit and Miss Method – hitandmiss.m, hitandmiss2.m

   The estimate for the integral is \( I_i = n_i/n \), where \( n_i \) are the number of hits in one realization. By doing \( N \) realizations of the method, you estimate the integral by using the mean of the generated \( I_i, i = 1, \ldots, N \). The mean is estimated by the well known formula

\[
\bar{I} = \frac{1}{N} \sum_{i=1}^{N} I_i
\]

(Use the function mean() from Matlab.) The error of the individual trial \( I_i \) gets estimated by using the standard deviation. We estimate the standard deviation \( \sigma_i \) by using

\[
\sigma_i = \frac{1}{N-1} \sum_{i=1}^{N} (I_i - \bar{I})^2
\]

(We can use the Matlab function std() for calculating the standard deviation.) This is the error of the individual trial \( I_i \), not the error of the mean value \( \bar{I} \) calculated above. To that end, we have to use the central limit theorem and get

\[
\sigma_T = \sigma_i / \sqrt{N}
\]

Some example outputs of the program:

left: One realization using up to 5000 points in steps of 50.
right: Up to 10000 realizations using stepsize 100 and 50 points for each realization.
2. Standard Method – \texttt{mcistandard.m}

Now we use the formula \( \hat{I} = \frac{1}{n} \sum_{i=1}^{n} f(x_i)(b-a) \) to estimate the integral. The mean value is
\[
\bar{I}_n = \frac{1}{n} \sum_{i=1}^{n} I_i
\]

and the standard deviation of the individual \( I_i \) is
\[
\sigma_I = \frac{1}{N-1} \sum_{i=1}^{N} (I_i - \bar{I}_N)^2 = \sigma_{\bar{I}} \times \sqrt{n}
\]

where we have used the central limit theorem for the last equation to get the error of the mean \( \sigma_{\bar{I}} \).

Now we calculate several \( (N) \) realizations of the above algorithm to get a better estimate of the integral. The mean of the ensemble of realizations is
\[
\bar{I}_N = \frac{1}{N} \sum_{i=1}^{N} I_i
\]

And the error of the mean is
\[
\sigma_{\bar{I}_N} = \frac{1}{N-1} \sum_{i=1}^{N} (\bar{I}_i - \bar{I}_N)^2 = \sigma_{\bar{I}} / \sqrt{N}.
\]

Again we have used the central limit theorem and assumed that we have a “representative” \( \sigma_{\bar{I}} \) to get the last equality.

Some example outputs of the program:

left: Using increasing ensemble size from 1 to 5000 in steps of 100. Always using 50 points in the interval \((a,b)\).

right: The standard deviation and the distance to the exact result for the same run as in the left figure.

The CPU time used plotted versus the accuracy of the estimate.
SOLUTION TO EXERCISE 4.3

Eulers Constant using Monte-Carlo Algorithm – darts.m

SOLUTION TO EXERCISE 3.3

The Standard Deviation - variance.m
As you may soon recognize, the second and the third formula produce by far the biggest errors, even for small samples using very harmless distributions. But the first and the fourth method seem to perform equally well.

For a good and extensive discussion of this problem see the good paper by [Chan et al., 1983]. They discuss also some other algorithms for calculating the variance.

D.2 Solutions for Chapter 2

SOLUTION TO EXERCISE 3.1

Random-Number Generator Check - momentsrand.m, pokertest.m

- Moments of the rand() function of Matlab
  Example output for the first 10 moments of rand() using 5000 random numbers. Shown are the mean moments of the ensemble, the error (standard deviation) of the mean and in the last plot the distribution as a histogram using 50 bins.
Poker Test
Shown are the results for a Poker test using 20000 hands, each using 5 random numbers (=cards). Only no pairs (=0), pairs (=1), three of a kind (=3) and four of a kind (=4) are counted (5 of a kind is obviously not allowed). The numbers above the bars are the actual number of hands found in the ensemble. The second plot shows the difference of the probability in our ensemble to the correct theoretical result.

Histogram of the distribution

Poker test for the random number generator

Error of the generated hands
The connection to the Pascal triangle is obvious, if you know how to get the number of balls in the next row (say n+1) from the number of balls in the boxes of the previous row (say n). That's exactly like in the Pascal triangle:

\[
\begin{array}{c}
1 \\
1 1 \\
1 2 1
\end{array}
\]
You start with row 2 of the Pascal triangle, which corresponds to one pin (1 row) in the Galton board. The sum in each row is $2^N$ in row $N$. Therefore the probability for each box of a row of the Galton board is just the corresponding number of the Pascal triangle divided by $2^N$.

### D.3 Solutions for Chapter 3

**Solution to Exercise 5.1**

**Random-Number Generator - linear_con.m**

All plots use the linear congruential method with the parameters given in the assignment. The input parameters are: Initial seed 1 and 1000 random numbers are generated.

The first plots show the generated random numbers, the second ones show the histogram of the distribution. The third ones show 2D vectors and the fourth ones 3D vectors from the generated sequence of random numbers.

- parameter set 1

- parameter set 2
At first you don’t see any difference between the two and you don’t think of any problems. But if you plot the 3D vectors and use the `rotate3d` command with Matlab to rotate the 3D plot, you can see plots like the ones below. On the left the plot is rotated until you see the planes and on the right you wouldn’t expect correlations. Both plots use the same sample of 5000 random numbers. This clearly visualizes the correlation between successive triple random numbers.

**SOLUTION TO EXERCISE 5.2**

Poisson distribution - `poisson.m`

The first figure shows the generated Poisson distributed random numbers. The second figure shows two dimensional
vectors of Poisson distributed random numbers. The last (third) figure finally shows the histogram of the generated sequence and the exact Poisson distribution.

The parameters for this run have been:

$$\lambda = 10 \quad \text{and} \quad N = 1000.$$
then the sum $S_N := X_1 + \cdots + X_n$ has a density 

$$g_n(x) = \frac{(\lambda x)^{n-1} e^{-\lambda x}}{(n-1)!}.$$ 

(This is a so called gamma density.) The proof can be done by induction. For the exponential distribution function $F(x) = 1 - e^{-\lambda x}$ we get the distribution function of $S_N$ as 

$$G_n(x) = 1 - e^{-\lambda x} \left( 1 + \frac{\lambda x}{1!} + \cdots + \frac{(\lambda x)^{n-1}}{(n-1)!} \right).$$

The second step is to introduce a new family of random variables $N(t)$: $N(t)$ is the number of indices $k \geq 1$ such that $S_k \leq t$, $S_k$ is the sum of exponentially distributed r.v. from above.) Because $S_k$ has the distribution $G_n(x)$ (see theorem above), the probability of events $\{N(t) = n\}$ has the distribution 

$$P \{ N(t) = n \} = G_n(t) - G_{n+1}(t) = e^{-\lambda t} \frac{(\lambda t)^n}{n!}.$$ 

That is exactly a Poisson distribution!

For details see [Feller, 1971, page 8-12].

**Solution to Exercise 5.3**

**Acceptance-Rejection-Method - rejection.m**

First of all we have to note that the volume of the $n$-dimensional sphere is given by 

$$V_{R,n} = R^n \times V_{1,n} = R^n \times \frac{\pi^{n/2}}{\Gamma(n/2+1)},$$

where $\Gamma(x)$ is the well known Gamma function:

$$\Gamma(x) := \int_0^\infty t^{x-1} e^{-t} dt = \lim_{n \to x} \frac{n!}{x(x+1)(x+2)\cdots(x+n)} \quad \text{for } x \neq 0.$$ 

(for all real numbers, except $0,-1,-2,\ldots$. We have $\Gamma(x+1) = x\Gamma(x)$ and therefore for natural numbers $n > 0 \Gamma(n+1) = n!$. As you can see in the figure, the volume of the sphere (red diamonds/line) is decreasing by going to higher dimensions (keeping the radius $R = 1$ constant).

You get the surface of the sphere by just differentiating the formula above and get (see blue diamonds/line) 

$$S_{R,n} = \frac{d}{dR} V_{R,n} = nR^{n-1} \times \frac{\pi^{n/2}}{\Gamma(n/2+1)}.$$
The blue lines represent the sample mean and the sample variance and the red line is the exact value. Both figures are made using 500 realizations each drawing 10000 points in the n-dimensional space. The result for the left figure was $V = 4,1910 \pm 0,0017$ (exact $V = 4,1888$), for the right figure we got $V = 2,534 \pm 0,022$ (exact $V = 2,550$).

The algorithm for the surface of the hyper sphere is exactly the same, except that in the case of an accepted hit, you have to divide the accepted coordinates by the radius $R$. This gives you a point on the surface of the sphere.

As you can see from the plots above, the method is getting worse with increasing dimension, because $V_{\text{hypersphere}}/V_{\text{hypercube}}$ is going to zero for large $n$.

**SOLUTION TO EXERCISE 5.4**

**Importance Sampling - importance.m**

The idea of the importance sampling here is instead of sampling uniform random numbers and putting it into the function, we use normally distributed random numbers.

A very good book about importance sampling is [Kalos und Whitlock, 1986, Chapter 3 and 4.1].

- Standard sampling:

  $$ I_S = \frac{1}{N} \sum_{i=1}^{N} f(\xi), $$

  where the $\xi$ are from a uniform distribution.

  Easy to implement and understand, but the error is usually very big.
D.3. SOLUTIONS FOR CHAPTER 3

- Importance sampling:
  Here we suggested to use a normal distribution instead of the uniform one for the $\xi$. Then the formula reduces to:

$$I_i = \frac{1}{N} \sum_{i=1}^{N} |\xi_i| \times \frac{\sqrt{2\pi \sigma^2}}{2} \times \frac{1}{\sqrt{2}} = \frac{\sqrt{\pi}}{2N} \sum_{i=1}^{N} |\xi_i|.$$ 

First of all you have to realize that the $\texttt{randn()}$ function of Matlab produces normally distr. random numbers with mean $\mu = 0$ and variance $\sigma^2 = 1$. It also produces random numbers in the open interval $(-\infty, +\infty)$ and not, like desired, in the interval $[0, +\infty)$.

To correct for that, use the fact, that

$$\int_{-\infty}^{\infty} ve^{-v^2} dv = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{+\infty} \sqrt{\pi} e^{-v^2} dv,$$

(because the function is symmetric with respect to the y-axis.) Matlab produces numbers from

$$p(x) = \frac{1}{\sqrt{2\pi \sigma^2}} \exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right), \mu = 0, \sigma^2 = 1.$$ 

Now we have to transform this density to one with variance $\sigma^2 = 1/2$, which then has the form appearing in our integral ($e^{-v^2}$). We just divide the random number generated with $\texttt{randn()}$ by $\sqrt{2}$.

Then we accept only random numbers falling in the interval $[-c, c]$, because we are integrating over that interval (we have to take this into account, when calculating moments). Actually we do not need that restriction, but it demonstrates some additional complications, which could arise in actual problems.

The last step is to correct for the additional factor $\sqrt{2\pi \sigma^2}$ introduced by the normal distribution, and the factor $1/2$ because of the extended interval. This gives an overall factor of these two corrections of $\sqrt{\pi}$.

The first two figures show a run with 15,000 points (normally distributed random numbers) and a maximum cut-off $c_{\text{max}} = 10$. That means I have done many runs with increasing $c_{\text{max}}$. The red boxes indicate the results of the importance sampling, the blue line represents the exact result.

The figure on the right gives the systematic error involved, if the cut-off $c$ is used for the calculation.

On the left I have plotted the function to be integrated. On the right there is again a result of a run using 1000 points: the blue line is the exact result, the red boxes are the importance sampling results and the black boxes are the simple sampling results.
short discussion about (exact) variance:
The variance of the importance sampling method can be analytically calculated for this case. The general formula is

\[ \sigma^2 = \int_a^b \frac{g^2(x)}{f(x)} f(x) dx - \mathcal{T}^2, \]

where

\[ \mathcal{T} = \int_a^b \frac{g(x)}{f(x)} f(x) dx \]

is the integral we are looking for (This is just the second moment minus the first moment squared, as usual.).

In our case the first moment (the solution) is just 1/2 for the interval 0 to \( \infty \). For an interval from 0 to \( c \) we have

\[ \int_0^c ve^{-v^2} dv = 1/2 - \frac{c^2}{2}. \]

The second moment is just

\[ \int_0^c v^2 e^{-v^2} dv = \frac{2e^{-c^2} c^{3/2}}{\sqrt{\pi}} - \sqrt{\pi} \text{erf}(\sqrt{c}) \frac{c}{\sqrt{2}}. \]

(\text{erf}(c) is the error function.)

With the help of the above formulas, we can discuss three possible sampling functions methods:

- we use as an example \( c = 10 \) -

1. the standard sampling using uniform deviates \( f(x) = \frac{1}{(b-a)} = 1/c \):

\[ \sigma^2 = \int_0^c v^2 e^{-v^2} \frac{1}{c} dv - \frac{1}{4} = 1,316642671. \]

2. use importance function \( f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \):

\[ \sigma^2 = \int_0^c v^2 e^{-v^2} \frac{1}{\sqrt{2\pi}} e^{-v^2/2} dv - \frac{1}{4} = 0,3545997878. \]

3. use importance function \( f(x) = e^{-v^2} \):

\[ \sigma^2 = \int_0^c v^2 e^{-v^2} e^{-v^2/2} dv - \frac{1}{4} = 0,1931134628. \]
As you can clearly see, the variance is greatly reduced by choosing an importance function very close to the function to be integrated. And of course the simple sampling Monte Carlo integration produces a big variance compared to the importance sampling method (here almost a factor of 4 to 6.).

**Solution to Exercise 5.5**

**First Passage Times (fpt) - first_passage.m**
The left figure shows a run using $R = 5$ and 10,000 walks. The mean first passage time is 30,03 steps.

On the right, I have done many runs with different $R$ and calculated the mean first passage time for each run. You can see the exponential growth of the mean fpt.

**Solution to Exercise 5.6**

**Scaling Behavior of Random Walk in 2D and 3D - rw_scaling.m**
The function to be fitted is:

\[ \langle R^2 \rangle = aN^b \]

For the plots below we used 100 realizations for each length $N$, We started with length 50 and the last length calculated was 2000.

Results of the 2D and 3D calculations:

- 2D scaling: $a = 1.1051$ and $b = 0.9833$, 

---

D.3. SOLUTIONS FOR CHAPTER 3

429
• 3D scaling: $a = 0.9829$ and $b = 1.0018$.

A second run was made, using 500 realizations for each length of the random walk. We started with length 10 and went up to length 10,000 (using 50 (in 3D) and 30 (in 2D) intermediate lengths). It took a total of 20,900 seconds to calculate on a Pentium 200MMX processor (Matlab 5.0).

For the long runs, the results are:
• 2D scaling: $a = 0.996679$ and $b = 1.001098$,
• 3D scaling: $a = 1.005113$ and $b = 0.999942$.

**Solution to exercise 5.7**

**Percolation in 2D and Cluster Algorithms - percolation.m**

The first two figures show $20 \times 20$ configurations with and without a spanning cluster. For the left figure we used $p = 0.5$ and for the right figure $p = 0.55$.

The following two figures are created using:
1. figure on the left:
   $N = 20 \times p$ from 0.3 to 0.9 with stepsize 0.02 and 20 realizations each.
2. figure on the right:
   $N = 30 \times p$ from 0.3 to 0.9 with stepsize 0.02 and 20 realizations each.
To estimate the critical probability $p_c$, you have to extract $p_c$ from the figures above. There are many ways of doing it. One would be to fit a sigmoidal function

$$p(x,a) = \left( \frac{1}{1 + e^{-a(x-0.5)}} + 1 \right)/2,$$

or a tangent hyperbolicus

$$p(x,a) = (\tanh(a(x/2-0.5)) + 1)/2 = \left( 1 + \frac{1 - e^{-a(x-0.5)}}{1 + e^{-a(x-0.5)}} \right)/2.$$

$a$ is the parameter to fit.

**SOLUTION TO EXERCISE 5.8**

The Einstein Solid and the Boltzmann Distribution - einstein_solid.m

A typical 20x20 plane of the Einstein solid after 10000 jumps.

```
1 2 0 0 0 3 5 2 0 1 1 1 0 0 1 1 2 0 1 0
2 0 2 1 4 1 1 0 1 0 2 2 0 1 0 4 0 0 1 0
3 1 3 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 1
0 1 0 0 0 0 1 2 1 1 0 0 1 1 1 2 0 0 0 1
0 0 6 1 0 4 0 0 1 1 1 0 0 0 0 0 0 0 0 2
0 0 2 1 9 3 0 1 2 1 0 0 3 0 6 1 2 0 0 0
0 2 2 1 0 2 0 0 0 0 0 2 4 0 0 0 0 0 1 1
0 0 0 6 1 0 0 6 1 1 0 0 0 0 0 0 3 0 1 0
2 0 1 0 0 2 0 1 0 2 0 0 0 1 3 0 0 1 0 3
0 1 1 0 2 0 0 0 0 1 0 2 1 2 2 1 0 2 2 1
1 0 2 0 1 1 2 0 0 4 1 1 3 0 2 0 4 1
1 0 0 0 1 0 3 2 0 0 0 0 0 0 0 1 0 1 1
0 2 2 0 1 3 0 2 1 2 1 0 0 1 2 0 0 2 2 3
0 2 0 0 0 3 2 0 1 3 0 0 0 2 0 0 0 1 1 4
2 0 0 0 0 5 0 3 0 0 0 1 0 0 0 2 2 2 1 2
0 4 1 0 5 0 0 0 1 0 0 1 4 0 5 2 0 0 0
1 0 1 2 0 0 0 2 0 5 0 1 2 1 0 2 0 0
0 0 0 1 0 1 0 0 1 5 0 4 0 0 0 4 1 0 4 1
0 1 0 3 4 0 1 2 1 5 0 3 9 0 0 1 0 3 0 2
1 3 1 0 7 0 0 2 0 0 1 0 0 2 0 3 0 1 0 2
```
On the next 9 figures you can see the time development of a 20x20 grid after 10, 20, 30, 40, 50, 60, 70, 150 and 20000 steps (or jumps). Although it is a contour plot and not the whole cell is filled with the same color (which would make it easier to view), you can still see the pile up of quantas in individual cells. On the other side more and more cells are unoccupied and after reaching equilibrium we are left with most of the cells unoccupied.

The last figure in this sequence is a semilogarithmic plot of the distribution of number of quantas in the cells (red curve). The blue one is a least square fit to the Boltzmann distribution, which is in excellent agreement.
As an example we have also done a calculation with a 40x40 plane. On the left is the filled contour plot and on the right is the same configuration using a simple contour plot.

D.4 Solutions for Chapter 4

Solution to Exercise 6.1
Linear one-step process - quantized harmonic oscillator in a radiation field - onestepfast.m, qmhar-moscimaster.m

First of all we want to summarize some of the results for the one-step processes (also called birth-and-death processes). They are governed by the following Master-equation (the phase space is discrete)

\[ \frac{d}{dt} P(n,t) = r(n)P(n-1,t) + g(n)P(n+1,t) - (r(n) + g(n))P(n,t). \]

The solutions are continuous in time, in contrast to the random walks discussed in the first few chapters. We can rewrite the Master-equation, using the transition matrix

\[ \mathbb{M}_{n'n} := g_{n'n} \delta_{n'n'-1} + r_{n'n} \delta_{n'n'+1} \quad \text{for} \quad n \neq n', \quad \text{and} \]

\[ \mathbb{M}_{nn} := -(r_n + g_n), \]

as

\[ \frac{d}{dt} P(n,t) = \mathbb{M}_{nn'} P(n',t), \]

There are three classes of one-step processes, depending on the interval of \( n \):

1. \((-\infty, +\infty)\), no limits on \( n \).
2. \([0, +\infty)\), one-sided open interval.
3. \((0, N)\), finite interval.

We can take these boundary conditions into account by supplying the correct equations for the boundaries, or in most cases it is sufficient to just set e.g. \( r(0) = 0 \) and \( g(\infty) = 0 \) or e.g. \( r(N+1) = g(N) = 0 \).

The exact equation for the first moment is:

\[ \frac{d}{dt} \langle n \rangle = - \langle r(n) \rangle + \langle g(n) \rangle, \quad \text{FIRST MOMENT} \]

and for the second moment

\[ \frac{d}{dt} \langle n^2 \rangle = 2 \langle n(g(n) - r(n)) \rangle + \langle r(n) + g(n) \rangle, \quad \text{SECOND MOMENT} \]

With these two solutions we can easily calculate the equation for the variance.

There are some interesting special cases, depending on the form of the gain \( g(n) \) and the loss \( r(n) \) probability:

<table>
<thead>
<tr>
<th>( g(n) )</th>
<th>( r(n) )</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>const.</td>
<td>0</td>
<td>Poisson Process</td>
</tr>
<tr>
<td>( q \equiv \text{const.} )</td>
<td>( q \equiv \text{const.} )</td>
<td>symmetric RW (diffusion)</td>
</tr>
<tr>
<td>( q_1 \equiv \text{const.} )</td>
<td>( q_2 \equiv \text{const.} )</td>
<td>asymmetric RW (diffusion with external force)</td>
</tr>
<tr>
<td>0</td>
<td>( \gamma n )</td>
<td>radioactive decay</td>
</tr>
<tr>
<td>( q \equiv \text{const.} )</td>
<td>( \alpha n )</td>
<td>monomolecular chemical reaction</td>
</tr>
<tr>
<td>higher orders in ( n )</td>
<td>higher orders in ( n )</td>
<td>general linear (e.g. QM harm. osc., payroll)</td>
</tr>
<tr>
<td>higher orders in ( n )</td>
<td>higher orders in ( n )</td>
<td>nonlinear (e.g. competitive growth)</td>
</tr>
</tbody>
</table>

Remark: sometimes one refers to the nonlinearity of \( g(n) \) and \( r(n) \) by calling it a nonlinear Master-equation, but do not forget that the Master-equation is always linear in \( P(n,t) \)! For the linear case, one must have at least one boundary, otherwise the transition probabilities \( g(n) \) or \( r(n) \) become negative. But most of the physical systems have a natural boundary, like the radioactive decay at \( n = 0 \).

The stationary solution \((t \to \infty)\) is given by:

\[ P_{\text{stat}}(n) = \frac{\prod_{i=0}^{n-1} g(i)}{\prod_{i=1}^{n} r(i)} P_{\text{stat}}(0) \quad \text{for} \quad n \geq 0, \]

and

\[ P_{\text{stat}}(n) = \frac{\prod_{i=0}^{n+1} r(i)}{\prod_{i=1}^{n} g(i)} P_{\text{stat}}(0) \quad \text{for} \quad n < 0, \]
The first solution is for the class 2 and 3 above and the second solution is only for the class 1. $P_{\text{stat}}(0)$ is given by the normalization condition, e.g. for class 2

$$\sum_{n=0}^{\infty} P_{\text{stat}}(n) = 1,$$

If we take the Fokker-Planck limit for the Random Walk (both transition rates constant) we get a diffusion equation. The same limit for the Poisson process (no loss transition) does not exist!

For the quantized harmonic oscillator we have for the first moment (The initial condition will be always $n(0) = N_0$)

$$\frac{d}{dt} \langle n \rangle = (\beta - \alpha) \langle n \rangle \Rightarrow \langle n(t) \rangle = N_0 e^{(\beta - \alpha)t} \quad \text{and}$$

$$\frac{d}{dt} \langle n^2 \rangle = 2(\beta - \alpha) \langle n^2 \rangle + (\alpha + \beta) \langle n \rangle,$$

which has the solution

$$\langle n^2(t) \rangle = \frac{\alpha + \beta}{\alpha - \beta} N_0 e^{(\beta - \alpha)t} \left(1 - e^{(\beta - \alpha)t}\right) + N_0^2 e^{2(\beta - \alpha)t}.$$

(Remark: As you can see, we have a closed hierarchy of moment equations.)

The stationary solution is

$$P_{\text{stat}}(n) = \text{const.} \cdot \left(\frac{\beta}{\alpha}\right)^n.$$

Here are some results of the simulation:

- $\alpha = \beta = 0.5, N_0 = 50, t_{\text{END}} = 500$, 1 realization, $\Delta t = 1$

- $\alpha = \beta = 0.5, N_0 = 50, t_{\text{END}} = 100$, 100 realizations, $\Delta t = 2$

- $\alpha = 0.52, \beta = 0.48, N_0 = 500, t_{\text{END}} = 150$, 1 realization, $\Delta t = 1$
\[ \alpha = 0.51, \beta = 0.49, N_0 = 50, \Delta t = 100 \text{, 100 realizations} \]

\[ \alpha = 0.53, \beta = 0.47, N_0 = 50, \Delta t = 50 \text{, 300 realizations} \]

\[ \alpha = 0.50, \beta = 0.51, N_0 = 10, \Delta t = 50 \text{, 200 realizations} \]
D.4. SOLUTIONS FOR CHAPTER 4

Solution to Exercise 6.2

Non-linear one-step process - growth of a competitive population - onestepfast.m, nonlingrowthmaster.m

This time the transition rates are $r(n) = \beta n$ and $g(n) = \alpha n + \gamma n(n - 1)$. The solutions for the first moment are

$$< n(t) > = \frac{\beta - \alpha}{\gamma} \quad \text{and} \quad < n(t) > \equiv 0.$$  

Here are some examples of the simulation:

- LEFT: $\alpha = 0.50, \beta = 1.0, \gamma = 0.05 \ N_0 = 100, t_{\text{END}} = 500, 1 \text{ realization}, \Delta t = 2$ and
- RIGHT: $\alpha = 0.1, \beta = 1.1, \gamma = 0.01 \ N_0 = 120, t_{\text{END}} = 30, 1 \text{ realization}, \Delta t = 1$

- $\alpha = 0.1, \beta = 1.1, \gamma = 0.01 \ N_0 = 120, t_{\text{END}} = 10, 100 \text{ realizations}, \Delta t = 1$

- $\alpha = 0.1, \beta = 1.1, \gamma = 0.01 \ N_0 = 120, t_{\text{END}} = 10, 100 \text{ realizations}, \Delta t = 1$
SOLUTION TO EXERCISE 6.3

The Random Telegraph Process - onestepfast.m, telegraphmaster.m

First of all you have to rewrite the master-equation into the known form. Therefore you have to identify the transition rates as 

\[ r(n) = an \quad \text{and} \quad g(n) = b(1-n). \]

Some analytical results:

\[ \langle n(t) \rangle = \langle n^2(t) \rangle = \frac{a}{a+b}. \]

Here are some results of the simulation:

- LEFT: \( a = 0.1, b = 0.9, N_0 = 1, t_{\text{END}} = 10, 1 \text{ realization}, \Delta t = 1 \)
- RIGHT: \( a = 0.1, b = 0.9, N_0 = 1, t_{\text{END}} = 100, 1 \text{ realization}, \Delta t = 1 \)
• $a = 0.1, b = 0.9, N_0 = 1, t_{\text{END}} = 50, 100$ realizations, $\Delta t = 1$
  You can see that the first and the second moment have the same value, like expected by the analytical results. The probability distribution in the last figure shows $P(0)$ and $P(1)$.

• $a = 0.1, b = 0.9, N_0 = 1, t_{\text{END}} = 10, 100$ realizations, $\Delta t = 0.2$
  The left figure shows a patch plot of the correlation matrix. The colorbar identifies the colors with the corresponding values of the correlation matrix. The red line indicates the normalized second moment $<n(t)n(t')>$ $= <n^2(t)>$. The right figure displays five correlation functions with a fixed $t$ and varying time difference $t - t'$. As expected the correlation decays exponentially and the correlation length is about $1/(a+b)$ (see assignment).

SOLUTION TO EXERCISE 6.4
Monomolecular Chemical Reaction \( A \rightleftharpoons X \) - onestepfast.m, reactionmaster.m

First of all let us calculate the stationary distribution here as an example.

\[
P_{\text{stat}}(n) = \frac{\prod_{i=0}^{n-1} g(i)}{\prod_{i=1}^{n} r(i)} P_{\text{stat}}(0) = \frac{A^n}{n! k^n} P_{\text{stat}}(0),
\]

and

\[
\sum_{i=0}^{\infty} P_{\text{stat}}(i) = P_{\text{stat}}(0) + \sum_{i=1}^{\infty} P_{\text{stat}}(i) = P_{\text{stat}}(0) + \sum_{i=1}^{\infty} \frac{A^i}{i! k^i} P_{\text{stat}}(0) = P_{\text{stat}}(0) \left( 1 + (e^{Ak} - 1) \right) = P_{\text{stat}}(0) e^{Ak},
\]

and therefore

\[
P_{\text{stat}}(0) = e^{-Ak} \Rightarrow P_{\text{stat}}(n) = \frac{A^n}{n!} e^{-Ak}.
\]

The rest of the analytical results are given in the assignment, we do not repeat them here.

Here are some examples of the simulation:

- \( k = 1, A = 100, N_0 = 100, t_{\text{END}} = 80, 1 \) realization

![Graph 1](image1)

- \( k = 1, A = 100, N_0 = 100, t_{\text{END}} = 50, 100 \) realizations, \( \Delta t = 1 \)

![Graph 2](image2)
The red line in the right figure is a Poisson distribution (which is the analytical result for the stationary distribution for this process.) with mean 100. You can see the excellent agreement with the simulated curve.

D.5 Solutions for Chapter 5

SOLUTION TO EXERCISE 7.1

Johnson Noise - sdeornstein.m

First of all you have to identify the parameters (functions) \( q \) and \( D \) of the SDE form used in the lecture:

\[
q = \frac{1}{\tau}, \quad \text{and} \quad D = \epsilon.
\]

Some simulation results:

- \( i_0 = 0, q = 1, D = 1, t_{\text{END}} = 1000, \Delta t = 0.2, 1000 \) realizations
- \( i_0 = 0, q = 1, D = 1, t_{\text{END}} = 100 \)

The last figure on the right is a simulation for different time discretizations \( \Delta t = 0.5, 0.2 \) and 0.1 using 2000 realizations and \( t_{\text{END}} = 100 \).
One realization of the process

Ornstein-Uhlenbeck process
D.6 Solutions for Chapter 6

Solution to Exercise ??

D.7 Solutions for Chapter 7
Bibliography


Appendix E

GNU Free Documentation License,
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
written document "free" in the sense of freedom: to assure everyone
the effective freedom to copy and redistribute it, with or without
modifying it, either commercially or noncommercially. Secondarily,
this License preserves for the author and publisher a way to get
credit for their work, while not being considered responsible for
modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense. It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does. But this License is not limited to software manuals; it
can be used for any textual work, regardless of subject matter or
whether it is published as a printed book. We recommend this License
principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a
notice placed by the copyright holder saying it can be distributed
under the terms of this License. The "Document", below, refers to any
such manual or work. Any member of the public is a licensee, and is
addressed as "you".

A "Modified Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use
technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions
(which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or
distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.
Appendix F

GNU GENERAL PUBLIC LICENSE,
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.
Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source
code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS
Index

2D plots, 99
3D plots, 110

Applet package, 86
barplots, 100
Biss-AWT, 10

C in Java, see native code in Java
C++ in Java, see native code in Java

class variables, 46
CLASSPATH, 22
classpath, 22
color models, 109
color setting, 94
complex numbers, 71
constants, 46
debugging, 75

Emacs, 9
empty package, 21
errorbar plots, 100

final keyword, 46
Fortran to Java, 9
Freebuilder, 9
function plots, 100

Gaussian random numbers, 83
GCJ, 8
GJ, 8
Gnuplot, 111
GuavaC, 8

histogram plots, 101
import statement, 21

Java 3D, 111
Java Numerical Library, 10
Java Workshop, 9
JDE, 9
JDK, 8, 64
documentation, 64
Jikes, 8
JNLI, 10, 71
complex numbers, 71
list of functions, 71

JSci, 10
Kaffe, 8
modifiers, 46, 47
mpEdit, 9
MRJ, 8

native code in Java, 75
Netbeans, 9
Netscape, 9

packages, 21
postscript files and \TeX, 109
printing from an applet, 109
Ptolemy, 10, 94
Ptplot, 94
ptplot, 10
public modifier, 46

random numbers
integer valued, 83

SciVis, 111
Simplicity for Java, 9
simulation package, 10, 99
Swing, 10
Swing and AWT, 86

TOBA, 9
tools for Java, 8

variable modifiers, 47
XEmacs, 9