Laboratory Class Scientific Computing

course book

The Scientific Computing Group, Utrecht University

March 17, 2015

Contents

List of Exercises

List of Algorithms

Introduction

This course book contains the material for the course 'Scientific Computing Laboratory Class' which is part of the Master Program of Mathematical Sciences at Utrecht University. The course is in the field of Computational Science and its application area, Scientific Computing. It is a 'hands-on' course which is intended to expose students to computer techniques used in scientific computation. In this year's course we will focus our attention on the topics of Monte Carlo integration and Genetic Algorithms. The aim of the course is to learn about a number of important Computational Science subjects, and to meet with the various aspects of Scientific Computing. You will write your own computer code and run simulations, handle output data and visualise it. The theory and results of the simulations are to be presented by means of written reports.

Written reports. For each of the subjects

1. Random Number Generators (RNGs) and Monte Carlo integration,
2. Genetic Algorithms

you have to write a report. Every student is individually responsible for his/her *own* report. We have included several hints on writing a good report in Appendix A. The deadlines for the reports are published on the course web site [36]. You are also required to send your source code by email to the instructor, which should compile with the `g++` compiler. If you have little experience with `C++`, then we recommend you team up with someone who has used `C++` before.

Assignments. Throughout the course there will be several small assignments to keep you 'on track'. Visit the website for details on this year's assignments. Assignments are no reports! You can use LaTeX of course, but pencil and paper is also fine. One or two pages should suffice.

This book. The course book is set up as follows, Chapter 0 starts with an introduction to the practical aspects of scientific computing on UNIX systems. If you are already familiar with this, at least still have a look at the hints (marked with ☞).

★ A paragraph or exercise with a ★ in the side margin is required, and should be part of your report on the subject of concern.

☞ A paragraph with a ☞ in the side margin is recommended. It usually contains hints to make life easier. Exercises marked as such are optional (unless otherwise specified; see course website [36]), but are interesting to have a look at.

Chapter 2 deals with the implementation of efficient random number generators. This chapter is meant as an introduction; the theory is not too difficult but you will learn the basics of Scientific Computing and `C++` programming by writing small programs. Probability Theory is central in the mathematics of the topic of this chapter and of the other two topics in this course. Chapter 1 gives a summary of the important notions and results from Probability Theory and Statistics that we will encounter in this course. The first main topic is Monte Carlo integration, presented in Chapter 3, which is a technique for computing integrals by 'averaging' over random samples. You will learn in which cases this technique may be competitive with traditional methods (as the Trapezoidal Rule), specifically in case of high dimensional integrals.

The second topic, covered in Chapter 4, is Genetic Algorithms for solving 'hard' optimisation problems. A set of possible solutions are represented as a population of genes where the 'better ones' (more 'optimal ones') have greater probability of 'survival'. Simulation of 'natural' evolution then yields a (possibly local) optimal solution.

Appendix A gives many hints on writing good reports, concerning both content and style. Appendix B describes some technical aspects of `C` and `C++` programming.

The Scientific Computing Group,[1]
Department of Mathematics,
Utrecht University, March 17, 2015

[1]This course is modelled after the 2003 course by Marciá Alves de Inda. We also thank Eduard Belitser for allowing us to use material from his course notes. In particular Section 2.4 and Chapter 3 benefited from this. Chapter 4 on genetic algoritms and the accompanying software was (re)written in 2005 by Arno Swart and Arthur van Dam. The code on Monte-Carlo simulation was updated in 2007 by Arthur van Dam and Albert-Jan Yzelman. Bas Fagginger Auer implemented corrections and updates (2011). Gerard Sleijpen revised in 2013 the 'mathematical' Chapters.

Chapter 0

Common Techniques

This chapter shortly considers a range of practical aspects of scientific computing. The following sections discuss the efficient use of a Linux system, C++ programming, and data analysis and visualisation with MATLAB.

0.1 System usage

This section gives some quick-start hints on using a Linux system efficiently. If you are already experienced with this, read the notes below and decide for yourself whether to use them or not.

0.1.1 Use a good shell

To get things done on Linux-like systems, you often enter commands in a terminal window. Whenever you need a terminal, use an existing one, or open a new one by right-clicking on your desktop and choosing '`Open Terminal`'.

☞ The program that handles the commands you enter, is called a 'shell'. We suggest making `bash` your default shell from now on, if it is not so already.[1] To do so, you only need to run the following command just once (do not type the `$`, that stands for the prompt):

```
$ chsh `whoami` `which bash`
```

(Note that these are all back-quotes (next to the `1` key).) Amongst others, you can now use the arrow up key to recall commands you issued before; this saves you a lot of (re-)typing. Also try pressing the `Tab` key for completing commands and file names.

0.1.2 Organise your files

☞ To avoid getting lost in all the different files by the end of this course, think of a good directory structure now. For example:

```
$ cd            ▷ This always takes you to your home dir; equivalent to ``cd ~''
$ mkdir labsci
$ cd labsci
$ mkdir rng         ▷ First topic is random number generators.
```

[1]To see your current shell, type `echo $SHELL` in a terminal.

0.1.3 Use multiple workspaces

The `gnome` environment that runs on our lab machines offers you multiple workspaces; use them. If you do not, you will end up with ten or more windows cluttered on top of each other.

☞ At the bottom of your desktop is the *workspace manager* to switch between workspaces. Alternatively you can switch to neighbouring workspaces using `alt+ctrl+←` and `alt+ctrl+→`. For example, use one workspace for a browser (`firefox`), one for your C-programming (editor and terminal), one for `matlab`, and—if necessary—one for writing your report.

0.1.4 Use a good editor

Choosing an editor is a matter of taste; if you already have a favourite one (`vim`, `emacs`, ...), use it. If not, we suggest `nedit`.

☞ Here are some hints for setting up `nedit` for the first time (you only need to do this once). Open a terminal and type:

```
$ nedit &
```

Next, issue the following commands through the menu bar (in `Preferences > Default Settings`):

```
Auto Indent > Smart
Tab Stops > Tab spacing: 8
Tab Stops > 'Use Tab characters...' checked
Syntax Highlighting > On
Statistics Line
Show Line Numbers
Show Matching (..) > Delimiter
```

followed by

```
Preferences > Save Defaults...
```

0.2 Developing and running C++ programs

During this course we use C++ as programming language. Most topics start with an existing set of source files that you should extend. If you have not worked with C++ before, these are illustrative examples of how basic programming works. There are basically four parts when developing scientific simulation software: programming, compiling, running, and (hopefully not) debugging. We will shortly discuss each of these now. Be sure to read Appendix B, it contains more technical details to help you understand and develop C(++) programs.

0.2.1 Programming in C++

Source files The simplest case of a C++ program is a single source file `example.cc`. For bigger programs, you can use multiple `.cc` files and compile them into one executable (See next section).

The main function The C++ code for a program should contain a function `int main`. When running your program, this function will be called (executed). In a clean program, not much code is within `main`. Put all (mathematical) functionality in separate functions and just call them from the `main` function.

Header files Header files (`example.h`) can contain function declarations and data type definitions. Compilers come with a set of useful functions, and you request to use them by including the appropriate header files:

```
#include <iostream>
using namespace std; //Makes all standard functionality available
```

It is also possible to use your own header files:

```
#include "myutils.h"
```

Notice the double quotes and the additional `.h`, necessary for non-system header files.

Object-oriented programming in C++ Object oriented (OO) programming is a technique that structures a program more neatly, based on what parts/entities play a role in the algorithm. Detailed implementation of certain subparts of an algorithm can be hidden for the rest of the program, resulting in a program that is easier to understand and maintain. Section B.3 gives an introduction into OO programming.

The C++ language In essence, C++ is pretty much like other imperative programming languages (e.g. Java, C). Many good reference material exists.

☞ If you are not familiar with this way of programming yet, here is some suggested reading material:

- The 'C++ Resources Network' also has an excellent tutorial on C++.
  `http://www.cplusplus.com/doc/tutorial/` [28].
- *C++*, Leen Ammeraal (in Dutch) [1].
- *Thinking In C++ 2nd Edition* by Bruce Eckel, a 2-volume excellent introduction to C++:
  `http://oopweb.com/CPP/Documents/ThinkingInCpp1/VolumeFrames.html`
  `http://oopweb.com/CPP/Documents/ThinkingInCpp2/VolumeFrames.html` [10, 11].
- *De kleine C gids* (in Dutch), a handy quick reference for all standard C libraries. Also available in English (original) as '*C Quick Reference*' [31]
- Appendix B contains own material, mainly on data types and pointers in C/C++. Contrary to Java, with C/C++ you have full (direct) control over computer memory, using pointers and addresses.
- Rob Pooley has two excellent online courses on C and C++. Highly recommended!
  `http://www.macs.hw.ac.uk/~rjp/Coursewww/` [32].
- For the more experienced programmers, `cppreference.com` is a reference on language features, the C Standard Library and the C++ Standard Template Library. `http://www.cppreference.com` [13].
- Also for advanced programmers, a well-written FAQ on more obscure C++ features, and good object-oriented C++ programming guidelines in general.
  `http://www.parashift.com/c++-faq-lite/` [4].

0.2.2 Compiling C++ programs

Before you can run your program, it needs to be *compiled*. The compiler makes an executable program out of your C++ source file(s). We use the `g++` compiler here.

To compile your program by hand, do the following:

```
$ g++ example1.cc somehelpfile.cc ... -o exampleprogram
```

☞ Usually, the compilation process is automated by a `Makefile`. We provide them with the exercises. To compile your program automatically, do the following:

```
$ make
```

If no files have changed, no new compilation will be done, as it is unnecessary.

You may notice that `make` runs slightly different compilation commands. When multiple files are involved, *separate compilation* is often used. This means that each source (`.cc`) file is compiled to an object (`.o`) file. As a last step, all object files are *linked* into one executable. The linker then again checks if all called functions are available. Separate compilation uses the `-c` flag to tell the compiler to compile only and not link:

```
$ g++ -c example1.cc -o example1.o
$ g++ -c somehelpfile.cc     ▷ somehelpfile.o is the default name
$ g++ example1.o somehelpfile.o -o exampleprogram
```

The advantage of this is that when one source file has changed (e.g. `example1.cc`), the other source file(s) need not be recompiled. This saves compilation time. Of course (re-)linking is always necessary.

The compiler gives error messages, now what?

Especially when you are not an experienced programmer, you are likely to make a lot of mistakes in your program in the early stages. They are very easy to fix in most of the cases. The compiler will print error messages in your terminal. Just take the time to *read and understand* the error messages in your terminal. Always, the line number is shown, so look it up in your program and find the mistake. below are some common examples:

Parse error

```
myfile1.cc: In function 'int main(int, char**)':
myfile1.cc:5: parse error before 'return'
```

In file '`myfile1.cc`', in the '`main`' function, there is an error *in or before* line 5. Usually, it is a forgotten '`;`', '`)`', or '\' in one of the preceding lines.

Undeclared variables or functions

```
myfile1.cc: In function 'int main(int, char**)':
myfile1.cc:5: 'w' undeclared (first use this function)
myfile1.cc:5: (Each undeclared identifier is reported only
   once for each function it appears in.)
```

A variable is used in an expression (e.g. '`w + 1`'), but it has not been declared yet. All variables that you use in your program should be declared. You may do so anywhere in the function as long as it is before the first use of the variable, but putting all declarations at the start is the cleanest.

The same error is also given for undeclared functions that are called. Always have your function declaration *before* its first use.

Conflicting types

```
myfile1.cc: In function 'int main(int, char**)':
myfile1.cc:17: warning: passing 'double' for argument
   passing 1 of 'void testi(int)'
myfile1.cc:17: warning: argument to 'int' from 'double'
```

When calling function ‘testi’ from with the ‘main’ function (at line 17), a variable of type double was used as argument, but the function expects a variable of type integer. Although this is ‘only’ a warning, you should fix it. Otherwise the system will round off your double value, which is probably not desirable.

0.2.3 Running the compiled program

Once compilation is successful, you have an executable file ‘myprogram’ in the same directory. Run it like this:

```
$ ./exampleprogram
```

Some programs need additional user input (e.g. specifying parameters for the algorithm), just add them on the command line:

```
$ ./someotherprogram 5 0.3
```

The `main` function receives these arguments as an array of strings.

0.2.4 Debugging your program

A successful compilation is no guarantee for correct results. For example, you might see your program crashing with:

```
Bus Error (core dumped)
```

In this case, something probably went wrong with memory pointers, check whether all ‘*’s and ‘&’s are correct.

Maybe your program runs fine, but produces wrong results. You might want to print out some intermediate results in your program. This is shown in the following code sample:

```
double err, tol;
err = 1e+9;
tol = getToleranceValue();

cout << "my debug, tolerance value = " << tol << endl;

while (err > tol) {
    // Do something
}
```

By choosing sensible printouts, you can follow what your program is doing, and where things start to go wrong.

More dedicated debugging can be done with the programs `gdb` or `valgrind`. When necessary, this can be demonstrated during the laboratory sessions.

0.2.5 Writing your results to a file

Your experiments will result in data, often in the form of long series of numbers or series of pairs of numbers. It is convenient to write this output to a file, in order to further process your data. In this section we will present the example function `print2file1d`, found in the file `util.cc`. Use this file to add your own functions that write data later.

The function `print2file1d` uses *streams* to write data to the disc. A stream is associated to a file (here `randdata.txt`) and we can use the << operator to write data to the stream. Examine the source file, you will see that the following actions were taken

1. The file is opened
2. Data is written to the stream associated with the opened file; first the number, then a whitespace
3. The file is closed
4. There is a check for possible errors

Later you will expand on this function and write your own output routines. The commands

```
$ g++ rand.cc util.cc RNG_factory.cc generators.cc
                                        LC_RNGs.cc EX_RNGs.cc  -o rand
$ ./rand 100
```

will now produce a hundred random numbers between zero and one and write them to a file `randdata.txt`.

0.3 Data visualisation with MATLAB

This section will explain how to get started using MATLAB for visualisation purposes. MATLAB is an interactive software package for handling a wide range of linear algebra problems. It is also possible to write programs in MATLAB, but in contrast to C++ it is an interpreted language (i.e. executed line by line and not compiled). We will use MATLAB primarily for visualisation, but we would like to refer the reader to [35] for more information on the general usage of MATLAB.

You may invoke an instance of MATLAB using

```
$ Matlab &
```

Depending on previous usage of MATLAB you will be presented a number of panels within the MATLAB window. Make sure to have the `editor` and `command` window open. Check if you are in the right directory using the `pwd` command. At any time you can get help on MATLAB commands using `help`, or the help item from the menu. There is also an excellent online help from the authors of MATLAB at [22].

In the following we assume that you have a file `randdata.txt` containing some numbers, created as explained in Section 0.2.5. MATLAB files have the extension `.m`, we prepared the simple code `plotrand.m` for you, which reads numbers from a file and plots them in various ways. Try it, enter the following in the command window,

```
>> plotrand('randdata.txt')
```

You will see two new windows named `Figure 1` and `Figure 2` with in them numbers from `randdata.txt` and a histogram. Try some of the zoom and edit tools from the menu in the figure window. Also look at the built-in help for `plot` and `hist` and see if you understand what happens.

specifically

Chapter 1

Probability Theory

Sequences of random numbers are central in this course. In Chapter 2, we discuss algorithms that 'aim' to construct such sequences. The sequences will be exploited in the subsequent chapters in methods for accurately solving problems that are computationally intensive. To gain some confidence in how successful we are in generating sequences of numbers that are random according to some prescribed distribution we will apply several statistical tests.

This chapter reviews the basics of probability theory that we will need.

1.1 Sample spaces and probability measures

Probability theory is about the study of random processes. If we throw a dice, we do not know the outcome in advance, but we *can* describe the set (*sample space*) of all possible outcomes (*events*), including their *probabilities*. This and other terminology is summarised below:

1. *Sample space*: the set, here denoted by Ω, that contains all possible outcomes of an experiment.
2. *Event*: a 'nice' subset of the sample space.[1]
3. *Probability measure* P: a real valued function defined on the set of events that satisfies
   (a) $0 \leq P(A) \leq 1$, for every event $A \subseteq \Omega$.
   (b) $P(\Omega) = 1$.
   (c) $P(A_1 \cup A_2 \cup \cdots) = P(A_1) + P(A_2) + \cdots$ for every finite or infinite sequence of disjoint events $A_1, A_2, \ldots$.

   $P(A)$ is the probability on event $A \subset \Omega$.
4. *Complementary event*: if $A \subseteq \Omega$ is an event then $A^c \equiv \Omega \setminus A$ is the complementary event.
5. *Addition rule*: $P(A_1 \cup A_2) = P(A_1) + P(A_2) - P(A_1 \cap A_2)$. (Use 3(c).)
   In particular, $P(A_1 \cup A_2) = P(A_1) + P(A_2)$ if A_1 and A_2 are mutually exclusive events (i.e., A_1 and A_2 are disjoint). Moreover, $P(A \cup A^c) = 1$ (use 3(b)) and $P(A^c) = 1 - P(A)$.
6. *Conditional rule*: the probability of A_1 given A_2, denoted by $P(A_1|A_2)$, is defined as $P(A_1|A_2) \equiv \frac{P(A_1 \cap A_2)}{P(A_2)}$.
   Note that for a given subset A_2, $P(\,\cdot\,|A_2)$ defines a probability measure (on Ω, but also on A_2).

[1]If Ω is topological space, then open subsets plus the subsets that can be constructed by countably many repetitions of countably many unions, intersections, and complements, from open subsets, are so-called *Borel* subsets. They are 'nice'.

7. *Multiplication rule*: $P(A_1 \cap A_2) = P(A_2)P(A_1|A_2)$.
8. *Independent events*: two events A_1 and A_2 are independent if $P(A_1 \cap A_2) = P(A_1)P(A_2)$.
   In this case $P(A_1|A_2) = P(A_1)$.

The sample space Ω, the sets of events, and the probability measure P together form a *probability space*. Elements of Ω are called *samples*. In experiments, samples will randomly be drawn from the sample space Ω according to the probability measure P, i.e., the probability that a sample ω belongs to an event A is equal to $P(A)$: if $\omega_1, \omega_2, \ldots$ is a sequence of samples randomly selected from Ω then

$$\frac{1}{n}\#\{i \leq n \mid \omega_i \in A\} \to P(A) \qquad (n \to \infty).$$

Here $\#B$ denotes the *cardinality* of a subset B, i.e., the number of elements of B.

Example 1.1
If the experiment is tossing a dice once, then $\Omega = \{1, 2, 3, 4, 5, 6\}$. The probability of throwing the value $\omega \in \Omega$ equals $\frac{1}{6}$ (if the dice is 'fair'). Therefore (by 3(c)), $P(A) = \frac{\#A}{6}$. In particular, $\#A = 1$ if A consists of one number $i \in \Omega$ only: $A = \{i\}$. ♦

Example 1.2
If the experiment is throwing a coin 8 times in a row, then Ω consists of sequence as 10110100, where 0 represents a head in the first, second, fourth and seventh throw and 1 a tail in the others: $\Omega = \{0, 1\}^8 \equiv \{d = (d_1, d_2, \ldots, d_8) \mid d_j \in \{0, 1\}\}$. The probability of throwing heads only, i.e., $A = \{00000000\}$, is 2^{-8}. More generally, if $A \subset \Omega$, then $P(A) = 2^{-8} \times \#A$. In particular, the probability of throwing exactly k heads and $8 - k$ tails for some $k \in \{0, 1, \ldots, 8\}$, i.e., $A = \{d \in \Omega \mid \#\{j \mid d_j = 0\} = k\}$, equals $\frac{8!}{k!(8-k)!2^8}$.

Note that the sequence $d = (d_1, d_2, \ldots, d_8) \in \{0, 1\}^8$ can be identified with a rational in $[0, 1]$ represented in the binary number system as $0.d_1d_2 \ldots d_8 \equiv \sum_{j=1}^{8} d_j 2^{-j}$. Actually, the space Ω in this course from which we intend to randomly sample is (a variant of) $\{0.d_1d_2 \ldots d_{32} \mid d_i \in \{0, 1\}\}$ (or, equivalently, $\{0, 1\}^{32}$). All rationals in this space have equal probability to be sampled: $P(\omega) = 2^{-32}$; see also Ex. 1.3. ♦

The probability spaces in the above examples are discrete: Ω is finite and P 'counts'. A probability measure P is also said to be discrete if P 'counts' in a weighted manner (on a countably subset Ω_c of Ω taking the value 0 outside this set: $P(A) = 0$ of A is disjoint from Ω_c). Next we present three continuous examples. The third one, Ex. 1.5, is rather elaborate, but is an indication that 'Monte Carlo' methods can solve 'complicated' problems (as computing the number π to some accuracy).

Example 1.3
If the experiment is selecting randomly a real number in $[0, 1]$ then $\Omega = [0, 1]$ and $P(A)$ measures the 'length' of (measurable) subsets A of $[0, 1]$: $P([a, b]) = b - a$ if $0 \leq a \leq b \leq 1$.

In this course, we will view the sample space $\{0.d_1 \ldots d_{32} \mid d_j \in \{0, 1\}\}$ of binary rational numbers in $[0, 1]$ with associated probability measure as discussed at the end of Ex. 1.2 as a discretization of $[0, 1]$. For ease of discussions, however, we will usually assume in our theoretical expositions that $\Omega = [0, 1]$ and that P measures the length of intervals in $[0, 1]$, rather than that Ω consists only of binary rationals in $[0, 1]$ of restricted length. ♦

Example 1.4
Let Ω be the product $[0, 1] \times [0, 1]$ of two intervals endowed with the measure P that is determined by $P([a, b] \times [c, d]) = (b - a)(d - c)$ for sub intervals $[a, b]$ and $[c, d]$ of $[0, 1]$: P measures the area of subsets of Ω. This probability space is the *product* of two copies of the probability space of Ex. 1.3.
Note that the event $A_1 \equiv [a, b] \times [0, 1]$ is independent of the event $A_2 \equiv [0, 1] \times [c, d]$, because, $P(A_1) = b - a$, $P(A_2) = d - c$, and $A_1 \cap A_2 = [a, b] \times [c, d]$.

If an experiment is repeated with randomly sampling from, say $[0,1]$, independent of the sampling used in the first experiment (also randomly sampled from $[0,1]$), then this setting can be represented with the product space $[0,1] \times [0,1]$, where, with (ω_1, ω_2), the sampling for the first experiment is represented in the first coordinate ω_1 and the sampling for the second experiment is represented in the second coordinate ω_2. To describe N repetitions of the same experiment with independent sampling, the space of the product of N copies of the probability space for one experiment can be used. ♦

Example 1.5
Consider a huge sheet of paper on the floor with parallel lines with neighbouring lines at distance 1 cm from each other. The experiment is randomly tossing a needle of one cm flat on the paper. We are interested in probability of (the event of) 'hitting' one of the lines with the needle.
The position of the the needle after throwing can be described as follows. Endow the sheet of paper with Cartesian coordinates with x-axis on one of the parallel lines and the other lines with y-value in $\mathbb{Z}$. Then the position of the needle on the paper can be described by the position of its tip, say (x, y), and the angle, say $\phi \in [0, 2\pi)$, with the x-axis. We are interested in hitting a line. Therefore, the x-value is irrelevant. We assume the tip to be on the y-axis: (y, ϕ) with $y \in \mathbb{R}, \phi \in [0, 2\pi)$ describes (the position of) the needle. Moreover, the needle (y, ϕ) hits a line if and only the needle $(y \bmod 1, \phi)$ hits a line. Therefore, for the sample space Ω we take the collection (y, ϕ) with $y \in [0, 1)$ and $\phi \in [0, 2\pi)$. The needle (y, ϕ) hits the line $y = 0$ if and only if $y + \sin(\phi) < 0$ and hits the line $y = 1$ if and only if $y + \sin(\phi) > 1$. We are interested in the event $A_{\text{hit}} \equiv \{(y, \phi) \in \Omega \mid y + \sin(\phi) < 0 \text{ or } y + \sin(\phi) > 1\}$. Since all needle positions are equally likely, the probability $P(A)$ of finding a needle in a subset A of Ω equals the size of the area of A relative to the size of Ω. The size of Ω is 2π. Note that $y + \sin(\phi) < 0$ implies that $\phi \in [\pi, 2\pi)$, while for $y + \sin(\phi) > 1$ we have $\phi \in [0, \pi)$. Therefore, we can split A_{hit} in two disjoint parts. The second part has area

$$\int_0^\pi \int_{1-\sin\phi}^1 1 \, dy \, d\phi = \int_0^\pi \sin\phi \, d\phi = -\cos\phi \,|_0^\pi = 2$$

Similarly, the first part has also area 2. Since the size of Ω is 2π, we see that $P(A_{\text{hit}}) = \frac{4}{2\pi} = \frac{2}{\pi}$.
If we throw the needle N times and $h(N)$ is the number of times one of the lines has been hit by the needle then $\lim_{N\to\infty} \frac{h(N)}{N} = \frac{2}{\pi}$. ♦

In practise the sample space may be described in a less mathematical way. For instance, Ω may be the collection of all first year students at Utrecht University (UU) this year. To get some insight in the average length of these students, or the average grades for Mathematics that they had on High School, a subset of this set of students may be examined. The probability P that a specific student is selected is then equal to $1/N$, where N is the number all students in Ω. The subset Ω_m of all males is an event and $P(\cdot \mid \Omega_m)$ is the probability measure that restricts to the male first years students.

1.2 Random variables

We consider a probability space with sample space Ω and probability measure P.

A *random variable* X is a real valued function defined on Ω. The subset $\{X(\omega) \mid \omega \in \Omega\}$ of the real numbers is the *observation space*. We denote random variables by uppercase letters from the end of the alphabet, e.g., X, Y, Z. We use notations as $\{X \leq a\}$ to indicate the event $\{\omega \in \Omega \mid X(\omega) \leq a\}$. With $P(X \leq a)$ we mean $P(\{X \leq a\})$.

The variable is said to be 'random', since the idea is that the *realisations* $X(\omega)$ are 'observed' by randomly 'sampling' ωs from the sample space Ω, where the sampling is random according to the probability P. A sequence $(x_0, x_1, \ldots)$ of real numbers is a sequence of realisations of the random variable X if $x_i = X(\omega_i)$ for a sequence $(\omega_0, \omega_1, \ldots)$ of ω_i in Ω that are randomly selected according to probability P.

If Ω is the set of first years students at UU, then the length $X(\omega)$ of student ω defines a random variable X (if the idea is to access, say the average length of this student population, by averaging over a subset of Ω

consisting of random samples).

1.2.1 Discrete random variables

A *discrete random variable* is a random variable that can take a finite or at most a countably infinite number of values: $X(\Omega)$, the observation space is finite or countable.

Let X be a discrete random variable. Then

1. the *frequency function* of X (also called *discrete probability density function* or *probability mass function*) is the function f defined on $\mathbb{R}$ by
$$f(x) \equiv P(X = x).$$

2. The *cumulative distribution function (cdf)* of X is defined by
$$F(x) \equiv P(X \leq x) = \sum_{t \leq x,\, t \in X(\Omega)} f(t) = \sum_{t \leq x} f(t).$$

Note that f takes values in $[0,1]$, $\sum_{x \in X(\Omega)} f(x) = 1$, and $f(x) = 0$ if $x \notin X(\Omega)$. The cdf satisfies
$$\lim_{x \to -\infty} F(x) = 0 \quad \text{and} \quad \lim_{x \to \infty} F(x) = 1.$$

Example 1.6 (continues Ex. 1.1)
$\Omega \equiv \{1, 2, \ldots, 6\},\ P(\omega) \equiv \frac{1}{6}\ \ (\omega \in \Omega)$.
• $X(\omega) \equiv \omega$ for all $\omega \in \Omega$, i.e., X is the outcome of a toss of a fair dice. X has frequency function $f(x) = \frac{1}{6}$ if $x = j$ $(j = 1, 2, \ldots, 6)$ and $f(x) = 0$ for other $x \in \mathbb{R}$.
• $Y(\omega) \equiv 1$ if ω is odd $(\omega \in \Omega)$ and $Y(\omega) \equiv 0$ if ω is even with frequency function $f(x) = \frac{1}{2}$ for $x \in \{0, 1\}$ and $f(x) = 0$ for other $x \in \mathbb{R}$. ♦

Example 1.7 (continues Ex.1.2)
$\Omega \equiv \{d = (d_1, \ldots, d_8) \mid d_j \in \{0, 1\}\},\ P(d) \equiv 2^{-8}\ \ (d \in \Omega)$.
• $X(d) \equiv \#\{j \mid d_j = 1\}$ for $d \in \Omega$.
• $Y(d) \equiv d_1 2^{-1} + d_2 2^{-2} + \ldots + d_8 2^{-8}$: the sequence $(d_1, d_2, \ldots, d_8) \in \{0, 1\}^8$ is viewed as a rational in $[0, 1]$ represented in the binary number system as $0.d_1 d_2 \ldots d_8$. ♦

1.2.2 Continuous random variables

A *continuous random variable* X is a random variable that can take a continuum of values that are distributed according to a *(probability) density function* (pdf), [2] i.e., a real-valued function f on $\mathbb{R}$ such that

1. $f(x) \geq 0, \qquad (x \in \mathbb{R}),$
2. $\displaystyle\int_{-\infty}^{\infty} f(x)\, dx = 1,$ and
3. $\displaystyle\int_a^b f(x)\, dx = P(a \leq X \leq b) \qquad (a, b \in \mathbb{R},\ a < b).$

[2]In addition, the sets $\{\omega \in \Omega \mid X(\omega) \leq x\}$ should be Borel for all $x \in \mathbb{R}$. Note that a continuous random variable X is not required to be continuous as function from Ω to $\mathbb{R}$.

f is also called the *distribution function* of X.
Note that $P(X = a) = 0$ if X is a continuous random variable, because then

$$P(X = a) = \int_a^a f(x)\,dx = 0.$$

Associated to f is the *cumulative distribution function (cdf)* F of X defined by

$$F(x) = P(X \le x) = \int_{-\infty}^x f(y)\,dy.$$

Note that the cdf of a continuous random variable is continuous,[3] while the cdf of a discrete random variable is a step function. The cdf satisfies

$$\lim_{x\to-\infty} F(x) = 0 \quad \text{and} \quad \lim_{x\to\infty} F(x) = 1.$$

If f is continuous at x then

$$f(x) = F'(x).$$

The cdf can be used to compute probabilities

$$P(a \le X \le b) = F(b) - F(a).$$

If $(\omega_1, \omega_2, \ldots)$ is a sequence in Ω randomly selected according to P, and $x_j \equiv X(\omega_j)$, then

$$\frac{1}{n}\{j \le n \mid x_j \in [a,b]\} \to \int_a^b f(x)\,dx = F(b) - F(a) \qquad (n \to \infty).$$

Both functions f and F inform us on X without explicit reference to Ω or P.

Example 1.8 (continues Ex.1.3)
$\Omega \equiv [0,1],\ \ P([a,b]) \equiv b - a \ \ (a, b \in [0,1], a \le b)$.
• Consider the random variable $X : [0,1] \to [\mu, M]$ given by $X(\omega) = (M-\mu)\omega + \mu \ \ (\omega \in \Omega \equiv [0,1])$.
The pdf f and the cdf F of X are

$$f(x) = \begin{cases} 0, & \text{if } x < \mu, \\ \frac{1}{M-\mu}, & \text{if } \mu \le x \le M, \\ 0, & \text{if } x > M, \end{cases} \qquad F(x) = \begin{cases} 0, & \text{if } x < \mu, \\ \frac{x-\mu}{M-\mu}, & \text{if } \mu \le x \le M, \\ 1, & \text{if } x > M. \end{cases} \tag{1.1}$$

This random variable has a '*uniform*' distribution function: all numbers in $[\mu, M]$ are equally likely to occur as realisations $X(\omega)$ of X. Note that the value of the functions f at a single point, as $x = \mu$ or $x = M$, is of no importance, since, in this case, it does not affect the value of $\int_a^b f(x)\,dx$.
A random variable with uniform distribution in $[\mu, M]$ will be denoted by $U_{[\mu,M]}$.
• Consider the random variable $X : [0,1] \to [0,1]$ defined by $X(\omega) = 2\omega$ for $\omega \in [0, \frac{1}{4}]$, $X(\omega) = \frac{1}{2}$ for $\omega \in [\frac{1}{4}, \frac{3}{4}]$ and $X(\omega) = 2\omega - 1$ for $\omega \in [\frac{3}{4}, 1]$. Note that X is a continuous function. However, since $P(X = \frac{1}{2}) = P([\frac{1}{4}, \frac{3}{4}]) = \frac{1}{2}$, X is not a continuous random variable variable. Neither is it a discrete variable, since X takes all values in the continuum $[0,1]$: X mixes discrete and continuous. The pdf f can be discribed as the sum of f_0 and $\frac{1}{2}\delta_{\frac{1}{2}}$, where $f_0(x) = 0$ for $x \notin [0,1]$ and $f_0(x) = \frac{1}{2}$ for $x \in (0,1)$, while δ_z is the so-called Dirac delta function at z.[4] $F(x) = 0$ for $x < 0$, $F(x) = \frac{1}{2}x$ for $x \in [0, \frac{1}{2})$, $F(x) = \frac{1}{2}(x+1)$ for $x \in [\frac{1}{2}, 1]$, $F(x) = 1$ for $x \ge 1$. Note that F is discontinous.

[3] Actually, the continuity of F is slightly stronger than required by the standard continuity definition: F is *absolutely continuous*.
[4] The Dirac delta function δ_z is frequently used by Physicists. It is is not a function in the classical sense. It takes the value 0 outside $\{z\}$: $\delta_z(x) = 0$ if $x \ne z$. But it is ∞ at z with a certain weight: $\int_{-\infty}^{\infty} g(x)\delta_x(x)\,dx = g(z)$ for all continuous functions g. For mathematicians, δ_z defines a discrete point measure at z.

• Consider the random variable Y on $[0,1]$ given by $Y(\omega) \equiv \tan \pi(\omega - \frac{1}{2})$ $\quad (x \in [0,1])$. The pdf f and cdf F of Y are given by

$$f(x) = \frac{1}{\pi(1+x^2)} \quad \text{and} \quad F(x) = \frac{1}{2} + \frac{1}{\pi}\arctan(x) \qquad (x \in \mathbb{R}).$$

This pdf is the so-called *Cauchy distribution*.
• Another important pdf is the *standard normal distribution* given by

$$f(x) \equiv f_{\text{st}}(x) \equiv \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \qquad (x \in \mathbb{R}).$$

There is no expression for the associated cdf in terms of familiar functions as logs, exponentials, polynomials,.... The same remark holds for the random variable Z on $[0,1]$ for which f_{st} is the distribution function (see also §2.4.2).
Shifting by $\mu \in \mathbb{R}$ and scaling by $\sigma \in \mathbb{R}, \sigma \neq 0$, of the standard normal distribution leads to more general *normal distributions*:

$$f(x) \equiv \frac{1}{\sigma} f_{\text{st}}\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \qquad (x \in \mathbb{R}).$$

The scaling by σ is required in order to have $\int f(x)\, dx = 1$. ♦

Exercise 1.1: *Squaring the normal distribution.*
Let X be a random variable with standard normal distribution. Show that the distribution of X^2 is given by

$$f(x) = \frac{1}{\sqrt{2\pi x}} \exp\left(-\frac{x}{2}\right) \quad \text{for} \quad x \in (0,\infty) \quad \text{and} \quad f(x) = 0 \quad \text{for} \quad x \in (-\infty, 0]. \qquad ■$$

Example 1.9 (Bernoulli random variables)
A so-called *Bernoulli random variable* takes, for some $p \in [0,1]$, the values 1 and 0 with probability p and $1-p$, respectively. Its frequency function is

$$f(1) = p,\ f(0) = 1-p, \text{ and } f(x) = 0 \text{ for } x \notin \{0,1\}.$$

An example of a Bernoulli random variable B can be defined on the sample space $[0,1]$ (from Ex. 1.3) by $B(\omega) \equiv 1$ if $\omega \in [0,p]$ and $B(\omega) \equiv 0$ if $\omega \in [p,1]$. Note that this random variable is discrete while the sample space is a 'continuum'.
Actually, any *indicator random variable* χ_A of an event A in a sample space Ω is a Bernoulli random variable for some $p \in [0,1]$. Here,

$$\chi_A(\omega) \equiv \begin{cases} 1, & \text{if } \omega \in A, \\ 0, & \text{otherwise.} \end{cases} \tag{1.2}$$

The value p is given by $p = P(A)$, where P is the probability measure on Ω.
Note that a Bernoulli random variable can also be realised on a discrete probability space (with a 'non-uniform' probability): for instance, $\Omega = \{0,1\}$, $P(\{0\}) = 1-p$, $P(\{1\}) = p$, and random variable X with $X(0) = 0$ and $X(1) = 1$. ♦

1.2.3 Independent random variables

Two random variable X and Y (on the same probability space) are *independent* if for every a and b in $\mathbb{R}$ the sets $\{\omega \in \Omega \mid X(\omega) \leq a\}$ and $\{\omega \in \Omega \mid Y(\omega) \leq b\}$ are independent events.

Example 1.10 (continues Ex.1.4)
Consider the product sample space $\Omega \equiv [0,1] \times [0,1]$ with probability measure P fixed by requiring $P([a,b] \times [c,d]) \equiv (b-a)(d-c)$ for sub intervals $[a,b]$ and $[c,d]$ of $[0,1]$.

Now, consider a non-trivial random variable Z on $[0,1]$. This random variable can be 'lifted' to a random variable X on Ω and to a random variable Y by

$$X(\omega_1,\omega_2) \equiv Z(\omega_1) \quad \text{and} \quad Y(\omega_1,\omega_2) \equiv Z(\omega_2) \qquad ((\omega_1,\omega_2) \in \Omega).$$

One easily checks that the events $\{(\omega_1,\omega_2) \mid Z(\omega_1) \leq a\}$ and $\{(\omega_1,\omega_2) \mid Z(\omega_2) \leq b\}$ are independent w.r.t. P. Hence, X and Y are independent random variables. Nevertheless, X and Y have identical distribution functions (Why?).
The situation where random variables are independent but have identical distributions is denoted by *i.i.d.* in text books.
This example basically gives a formal description of the fact that that if we randomly sample twice from $[0,1]$, say with samples ω_1 and ω_2 in $[0,1]$, then the realisation $Z(\omega_2)$ from the second sample is independent from the realisation $Z(\omega_1)$ from the first sample. ♦

Exercise 1.2: *Independent random variables.*
Consider the situation of the above example (Ex. 1.10). Check that X and Y are independent random variable with identical distribution functions (both equal to the distribution function of Z). Check that $X(\omega_1,\omega_2) \equiv Z(\omega_1)$ and Y, now with Y defined by $Y(\omega_1,\omega_2) \equiv Z(\omega_1) + Z(\omega_2)$, are not independent random variables on Ω. If X_1 and Y_1 are random variables on $[0,1]$, then, check that, the 'lifted' versions X and Y, $X(\omega_1,\omega_2) \equiv X_1(\omega_1)$ and $Y(\omega_1,\omega_2) \equiv Y_1(\omega_2)$, are independent variables on the product space $[0,1] \times [0,1]$. ■

As we learnt form the above example, independent variables can easily be defined on product spaces. More complicated independent variables exist as well, for instance, arising by rotating (i.e., by applying a unitary transformation) product spaces.

Theorem 1.1
If X and Y are continuous independent random variables with distribution function f_X and f_Y, respectively, then $X+Y$ is a random variable with distribution function f_{X+Y} given by the *convolution product* $f_X * f_Y$ of f_X and f_Y:

$$f_{X+Y}(x) = f_X * f_Y(x) \equiv \int_{-\infty}^{\infty} f_X(x-y) f_Y(y)\, dy \qquad (x \in \mathbb{R}).$$

A similar expression holds for discrete random variables and their frequency functions: replace the integral by a sum. ♦

Exercise 1.3: *Sums of independent variables.*
Consider the situation of Ex. 1.10. With $X(\omega_1,\omega_2) \equiv Z(\omega_1)$ and $Y(\omega_1,\omega_2) \equiv Z(\omega_2)$, show that the distribution function of $(\omega_1,\omega_2) \rightsquigarrow Z(\omega_1) + Z(\omega_2)$ equals $f_Z * f_Z$. ■

Exercise 1.4: *Square sums of independent variables with normal distribution.*
Let X and Y be independent random variables both with distribution function the same standard normal distribution. Use the results of Exer. 1.1 to show that the distribution of $X_1^2 + X_2^2$ equals

$$f(x) = \frac{1}{2}\exp\left(-\frac{x}{2}\right) \quad \text{for} \quad x \in (0,\infty) \quad \text{and} \quad f(x) = 0 \quad \text{for} \quad x \in (-\infty, 0]. \qquad ■$$

Exercise 1.5: *Sums of discrete independent variables.*
Consider polynomials $p(x) = \alpha_0 + \alpha_1 x + \ldots + \alpha_k x^k$ and $q(x) = \beta_0 + \beta_1 x + \ldots + \beta_m x^m$ $(x \in \mathbb{R})$. Suppose f_p and f_q are the frequency functions of independent discrete random variables X_p and X_q, respectively, such that $f_p(x) = \alpha_j$ if $x = j$ $(j = 0,\ldots,k)$ and $f_p(x) = 0$ elsewhere. Similarly, $f_q(j) = \beta_j$ $(j = 0,\ldots,m)$ and $f_q(x) = 0$ for other values of x. Prove that the j-th coefficient of the product polynomial pq equals $f_p * f_q(j)$ $(j = 0,\ldots,km)$, while $f_p * f_q(x) = 0$ if $x \notin \{0,\ldots,km\}$. ■

Example 1.11 (continues Ex. 1.10)
Example 1.10 can easily be extended to create an infinite sequence of independent random variables X_1,

$X_2, \ldots$ with identical distribution: form the product sample space $\Omega \equiv [0,1]^\infty$ endowed with the product measure P, i.e.,

$$P([a_1,b_1] \times \ldots \times [a_n,b_n] \times [0,1]^\infty) \equiv \prod_{j=1}^{n} (b_j - a_j) \quad ([a_j,b_j] \subset [0,1],\ n \in \mathbb{N})$$

Define X_i by

$$X_i(\omega_1, \omega_2, \ldots) \equiv Z(\omega_i).$$

Then, the X_i are independent random variables all with distribution function equal to the one of Z. Let $\overline{X}_n$ the random variable (on Ω) that arises by averaging the first n random variable X_i:

$$\overline{X}_n \equiv \frac{1}{n}\sum_{i=1}^{n} X_i. \quad \text{Then} \quad \overline{X}_n(\omega_1, \omega_2, \ldots) = \frac{1}{n}\sum_{i=1}^{n} Z(\omega_i).$$

Note that there is a one to one correspondence between randomly selected elements $\omega \equiv (\omega_1, \omega_2, \ldots)$ in Ω (random selection according to the product probability P) and sequences $(\omega_1, \omega_2, \ldots)$ of numbers randomly sampled from $[0,1]$ (with 'uniform' probability). If $\omega \equiv (\omega_1, \omega_2, \ldots)$ is a randomly selected element in Ω, then $\overline{X}_n(\omega)$ is a realisation of $\overline{X}_n$ and this realisation is the average of n realisations $z_1, z_2, \ldots, z_n$ of Z, where $z_i \equiv Z(\omega_i)$:

$$\overline{X}_n(\omega) = \frac{1}{n}\sum_{i=1}^{n} z_i.$$

This average is also called the *sample mean*. Note that the notation 'ω_i' now has been used in a second meaning: first we used it to denote the i-th coordinate of the variable $(\omega_1, \omega_2, \ldots)$ in Ω and now we use it as the i-th specific choice of a random sample from $[0,1]$. ♦

1.2.4 Applications

Random variables with a Bernoulli distribution, uniform distribution, or a normal distribution are important in practise. In tests to check whether a sequence of numbers can be accepted as realisation of such variables, random variables with derived distributions, as binomial and chi-square, show up.

Chi-square distributions. Let $X_1, \ldots, X_\nu$ be an i.i.d. standard normal sequence, i.e., a sequence of independent random variables with identical distribution all equal to the standard normal distribution. Then, the square sum

$$\sum_{j=1}^{\nu} X_j^2 \tag{1.3}$$

of these random variable is a random variable distributed according the so-called *chi-square distribution* function χ^2_ν (for ν *degrees of freedom*):

$$\chi^2_\nu(x) = \frac{1}{\kappa_\nu} x^{(\nu-2)/2} \exp\left(-\frac{x}{2}\right) \quad (x \in (0,\infty)) \quad \text{and} \quad \chi^2_\nu(x) = 0 \quad (x \in (-\infty, 0]). \tag{1.4}$$

Before we specify the constant κ_ν, note that this result generalises the ones in Exer. 1.1 and Exer. 1.4. The constant κ_ν is determined by the fact that $\int_{-\infty}^{\infty} \chi^2_\nu(x)\,dx = 1$. Hence,

$$\kappa_\nu = 2^{\nu/2}\,\Gamma(\nu/2), \quad \text{where} \quad \Gamma(\mu) \equiv \int_0^\infty t^{\mu-1} e^{-t}\,dt \quad (\mu > -1). \tag{1.5}$$

Γ is the *gamma function*. The value of the gamma function reduces to a factorial for positive integers μ: $\Gamma(m+1) = m! \quad (m \in \mathbb{N})$.

Binomial distributions. This application continuous Ex. 1.9. Let $B_1, B_2, \ldots, B_n$ be an i.i.d. Bernoulli sequence: $B_i = 1$ with probability p and $B_i = 0$ with probability $1-p$ (i.e., the B_i are independent and

they all have the same distribution function f with $f(1) = p$ and $f(0) = 1 - p$). Let $Y \equiv \sum_{i=1}^{n} B_i$.
Interpretation: each Bernoulli random variable B_i may arise in the description of an experiment that has a probability p of failure and probability $1 - p$ of success (see Ex. 1.9). This experiment is repeated n times in such a way that the result of one trial does not depend on the results of the other trails. Then, Y is the random variable that counts the number of successes in these n trials.
The distribution f_n of Y is *binomial*:

$$P(Y = k) = f_n(k) \equiv \binom{n}{k} p^k (1-p)^{n-k}. \tag{1.6}$$

Exercise 1.6: *Binomial distribution.*
Use the results of Exer. 1.5 to show that (1.6) is correct. ■

Theorem 1.2 (Moivre–Laplace)
The binomial distribution resembles a normal distribution at a discrete set of points for large n:

$$\binom{n}{k} p^k (1-p)^{n-k} \approx \frac{1}{\sqrt{2\pi np(1-p)}} \exp\left(-\frac{(k-np)^2}{2np(1-p)}\right) \quad (k = 0, 1, \ldots, n). \tag{1.7}$$

♦

The approximation is better for larger n.'Large' and 'better' depends on p: the approximation appears to be already quite accurate if both $np \geq 1$ and $n(1-p) \geq 1$.

Exercise 1.7: *Binomial distributions as approximate normal distributions.*
Check that the right-hand side expression in (1.7) equals

$$\frac{1}{\sigma} f_{\text{st}}\left(\frac{k-\mu}{\sigma}\right), \quad \text{where} \quad f_{\text{st}}(x) \equiv \frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2}), \quad \mu \equiv np, \quad \sigma \equiv \sqrt{np(1-p)}$$

■

1.3 Mean and Variance

The *expectation* or *mean* of a discrete random variable X is defined as

$$E(X) \equiv \sum_{i=0}^{n} x_i \, P(X = x_i) = \sum_{x \in X(\Omega)} x \, P(X = x) = \sum_{x \in \mathbb{R}} x \, f(x), \tag{1.8}$$

where the x_i range over all possible values in the observation space $X(\Omega)$. Note that possibly $n = \infty$. The mean is not defined for random variables, or, equivalently, distribution functions f, for which the above sum does not exist.

As an example of a mean, when X is the outcome of a toss of a fair dice then $E(X) = \frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) = 3.5$, which nicely illustrates that the expectation is not necessarily a realisable event. Do not confuse the mean with the *average* $\bar{X}_n \equiv \frac{1}{n} \sum_{i=0}^{n} X_i$ of n realisations $X_i = X(\omega_i)$ of a random variable X. There are however theorems that relate the average to the mean (if the ω_i are randomly sampled), as we will see in §1.4.

For continuous random variables, distributed according to the probability distribution function f, we define the *mean* or *expectation* as

$$E(X) \equiv \int_{-\infty}^{\infty} x f(x) \, dx, \tag{1.9}$$

if the integral exists.

We now collect several facts on expectations in one theorem.
If X is a random variable on Ω and g is a real-valued function on $\mathbb{R}$ then the composition $Y \equiv g(X) \equiv$

$g \circ X$, i.e., $Y(\omega) = g(X(\omega))$ $(\omega \in \Omega)$, also defines a random variable on Ω. Unfortunately, it is not easy to relate the probability density function of $g(X)$ to (g and) the probability density function f of X (unless g is monotonic, as, $g \in C^1(\mathbb{R})$ and $g'(t) > 0$ all $t \in \mathbb{R}$). The situation is more favourable with respect to the mean:

Theorem 1.3
• Let X be a random variable with probability density function f. Let g be a real-valued function on $\mathbb{R}$.[5] If Y is the random variable $Y \equiv g(X)$, then

$$E(g(X)) = E(Y) = \sum_x g(x)f(x) \quad \text{or} \quad E(g(X)) = E(Y) = \int g(x)f(x)\,dx,$$

in the case X is discrete and continuous, respectively, .
• For independent random variables X and Y, we have that

$$E(XY) = E(X)E(Y).$$

• If X and Y are random variables, α, β, and γ are real numbers, then,

$$E(\alpha + \beta X + \gamma Y) = \alpha + \beta E(X) + \gamma E(Y) :$$

the expectation is linear (even if X and Y are not independent!). ♦

Of course, the above statements only hold if the sums and integrations that are involved exist (converge).

☞ **Exercise 1.8:** *Mean.*
Consider a continuous random variable X and an $\alpha \in \mathbb{R}$. Show that

$$E(X^2) = \int x^2\, f(x)\,dx, \quad E(|X|) = \int |x|\, f(x)\,dx, \quad E(X - \alpha) = \int (x - \alpha)\, f(x)\,dx. \qquad \blacksquare$$

We now turn to the *variance* $\mathrm{Var}(X)$ and the *standard deviation* $\sigma(X)$ of the random variable X:

$$\mathrm{Var}(X) \equiv E([X - E(X)]^2) \quad \text{and} \quad \sigma(X) \equiv \sqrt{\mathrm{Var}(X)}. \tag{1.10}$$

These quantities measure the deviation from the average $E(X)$. The variance is *not* linear. Actually,

$$\mathrm{Var}(\alpha + \beta X) = \beta^2\, \mathrm{Var}(X) \qquad (\alpha, \beta \in \mathbb{R}), \tag{1.11}$$

under the assumption that $\mathrm{Var}(X)$ exists. The proof of the following theorem is an exercise,

Theorem 1.4
The variance exists if $E(X^2) < \infty$ and can then also be written as

$$\mathrm{Var}(X) = E(X^2) - [E(X)]^2. \qquad \blacklozenge$$

☞ **Exercise 1.9:** *Mean and Variance.*
Consider a random variable X.
a) Use Cauchy–Schwartz to show that, for any $\mu \in \mathbb{R}$, $E(|X - \mu|) \leq \sqrt{E(|X - \mu|^2)}$. In particular,

$$E(|X - E(X)|) \leq \sqrt{E([X - E(X)]^2)} = \sigma(X). \tag{1.12}$$

b) Use Th. 1.4 to show that the variance is non-negative if $E(|X|^2)$ exists.
c) Let Y be a random variable, independent of X. Prove that

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y).$$

[5]In addition, g should be such that $\{x \in \mathbb{R} \mid g(x) \in I\}$ is a Borel subset for all intervals I of $\mathbb{R}$.

d) The *k-th moment* $m_k(X)$ of random variable X is defined as

$$m_k(X) \equiv E(|X|^k).$$

Assume the sample space Ω is finite and $P(\{\omega\}) > 0 \quad (\omega \in \Omega)$. Show that, $\sqrt[k]{m_k(X)}$, the k-th moment to the power $1/k$, can be viewed as a weighted k-norm on the vector $\mathbf{x}$ with coordinates the $X(\omega)$: if $(\mu_1, \ldots, \mu_n)$ are *weights*, i.e., $\mu_i > 0$, and $\mathbf{x} = (x_1, \ldots, x_n)^T$ is a vector then $\|\mathbf{x}\|_k \equiv \sqrt[k]{\sum_i \mu_i |x_i|^k}$ defines a *weighted k-norm.* What are the weights here? ■

Example 1.12 (continues Ex. 1.11)
As in Example 1.11, for a random variable Z on $[0, 1]$, let X_i be the random variable on $\Omega \equiv [0, 1]^\infty$ defined by $X_i(\omega_1, \omega_2, \ldots) \equiv Z(\omega_i)$. Let $\overline{X}_n$ the random variable that arises by averaging the first n X_i:

$$\overline{X}_n \equiv \frac{1}{n} \sum_{i=1}^{n} X_i.$$

Then

$$E(\overline{X}_n) = \frac{1}{n} \sum_{i=1}^{n} E(X_i) = \frac{1}{n} \sum_{i=1}^{n} E(Z) = E(Z) \tag{1.13}$$

and

$$\mathrm{Var}(\overline{X}_n) = \frac{1}{n^2} \mathrm{Var}(\sum_{i=1}^{n} X_i)) = \frac{1}{n^2} \sum_{i=1}^{n} \mathrm{Var}(X_i) = \frac{1}{n} \mathrm{Var}(Z). \tag{1.14}$$

This implies that

$$E(|\overline{X_n} - E(Z)|) = E(|\overline{X}_n - E(\overline{X}_n)|) \leq \sqrt{\mathrm{Var}\,(\overline{X}_n)} = \frac{1}{\sqrt{n}} \sqrt{\mathrm{Var}(Z)}. \tag{1.15}$$

Since $\mathrm{Var}(Z) = \frac{1}{n} \sum_{i=1}^{n} \mathrm{Var}(X_i)$, we also have that $\mathrm{Var}(Z) = E(\frac{1}{n} \sum_{i=1}^{n} [X_i - E(Z)]^2)$. The equality $E(\overline{X}_n) = E(Z)$ suggests that replacing $E(Z)$ by $\overline{X}_n$ also leads to an expression for $\mathrm{Var}(Z)$, which is true except for a small factor:

$$E\left(\frac{1}{n} \sum_{i=1}^{n} [X_i - \overline{X}_n]^2\right) = \frac{n-1}{n} \mathrm{Var}(Z). \tag{1.16}$$

♦

Exercise 1.10: *Special distributions.*
Consider a random variable X.
a) Assume X has standard normal distribution $f_{\rm st}$; see Ex. 1.8. Show that $E(X) = 0$ and $\mathrm{Var}(X) = 1$.
b) Assume X has normal distribution $f(x) \equiv \frac{1}{\sigma} f_{\rm st}((x - \mu)/\sigma)$. Compute $E(X)$ and $\sigma(X)$.
c) Assume X has the Bernoulli distribution. Show that $E(X) = p$ and $\mathrm{Var}(X) = p(1 - p)$. ■

If X and Y are random variables, then an application of Th. 1.3 shows that

$$\mathrm{cov}(X, Y) \equiv E([X - E(X)][Y - E(Y)]) = E(XY) - E(X)E(Y). \tag{1.17}$$

The quantity $\mathrm{cov}(X, Y)$ is called the *co-variance* of X and Y. From Th. 1.3, we learn that $\mathrm{cov}(X, Y) = 0$ if X and Y are independent. The co-variance quantifies how much X depends on Y. Note that

$$\mathrm{cov}(X, X) = \mathrm{Var}(X)$$

If $\vec{X} = (X_1, \ldots, X_n)$ is a vector of random variables, then the $n \times n$ co-variance matrix $\Sigma = \mathrm{cov}(\vec{X})$ of $\vec{X}$ has (i, j) entry defined by $\Sigma_{ij} \equiv \mathrm{cov}(X_i, X_j)$. Σ is also called the *variance* of the random vector $\vec{X}$ and is also denoted by $\mathrm{Var}(\vec{X})$ and the formula

$$\mathrm{cov}(\vec{X}) = E([\vec{X} - E(\vec{X})][\vec{X} - E(\vec{X})]^{\rm T})$$

is also used to define Σ (taking the expectation values coordinate (matrix entry) wise).

1.3.1 Applications

Binomial random variables. We continue the discussion §1.2.4 on binomial distributions.
Let Y be a random variable with binomial distribution f_n as in (1.6). The expectation is then

$$E(Y) = \sum_{k=0}^{n} k\, f_n(k) = \sum_{k=1}^{n} k \binom{n}{k} p^k (1-p)^{n-k}.$$

To simplify the expression for $E(Y)$, recall that Y can be written as a sum $\sum_{i=1}^{n} X_i$ of n independent identical Bernoulli random variables X_i with the same distribution f. If B is a Bernoulli random variable on $[0,1]$ with distribution f, then, as in Ex. 1.10, $X_i(\omega_1,\ldots,\omega_n) \equiv B(\omega_i)$ for each $i = 1,\ldots,n$, defines n independent random variable on $\Omega \equiv [0,1]^n$ all with distribution f. Moreover,

$$Y(\omega_1,\ldots,\omega_n) = \sum_{i=1}^{n} X_i(\omega_1,\ldots,\omega_n) = \sum_{i=1}^{n} B(\omega_i). \tag{1.18}$$

Therefore, since $E(X_i) = E(B) = p$ (cf. Exer. 1.10.c)), the linearity of E and Th. 1.3 shows that

$$E(Y) = np. \tag{1.19}$$

Similarly, since, $\mathrm{Var}(X_i) = \mathrm{Var}(B) = p(p-1)$ (cf. Exer. 1.10.c)), an expression for the variance $\mathrm{Var}(Y)$ can be derived using Exer. 1.9.c):

$$\mathrm{Var}(Y) = \sigma(Y)^2 = np(1-p). \tag{1.20}$$

In particular, according the the Theorem of Moivre–Laplace, we have for large n that

$$f_n(k) \approx \frac{1}{\sigma} f_{\mathrm{st}}\left(\frac{k-\mu}{\sigma}\right), \quad \text{where} \quad f_{\mathrm{st}}(x) \equiv \frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2}), \quad \mu \equiv E(Y), \quad \sigma \equiv \sigma(Y).$$

Exercise 1.11: *Mean and variance of the binomial distribution.*
The expressions for $E(Y)$ and $\mathrm{Var}(Y)$ in the above example can also be derived with algebraic arguments by noting that (see also Exer. 1.6 and Exer. 1.5)

$$((1-p)+px)^n = \sum_{k=0}^{n} f_n(k)\, x^k \quad (x \in \mathbb{R}). \tag{1.21}$$

Taking the derivative w.r.t. x yields $np((1-p)+px)^{n-1} = \sum_{k=1}^{n} f_n(k)\, k\, x^{k-1}$. With $x = 1$, we find (1.19). Use similar arguments to obtain (1.20). (Hint: use also the second derivative of (1.21).) ■

In §2.1, we will design algorithms to generate sequences $\omega_1, \omega_2, \ldots$ of numbers in $[0,1]$. The numbers in such a sequence should be random according to the uniform distribution. In order to check how successful we are in achieving this goal, we may select a sub-interval $[a, a+p]$ of $[0,1]$ and an $n \in \mathbb{N}$, and count, for $i \leq n$, the number of ω_i in $[a, a+p]$:

$$y \equiv \#\{i \leq n \mid \omega_i \in [a, a+p]\}.$$

The probability of having $\omega_i \in [a, a+p]$ should be equal to p. The probability of having $y = k$ should be equal to $f_n(k)$ with f_n the binomial distribution. 'Count the number of ω_i, $i \leq n$, in $[a, a+p]$', is the random variable Y from (1.18). Y has distribution f_n and y is a realisation of Y.
We learnt that if we shift Y by its expectation $\mu \equiv E(Y) = np$ and then scale it by its standard deviation $\sigma \equiv \sigma(Y) = \sqrt{np(1-p)}$, to

$$\widetilde{Y} \equiv \frac{Y-\mu}{\sigma},$$

then, for large n, the associated distribution $\widetilde{f}_n$ approximates the standard normal distribution at the points $(k-\mu)/\sigma$:

$$\widetilde{f}_n(x) = \sigma f_n(\sigma x + \mu) \approx \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \quad \text{for} \quad x = \frac{k-\mu}{\sigma} \quad (k = 0, 1, \ldots, n)$$

and $\widetilde{f}_n(x) = 0$ for other values of x. Note that for n large, the collection of $x_k \equiv (k-\mu)/\sigma$ is 'dense' in $\mathbb{R}$: in some sense the $\widetilde{f}_n$ converge towards the standard normal distribution for $n \to \infty$. The cumulative distribution $\widetilde{F}_n$ associated with $\widetilde{f}_n$, $\widetilde{F}_n(x) \equiv \sum_{x_k \leq x} \widetilde{f}_n(x_k)$ $(x \in \mathbb{R})$, is a *step* function. The $\widetilde{F}_n$ converge (both uniformly and in 2-norm) for $n \to \infty$ towards the cumulative standard normal distribution.

In the above approach for testing randomness with uniform distribution of our generated sequence, we focused only on one part of the interval $[0,1]$, that is, on $[a, a+p]$. The *Pearson's chi-square test* brings in information from other sub intervals as well.
Let $\{I_1, \ldots, I_M\}$ be a partitioning of $[0,1]$ into disjoint intervals: $I_j \equiv [a_j, a_{j+1}) \equiv [a_j, a_j + p_j)$ with $p_j \in (0,1)$ $(j = 1, \ldots, M)$ such that $\sum_{j=1}^M p_j = 1$ and $a_1 = 0$. Let B_j the random Bernoulli variable for interval I_j:

$$B_j(\omega) = 1 \quad \text{if} \quad \omega \in I_j \quad \text{and} \quad B_j(\omega) = 0 \quad \text{if} \quad \omega \in [0,1] \backslash I_j.$$

Then (cf., (1.18))

$$y_j \equiv \sum_{i=1}^n B_j(\omega_i) = \#\{i \leq n \mid \omega_i \in I_j\}$$

counts the number of $\omega_i \in I_j$ among the first n ω_i. Let Y_j be the associated random variable. As we know, with $\mu_j \equiv E(Y_j) = np_j$ and $\sigma_j \equiv \sqrt{np_j(1-p_j)}$, the shifted and scaled version $(Y_j - \mu_j)/\sigma_j$ resembles a random variable with standard normal distribution. Now it is tempting to consider the square sum of these variables hoping to be able to exploit properties from chi-square distributions. However, the variables are not independent. Not only add all Y_j to n, but there is all ready dependency between two Y_j, because, for any $\omega \in [0,1]$, we have that

$$B_i(\omega) B_j(\omega) = 0 \quad (i \neq j):$$

an ω can not be in two intervals at the same time. A careful analysis by Pearson (around 1900) revealed the following result.

Theorem 1.5 (Pearson)
For $n \in \mathbb{N}$, let V be the random variable

$$V \equiv \sum_{j=1}^M \frac{[Y_j - E(Y_j)]^2}{E(Y_j)}. \tag{1.22}$$

For large n, V is approximately χ^2_ν distributed for $\nu \equiv M-1$ degrees of freedom. ♦

Exercise 1.12: *Pearson's Theorem.*
Prove (directly), for $M = 2$, i.e., $p_1 + p_2 = 1$, (and all n), that

$$V \equiv \frac{[Y_1 - E(Y_1)]^2}{E(Y_1)} + \frac{[Y_2 - E(Y_2)]^2}{E(Y_2)} = \left(\frac{Y_1 - E(Y_1)}{\sigma(Y_1)}\right)^2 = \left(\frac{Y_2 - E(Y_2)}{\sigma(Y_2)}\right)^2. \qquad \blacksquare$$

1.4 The law of large numbers

We will also need a famous theorem known as 'the law of large numbers'. This theorem states in what way the average over a large number of experiments tends to the mean.

Theorem 1.6 (Law of Large Numbers)
Let $X_1, X_2, \ldots,$ be a sequence of independent random variables with identical distribution. If μ the mean of these variables, $\mu \equiv E(X_i)$ and $\overline{X}_n$ is the random variable arising by averaging the first n variables,

$$\overline{X}_n \equiv \frac{1}{n}\sum_{i=1}^{n} X_i,$$

then, for any $\varepsilon > 0$, we have that $\quad P(|\overline{X}_n - \mu| > \varepsilon) \to 0 \qquad (n \to \infty).$ ♦

Example 1.13 (continues Ex.1.11 and Ex. 1.12)
Recall that a realisation of $\overline{X}_n$ in this example is a sampled mean:

$$\overline{X}_n(\omega_1, \omega_2, \ldots) = \frac{1}{n}\sum_{i\leq n} X_i(\omega_1, \omega_2, \ldots) = \frac{1}{n}\sum_{i\leq n} Z(\omega_i).$$

Therefore, in this setting, the Law of Large Numbers reads

$$P\left(\left\{(\omega_1, \omega_2, \ldots) \mid \omega_i \in [0,1], \left|\frac{1}{n}\sum_{i=1}^{n} Z(\omega_i) - E(Z)\right| > \varepsilon\right\}\right) \to 0 \qquad (n \to \infty).$$

Basically, this states that for large numbers (large n) of random samples ω_i, it is extremely unlikely that the average of the realisations $z_i \equiv Z(\omega_i)$ of Z deviates even a little (ε) from the expectation $E(Z)$. ♦

Exercise 1.13: *Law of large numbers.*
Check that result (1.19) in Example 1.3.1 is in line with the result in Example 1.13. ■

The theorem below is a stronger variant of the law of large numbers. It states that the sampled mean estimates the expectation: the average of the random realisations tends to the mean with 100% probability if the sequence $(\omega_1, \ldots, \omega_n)$ of random samples becomes long.

Theorem 1.7 (Law of large numbers)
Let Z be a random variable on Ω. Let $(\omega_1, \omega_1, \ldots)$ be a sequence of random samples form Ω. Then,

$$\overline{z}_n \equiv \frac{1}{n}\sum_{i=1}^{n} z_i = \frac{1}{n}\sum_{i=1}^{n} Z(\omega_i) \to E(Z) \qquad (n \to \infty) \tag{1.23}$$

is not true with probability 0. ♦

In the remainder of these Lecture Notes, we will be a bit less strict in our formulations: we will simply state that (1.23) holds without mentioning precautions as 'not true with probability 0' or 'true with probability 1'.

With $X_i(\omega_1, \omega_2, \ldots) \equiv Z(\omega_i)$ and a a sequence $(\omega_1, \omega_2, \ldots)$ of samples in $[0,1]$, $\overline{z}_n$ is a realisation of $\overline{X}_n$ (see Example 1.11). According to (1.15), the error $|\overline{z}_n - E(Z)|$ in $\overline{z}_n$ as an estimator for $E(Z)$ will *probably* be less than $\sqrt{\mathrm{Var}(Z)/n}$, that is, when randomly taking many sequence $(\omega_1, \omega_2, \ldots)$ of samples ω_i then the average error will be less than $\sqrt{\mathrm{Var}(Z)/n}$ (to see this, view $\overline{z}_n$ is a realisation of $\overline{X}_n$ and apply (1.15) and the law of large numbers): $\sqrt{\mathrm{Var}(Z)/n}$ is an estimator for the error $|\overline{z}_n - E(Z)|$. Clearly $\sqrt{\mathrm{Var}(Z)/n} \to 0$ for $n \to \infty$. Therefore, (1.23) might have been anticipated. However, this argument tells only something about an average result from many sequences $(\omega_1, \omega_2, \ldots)$ of samples, whereas, (1.23) is a statement on only one sequence $(\omega_1, \omega_2, \ldots)$ of samples: (1.23) can be viewed as a stronger version of (1.15).

The estimator $\sqrt{\mathrm{Var}(Z)/n}$ for the error can also be estimated from the value of realisation z_i of Z by applying the law of large numbers and (1.15) or (1.16). For details, see Corollary 1.1 below and the subsequent discussion.

Exercise 1.14: *The Law of Large Numbers for variances.*
Consider the setting of Theorem 1.6 and prove the following consequence of the Law of Large Numbers. Let σ^2 be the variance of the random variables X_i: $\sigma^2 \equiv \text{Var}(X_i)$. For any $\varepsilon > 0$ we have that

$$P(|\overline{Y}_n - \sigma^2| > \varepsilon) \to 0 \quad (n \to \infty), \quad \text{where} \quad \overline{Y}_n \equiv \frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)^2. \qquad \blacksquare$$

From the observation in the above exercise and (1.23), we may conclude that, for a random variable Z and a sequence (ω_i) of random samples, the variance can be estimated from the realisations $z_i \equiv Z(\omega_i)$:

Corollary 1.1
With Z, Ω and $(\omega_1, \ldots)$ as in Theorem 1.7, then, with probability 1, we have that

$$\frac{1}{n}\sum_{i=1}^{n}[z_i - \overline{z}_n]^2 = \left(\frac{1}{n}\sum_{i=1}^{n} z_i^2\right) - (\overline{z}_n)^2 \to \text{Var}(Z) \quad (n \to \infty). \tag{1.24}$$

♦

Note that $\sum_{i=1}^{n}[z_i - \overline{z}_n]^2/n$ is a realisation of the random variable $\sum_{i=1}^{n}[X_i - \overline{X}_n]^2/n$ with expectation equal to $(n-1)\,\text{Var}(Z)/n$, see (1.16). For this reason, the quantity $\sum_{i=1}^{n}[z_i - \overline{z}_n]^2/n$ in (1.24) is called a *biased* estimator for the variance $\text{Var}(Z)$, or biased sample variance, while

$$\frac{1}{n-1}\sum_{i=1}^{n}[z_i - \overline{z}_n]^2 \tag{1.25}$$

is called is an *unbiased* estimator, or unbiased sample variance. Note that for $n = 1$ (one sample), the biased estimator equals 0, where the unbiased estimator is undefined, as it should be: the variance of a random variable can not be estimated form one realisation only.

In summary, the average $\overline{z}_n$ of n realisations $z_i \equiv Z(\omega_i)$ of a random variable Z estimates the expectation $E(Z)$ with an error that is probably less than $\sqrt{\text{Var}(Z)/n}$. $\text{Var}(Z)$ can be estimated by the quantity in (1.25) (or the one in (1.24)). The estimators approach exactness for $n \to \infty$. Note that the second expression in (1.24) allows efficient updating, in contrast to the expression in (1.25).

Chapter 2

Random number generators

Random number generators are algorithms that aim to produce sequences of random numbers according to a specified distribution. In this chapter, we discuss such algorithms and their implementations and ways of testing them in order to identify the good ones. The topics covered here include: linear congruential generators, non-uniform random variables, and statistical tests. The generators developed in this chapter can be used in, for instance, Monte Carlo integration and Genetic Algorithms, as described in the next chapters.

2.1 An Introduction to Random Number Generators

A random number generator (RNG) is a means for obtaining (uniformly distributed) random numbers. A roulette table, for example, is an RNG producing random values between 1 and 35, supposedly from a uniform distribution. For practical purposes physical systems are unsuitable for random number generation (not reproducible!). There are the following considerations to take into account:

- **Randomness:** Obviously we want our numbers 'as random as possible'. One can argue about a proper definition of randomness, but we will take the pragmatic viewpoint that our generator should pass a number of *statistical tests* (see Section 2.5).
- **Reproducibility:** In modern science results should always be reproducible, others should be able to verify your results. Therefore it is important that our generator can reproduce the exact same sequence of numbers as generated before. We can safely say that no physical system is able to do this, a temperature sensor might produce random numbers, but it does not generate reproducible sequences. Results should also be independent of the hardware and operating system used (*portability*).[1]
- **Efficiency:** If you need a huge amount of random numbers, your generator should be fast! We would like to keep the operations needed to construct the numbers to a minimum.

A computer based RNG is often called a 'pseudo random number generator' (PRNG), since the computer is completely deterministic and can therefore never truly randomly produce numbers. Popular PRNGs use iterations of the form

$$x_{i+1} = f(x_i)$$

to generate a sequence of numbers. The first number x_0, often supplied by the user, is called the *seed* of the generator.

[1] In some applications, pseudo random number generators (PRNG, to be discussed below) are used precisely for their reproducibility, while the fact that the produced numbers are seemingly random is irrelevant. For an example, see Footnote 10.

The generators that we will consider produce integers x_i in the set $\{0, 1, 2, \ldots, m-1\}$. To obtain numbers ω_i in $[0, 1]$, we simply scale:

$$\omega_i \equiv \frac{x_i}{m-1}.$$

Exercise 2.1: *Scaling.*
Or should we scale like $\frac{x_i}{m}$ or $\frac{x_i+\frac{1}{2}}{m}$? ■

We first focus (in §2.2) on producing sequences of numbers x_i such that the scaled versions of sequences of ω_i in $[0, 1]$ can pass tests of being random with a uniform distribution in $[0, 1]$. Some of these tests are suggested in §2.5. But before that we will explain in §2.4 how these 'uniform' sequences can be adopted to find sequences of 'random' numbers with other prescribed distributions. In §2.6, we briefly discuss generalisations and more recent developments in RNG.

In the next section, we examine a particular choice for f.

2.2 Linear congruential generators

Popular choices for f form the so-called *linear congruential (random number) generators* (LCRNG):

$$f(x) = (ax + c) \bmod m \qquad (x \in \mathbb{Z}), \tag{2.1}$$

where a, c and m are (carefully) preselected positive integers. Here, $\bmod m$ takes values modulo m. To be more precise, a $z \in \mathbb{Z}$ can uniquely be decomposed as

$$z = mj + r$$

with $j \in \mathbb{Z}$ and an $r \in \{0, 1, 2, \ldots, m-1\}$. We put $z \bmod m \equiv r$ and $z \operatorname{div} m \equiv j$.
For example, since $7 = 3 \cdot 2 + 1$ we have that $7 \bmod 3 = 1$ and $7 \operatorname{div} 3 = 2$.

Note that we may assume a and c to be in $\{0, 1, \ldots, m-1\}$ (why?). The LCRNG is called *multiplicative* if the, so-called, *carry* c equals 0 and is said to be *mixed* otherwise.

Calculating $\bmod m$ allows us the use only m different numbers. Therefore, after at most m steps the iteration $x_{i+1} = f(x_i)$ will have repeated itself, but repetition can occur earlier. We say that the generator has a cycle of *period* n when $x_{i+n} = x_i$ with n smallest, where we minimise over all i and all integers x_0. The generator has *full period* if $n = m$ if $c \neq 0$ and $n = m-1$ in case $c = 0$.

☞ **Exercise 2.2:** *Periods of poorly chosen iterations.*
a) Why is it undesirable to have an $x_i = 0$ in case $c = 0$?
b) Compute the period of $5\,x \bmod 13$ and of $7\,x \bmod 13$.
c) Compute the period of $(3\,x + 4) \bmod 60$. Note that, even for large m, the period can become small. ■

Exercise 2.3: *Increasing the period.*
a) Compute the period of $(7\,x + 3) \bmod 10$.
b) Consider the variant $y_i = 7x_i + c_i,\ x_{i+1} = y_i \bmod 10,\ c_{i+1} = y_i \operatorname{div} 10$ with seeds x_0 and c_0.
Compute the period of this variant with variable carry and compare with the fixed carry variant in a).
c) Show that y_i of b) satisfies $y_i = a\,y_{i-1} \bmod m'$ with $m' \equiv am - 1 = 69$, where $m = 10$ and $a = 7$: x_{i+1} is the $\bmod 10$ result of the y_i generated with a 'larger m'. The x_i inherit the period of the y_i. ■

Generators with full period 'shuffle' the numbers $(0,)1, 2, \ldots, m-1$ in one cycle. Note that a true RNG that generates numbers in $\{(0,)1, 2, \ldots, m-1\}$, is likely to have produce some numbers more than once in a sequence of length m $(m-1)$ and may have 'missed' others.

We can only expect to find numbers ω_i ($\omega_i \equiv \frac{x_i}{m}$) that can pass a test on randomness with uniform distribution if the period is large (and a or x_0 is not small, because, with, say $a = 2$ and $c = 0$, the sequence

"Park–Miller"	$a = 16807 = 7^5$	$c = 0$	$m = 2^{31} - 1$
No name	$a = 65539$	$c = 0$	$m = 2^{31} - 1$
A sample bad generator	$a = 5$	$c = 0$	$m = 2^7$
RANDU	$a = 65539$	$c = 0$	$m = 2^{31}$
quick	$a = 1664525$	$c = 1013904223$	$m = 2^{32}$
UNIX	$a = 1103515245$	$c = 12345$	$m = 2^{31}$

Table 2.1: Parameter values for LCRNGs.

$x_0 = 1, 2, 4, 8, 16, \ldots$ will certainly fail the randomness test. To have 'randomness' for any seed, a has to be non small).

Some values for the parameters that have been suggested in the literature (for example see [33]) are given in table 2.1.

The "Park–Miller" generator was proposed in the classic paper [30]. It is viewed as a '*minimal standard*' generator: the generator has full period, the generated numbers are highly statistically independent (pass basic statistical tests on randomness with uniform distribution) and an implementation is possible on any machine (independent of architecture). For a correct implementation, to avoid problems with 'overflow', Schrage's trick is needed, which will be discussed below.

☞ The first paragraph of Section 2.3 gives a short rehearsal of binary numbers on computer systems, which may help you to more easily understand the following paragraph.

The linear congruential generators can produce very large numbers. It might happen that the product ax becomes larger than the computer can handle ('overflow'). Let's make this more explicit. For efficiency we work with integers, which have 32 bits of data, 31 for the value and one for the sign, thus we have integers in the range $[-2^{31}, 2^{31})$. Now, how does the computer handle products of numbers that result in more than 32 bits? The computer truncates the product to a 32-bit number simply by throwing away the left *most* bits.[2] This is mathematically equivalent to working $\bmod 2^{32}$. Dropping the left most bits may be nicer for our RNGs than dropping the right ones: the value of the right most bits are 'harder to predict' and can be considered to be somewhat 'more random'. Unfortunately, the left most bit (after truncation to 32 bits) is the sign bit. It can be 1 and then our numbers are negative. Therefore, we also need another procedure for normalising our random numbers to $[0, 1)$. Division by m will not do the trick. Think about how you would do the normalisation. Also note that in 32-bit arithmetic the $\bmod 2^{32}$ is for free (see next section) if we use `unsigned ints`. The generator in Table 2.1 that allows to exploit this fast mod operation, is referred to as the 'quick' one.

The choice $m = 2^b$ for some $b \in \mathbb{N}$ nicely fits the fact that the computer works in the binary number system. Moreover, the case where the linear congruential generator has full period can easily be characterised (cf., e.g., [16]):

with $m = 2^b$, we have full periodicity if and only if $a \bmod 4 = 1$ and c is odd.

Unfortunately, mixed linear congruential methods with $m = 2^b$ appear to exhibit periodic behaviour with low period in the right most bits (see also Exer. 2.4). For this reason, multiplicative linear congruential generators with m prime are preferred. For these methods 'overflow' is harder to handle. Simply letting the numbers overflow is of course a bit fishy: we are no longer really iterating (2.1) then. The alternative, to 'cast' all integer numbers involved to reals (that is, to `doubles` in C++),[3] to do the computation in real arithmetic, and then to cast back the result to integers (`unsigned ints`) will highly effect the efficiency of the code. Fortunately, for a number of combinations of m and a, there is a trick to do $(ax) \bmod m$ exactly in 32 bits even if ax overflows. This trick, *Schrage's trick*, is inspired by the observation that if m would be factorisable as $m = aq$, then $ax \bmod m$ would be equal to $a(x \bmod q)$ and no number larger than m would be needed to 'generate' the sequence of x_i ($x_{i+1} = ax_i \bmod m = a(x_i \bmod q)$). Obviously

[2]The left most bit is also referred to as the most significant bit.

[3]`static_cast<double>(a)`, `static_cast<double>(x)`, etc.. On a 64-bit machine (cf., Footnote 4), it suffices to cast the integers to `long ints`.

a prime number can not be factorised, but if $m = aq + r$ with r small, then an adapted formula that avoids large numbers might be possible. *Schrage's algorithm* is based on this idea.

Schrage's algorithm. Factorise m as

$$m = aq + r \quad \text{with} \quad q \equiv m \operatorname{div} a \quad \text{and} \quad r \equiv m \bmod a.$$

Schrage proved that with

$$\begin{gathered} b \equiv a(x \bmod q) - r(x \operatorname{div} q) \\ ax \bmod m = b \quad \text{if} \quad b \geq 0 \quad \text{and} \quad ax \bmod m = b + m \quad \text{if} \quad b < 0. \end{gathered} \tag{2.2}$$

Moreover, non of the quantities to compute b exceeds the value of m provided that $r < q$.

Although the restriction $r < q$ limits the use of Schrage's algorithm, it can be used in an implementation of the *Park–Miller generator* (check this) where with $c = 0$ and $m = 2^{31} - 1$, $a = 7^5$ (see Table 2.1) and also in *Fishman's variants*, where for the same m and c, $a = 48271$ or $a = 69621$. (What about the other multiplicative linear congruential generator in Table 2.1 with m prime?)

Exercise 2.4: *UNIX.*
The standard random generator UNIX of the Unix operating system is known to have bad randomisation in the right most bits. As a remedie, Unix generates x_i, but it only outputs the left half of the bits of x_i: $b_{30}b_{29}\ldots b_{15}$ (that is, $\sum_{k=0}^{15} b_{15+k}\, 2^k$) rather than $x_i = b_{30}b_{29}\ldots b_0$. Discuss the consequences of this strategy (with respect to periodicity, etc.). ■

2.3 Implementation issues

Let us consider the binary number system. Integer numbers in the computer are stored as a row of bits, zeros and ones. A positive number x is represented in *binary* as $x = b_{n-1}\ldots b_0$ where the bit $b_i \in \{0, 1\}$. It is based on a representation by powers of two:

$$x = \sum_{i=0}^{n-1} b_i 2^i.$$

For example the number 23 is represented by 00010111 since $23 = 2^4 + 2^2 + 2^1 + 2^0$. The number of bits that can form an integer in the computer limits the largest number that can be stored. An n-bit positive number is evidently in $\{0, 1, 2, \ldots, 2^n - 1\}$.

How is addition performed by the computer when the result is larger than the number of bits available for an integer? The leftmost (unfortunately the most significant) bits are thrown away. For instance, $128 + 130 = 2$ in 8-bit arithmetic, because $10000000 + 10000010 = 100000010$ but the leftmost bit does not fit and is discarded, resulting in 00000010. You should now realise that what we are doing is in effect working modulo 2^8!
To conclude, addition (and subtraction) in n-bit arithmetic is equivalent to calculations modulo 2^n. The biggest number we can make in the C++ language using standard data types is of the type `unsigned int` which is 32 bits,[4] addition of `unsigned ints` is thus addition modulo 2^{32}. For multiplication we have the same result, as you can check this for yourself. The relevance for random number generation is of course the following: when calculating $f(x_i) = (ax_i + c) \bmod 2^{32}$, using `unsigned int`, the modulo 2^{32} operation is *for free*.

Now we introduce the C++ shift operators '<<' and '>>'. The command `a << b` will shift the bits in a to the left, b positions. Thus, the result is multiplication of a with 2^b. But be careful, bits at leftmost

[4]On 32 bit platforms, there is generally no difference between `ints` and `long ints`, both use 4 bytes (32 bits). One byte consists of 8 bits as 00010111. On 64 bit platforms `ints` are still 32 bits, while `long ints` are 64 bits.

position are 'pushed out' and disappear. Therefore `128 << 1` equals zero in 8-bit arithmetic. The '>>' operator is a completely analogous shift to the right, yielding a rounded division by powers of two. These two operators are very useful since they are executed very rapidly by the computer. You should now see why `(1<<31)-1` is a quick way to obtain the largest possible '`signed int`'-number. The compiler might give you a warning message, but that is OK, we created the overflow on purpose.

Note that when $c = 0$ and and the modulus is not equal to 2^{32} you may use Schrage's trick as explained in the previous section, however if $c \neq 0$ and also the modulus is not equal to 2^{32}, then you have to go through a lot of trouble to correctly evaluate $(ax_i + c) \mod m$ and your RNG is probably not efficient anymore.

2.3.1 Negative numbers

Working with signed integers is a bit more tricky. In this case the leftmost bit is used as a *sign bit*. If it is zero, the remaining 31 bits give us an integer in $[0, 2^{31})$. If it is one the remaining 31 bits encode a negative number, but in a special way called *two's complement*.[5]. The negative number $-a$ is encoded by inverting all bits of a and adding one. So, in 4 bit arithmetic, with one sign bit, the number -3 is encoded as 1101. What we did is write down the number which you would have to add to a positive number to get zero. Indeed $1101 + 0011 = 0000$ in 4 bit arithmetic! Using two's complement the computer hardware can add and subtract numbers the same way whether they are signed or unsigned. Effectively, we are working in $\mathbb{Z}/m\mathbb{Z}$ with $m = 2^{32}$ and the upper half of $\{0, 1, \ldots, m-1\}$ represents our negative integers. However, taking $\bmod m$ with $m = 2^{31}$ is no longer for free now. The best way to handle negative numbers is probably to work with positive numbers initially and, in the last step, 'normalise' the result to a positive number in case it is negative. This suggestion is inspried by the result in the following exercise.

Exercise 2.5: *Modulo m and modulo 2m.*
Let x and m be integers, $m > 0$.
Show that with $y \equiv x \bmod 2m$ and $z \equiv y \bmod m$ we have that $z = x \bmod m$. ■

★ **Exercise 2.6:** *Implementation of RNGs.*
Implement the generators from Table 2.1 in C++. Sample code is available from the course website [36] to get you started. Look at Appendix C if the code structure is unclear. Also remember that code documentation is available at the course webpage [36].
– Implementation of the 'quick' generator needs special attention ($m = 2^{32}$ equals 0 when working with `unsigned ints`. How do you deal with $\bmod m$ to form x_{i+1} and with $\cdot/(m-1)$ to form ω_{i+1}?).
– If possible and if it helps to avoid overflow, use Schrage's trick for the other generators. (Note that the detour via reals as suggested in the paragraph before Schrage's algorithm can be used for testing correctness of your code.)
– Make sure that your code is efficient (also test for efficiency; timings).
As explained in Appendix C, the sample code from the course website also allows easy inclusion of external random number generators. Include one from the standard C library. (You may also have a look at the more advanced generators of §2.6 as the Mersenne Twister as mentioned at the end of §2.6.5.) ■

2.4 Non-Uniform distributions

The RNGs that we discussed so-far aim for producing numbers sampled from its sample space (as $[0, 1]$) with a uniform distribution. For some purposes it is necessary to draw numbers with a probability according to a certain prescribed distribution function (as the normal distribution). There are two main methods for doing this. The *inversion method* in the next subsection is the most straightforward one: it adjusts the

[5]An interesting discussion of two's complement and other techniques for negative integer representation can be found at Wikipedia: `http://en.wikipedia.org/wiki/Signed_number_representations`

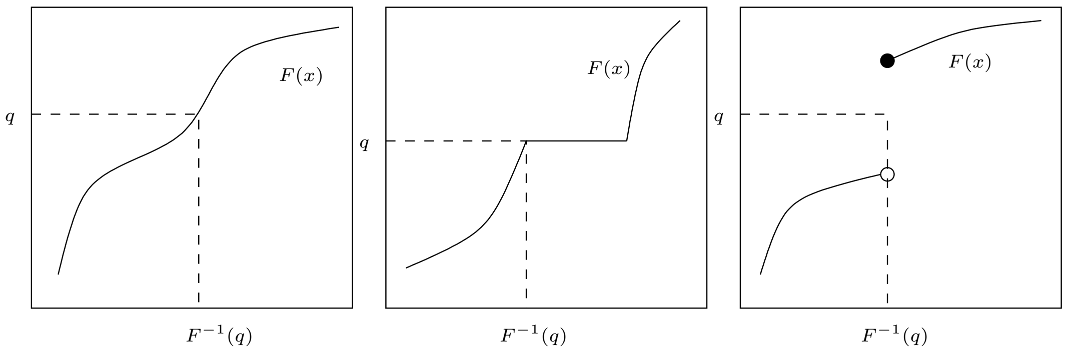

Figure 2.1: The generalised inverse for a continuous strictly monotone function (left), a continuous not strictly monotone function (middle) and a discontinuous function (right). (Note: the q in the figures correspond to the u in the text.)

domain of uniform distribution (by locally 'stretching' the domain to obtain lower density and locally 'squeezing' for higher density). Another method, the *rejection method* in §2.4.2 obtains locally a lower density by locally discarding ('rejecting') a part of the random numbers. Discarding is according to some randomness: this method combines two RNGs.

2.4.1 Non-uniform distribution: the inversion method

In Ex. 1.8, we learnt that the random variable X, with $X(\omega) \equiv \tan((\omega - \frac{1}{2})\pi)$ $(\omega \in [0,1])$, leads to the non-uniform distribution f with $f(x) = 1/[\pi(1+x^2)]$ $(x \in \mathbb{R})$. X 'stretches' the lower part of $[0,1]$ towards $-\infty$ and the upper part towards $+\infty$. As a consequence, the realisations $x_1, x_2, \ldots$ of X are more densely around 0 than around large (in absolute value) numbers. Here, $x_j \equiv X(\omega_j)$ and $\omega_1, \omega_2, \ldots$ are (uniformly) randomly sampled from $[0,1]$. Actually, $\#\{j \leq n \mid x_j \in [a,b]\}/n \to \int_a^b f(x)\,dx$ for $n \to \infty$. Note that the cumulative distribution F is given by $F(x) = \int_{-\infty}^{x} f(y)\,dy = \frac{1}{\pi}\arctan(x) + \frac{1}{2}$, which is precisely the inverse of X. This example suggests the inversion method: to obtain a random variable X (on the sample space $[0,1]$) with a prescribed distribution function f, compute and invert the cumulative distribution F.

Any cumulative distribution F is *increasing*, i.e., $F(x) \leq F(y)$ if $x \leq y$. However, the increase need not be strict ($F(y)$ may be equal to $F(x)$ also for $x < y$). If the associated random variable X is continuous, then F is continuous as well.[6] However F need not be continuous if X is discreet (or a mixture of discrete and continuous). Therefore, defining the 'inverse' of a cumulative distribution also for the general situation is a bit tricky:

Definition 2.1 (Generalised Inverse)
The generalised inverse of a cumulative distribution function F is defined by

$$F^{-1}(u) \equiv \inf\{x \in \mathbb{R} \mid F(x) \geq u\} \qquad (0 < u < 1). \qquad \blacklozenge$$

Figure 2.1 clarifies how the generalised inverse for such functions is defined.

Before stating the main theorem we prove the following lemma.

Lemma 2.1
For all $x \in \mathbb{R}$ and $\omega \in (0,1)$ we have, for right-continuous F,[7] that

$$u \leq F(x) \quad \Leftrightarrow \quad F^{-1}(u) \leq x. \qquad \blacklozenge$$

[6]In this situation, F is even 'absolute' continuous and, in some sense, $F' = f$.
[7]F is right continuous if for all $x \in \mathbb{R}$ we have that $F(x+h) \to F(x)$ if $h > 0, h \to 0$.

Proof. The conclusion $x \geq F^{-1}(u)$ if $u \leq F(x)$ follows immediately form the definition of $F^{-1}(u)$. Now suppose that $F^{-1}(u) \leq x$. The assumption that F is right continuous implies that

$$F(\inf(\{y \mid y \in C\}) = \inf\{F(y) \mid y \in C\} \quad \text{for any subset } C \text{ of } \mathbb{R}.$$

In particular, $F(F^{-1}(u)) = \inf\{F(y) \mid F(y) \geq u\} \geq u$. The fact that F increases implies that $F(F^{-1}(u)) \leq F(x)$, whence $u \leq F(x)$. □

Now the inversion method is easily stated.

Theorem 2.1 (Inversion Method)
Let f be a distribution function with cumulative distribution F. Let U be a random variable on Ω with uniform distribution $\chi_{[0,1]}$. Then the random variable $X \equiv F^{-1}(U)$ on Ω has distribution function f. ♦

☞ **Exercise 2.7:** *Correctness of the Inversion Method.*
Prove Theorem 2.1. (You may take $\Omega = [0,1]$ and $U(\omega) = \omega$. If F is continuous, then we have that $F(x) = \int_\infty^x f(y)\,dy \quad (x \in \mathbb{R})$.) ■

☞ **Exercise 2.8:** *Inverse distribution functions.*
Verify the following table analytically. Here λ and σ are positive real numbers, $\lambda, \sigma > 0$, $x \in \mathbb{R}$, $u \in [0,1]$. In our application, we draw samples (random numbers) ω_i from $[0,1]$ (according to a uniform distribution) and with U mapping ω to ω, the $u_i \equiv \omega_i = U(\omega_i)$ are realisations of U: U is a random variable on $[0,1]$ with uniform distribution.

Name Distribution	Cumulative distribution function	Inverse
Exponential	$F(x) = 1 - e^{-\lambda x}$	$F^{-1}(u) = -\frac{1}{\lambda}\log(1-u)$
Cauchy	$F(x) = \frac{1}{2} + \frac{1}{\pi}\arctan(x/\sigma)$	$F^{-1}(u) = \sigma\tan(\pi(u - 1/2))$

■

2.4.2 The Rejection Method

Sometimes we cannot calculate the inverse of F explicitly, or even F itself. Then it will be computationally costly to compute the inverse numerically and generating random numbers with cumulative distribution F will be costly when using the inversion method. For instance, the normal distribution, as the standard one $f(x) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{x^2}{2})$, is extremely important in practice. Unfortunately, the associated cumulative distribution F can not be expressed in terms of familiar functions as log, sines, polynomials, etc.. Therefore, we have to rely on numerical methods to compute values for $F(x)$ and for $F^{-1}(u)$. In such a case another popular method called *the rejection method* can be used.

For ease of discussion, we assume that $[0,1]$ is our sample space Ω.

Suppose we want realisations of a random variable X with distribution density function f and suppose we can efficiently produce realisations of a random variable Q with a distribution function q for which, for some $c \in \mathbb{R}$,

$$f(x) \leq c\,q(x) \qquad (x \in \mathbb{R}). \tag{2.3}$$

To explain the idea of the rejection method, consider a sequence $(x_1, x_2, \ldots)$ of realisations x_i of Q, i.e., $x_i \equiv Q(\omega_i)$, where the ω_i in $[0,1]$ have been randomly sampled with uniform distribution. The density of this sequence is according to q, that is, for all intervals $[a,b]$ of $\mathbb{R}$, we have that

$$\frac{\#\{j \mid j \leq n \ \&\ x_j \in [a,b]\}}{n} \to \int_a^b q(x)\,dx \quad \text{for} \quad n \to \infty.$$

To have a density according to f or, of you wish, according to f/c, simply reduce the numbers of x_i in $[a,b]$ by a factor $(\int_a^b f(x)\,dx)/(c\int_a^b q(x)\,dx)$, that is, by approximately $\approx f(a)/(cq(a))$ if $b \approx a$. If this

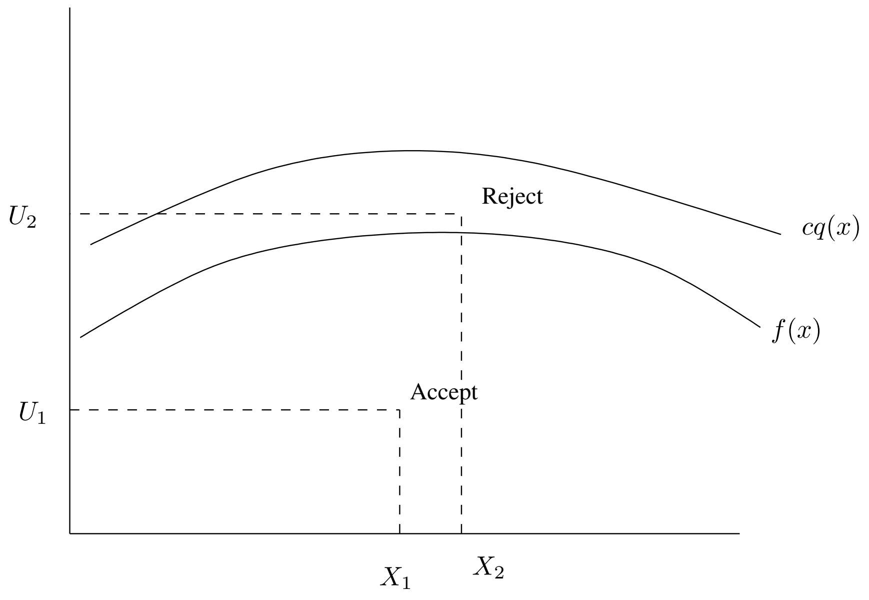

Figure 2.2: Two pairs of random realisations (x_1, u_1) and (x_2, u_2) have been generated ((X_1, U_1) and (X_2, U_2) in the picture). One is accepted and the other is not. It is clear that larger values for c would enlarge the area between f and cq and would lead to more rejections.

strategy is followed for all intervals that form a partitioning of $\mathbb{R}$, then we obtain a sequence of numbers with (approximate) density according to f/c. Since the density is scaled such that $\int f(x)\,dx = 1$, the c is irrelevant: by 'sparsifying' the sequence $(x_1, \ldots, x_n)$ the number elements in this sequence reduces, which explains the irrelevance of c. For 'better' randomness, the 'sparsification' should also be random. Therefore, parallel to the x_i, we also select randomly numbers x_i in $[0, 1]$ (with uniform distribution, independent of the samples ω_i that have been used for the x_i: $x_i = \omega_i'$) and we discard, that is, we 'reject', x_i if $x_i > f(x_i)/(cq(x_i))$. Then we (probably) have the required reduction factors. This strategy is known as the rejection method. See Figure 2.2 for a schematic representation. Note that the number of rejections is smaller if cq forms a more tight upper bound of f.

Below, we formally prove that this approach leads to the required result, but first we give an algorithmic formulation in Alg. 2.1. Here, $U_{[0,1]}$ is the random variable with uniform distribution on $[0, 1]$ (i.e., $U_{[0,1]}(\omega) \equiv \omega$ for $\omega \in [0, 1]$) and Q has distribution q.

Algorithm 2.1 The rejection method.

```
// Samples for a random variable $X$ with a prescribed distribution $f$.
Generate random realisations  $x_i = Q(\omega_i)$  of $Q$ and  $u_i = U_{[0,1]}(\omega_i')$  of $U_{[0,1]}$.
if  $u_i\,c\,q(x_i) \leq f(x_i)$  then
    accept $x_i$ as random realisation of $X$.
else
    reject $x_i$.
end if
```

If we can easily find a random variable Q with density q, then this is a nice and straightforward method. To find Q for a given density function q, the inversion method can be used if the primitive of q can be inverted by hand. Note that $\frac{1}{2\pi}\exp(-x^2/2) \leq \frac{1}{\pi(2+x^2)}$ $(x \in \mathbb{R})$. Therefore, a combination of the inversion method for a Cauchy distribution and the rejection method could be used to obtain a random variable with normal distribution.

Example 2.1

If f is a function bounded by M (i.e., $f(x) \leq M$ all $x \in \mathbb{R}$) and zero outside an interval $[a, b]$, then it suffices to take Q to be the uniform distribution on $[a, b]$ and to take $c = (b - a)\,M$. Think of f as a graph and $c\,q$ as the horizontal line lying above it. The pair (x_i, u_i) is then a point in $\mathbb{R}^2$ which may lie below the graph of f ($\rightarrow$ accept) or between the graph of f and c ($\rightarrow$ reject). The only reason for choosing q different from the constant function $q = c$ is to reduce the number of rejections, thereby making the procedure more efficient. On the sample space $[0, 1]$, Q is defined by $Q(\omega) \equiv \omega(b - a) + a$ and the density function q is given by $q(x) = 1/(b - a)$ for $x \in [a, b]$ and $q(x) = 0$ for other x (note that we need $\int_a^b q(x)\,dx = 1$). The algorithm is as in Alg. 2.2. ♦

Algorithm 2.2 The rejection method (simpler variant).

```
// Samples for a random variable $X$ with
// a prescribed distribution $f$ with $f(x) \leq M$ if $x \in [a, b]$ and $f(x) = 0$ for other $x$.
Generate random samples $\omega_i$ and $u_i$ of the uniform random variable $U_{[0,1]}$.
$x_i = \omega_i(b - a) + a$.
if $u_i\,M \leq f(x_i)$ then
    accept $x_i$ as our random realisation of $X$.
else
    reject $x_i$.
end if
```

★ **Exercise 2.9:** *Implementation of the Rejection Method.*
Program the procedure of Example 2.1 for a few bounded density distributions f on $[a, b]$. Generate non-uniform random numbers and test if the result is properly distributed by plotting bar plots. Take $a \neq 0, b \neq 1$ to test whether your code is really correct. Try to decrease the number of rejections by choosing a more suitable density function q. ■

Proof of the correctness of the rejection method

As discussed above, the rejection method aims for obtaining random realisations for a random variable X with non-uniform distribution f on (a subset of) $\mathbb{R}$. This is done by independently taking random realisations x_i of a random variable Q with probability density function q and random realisations u_i for the uniform random variable $U \equiv U_{[0,1]}$, i.e., with probability density function the indicator function $\chi_{[0,1]}$ (for a definition of indicator function, see (1.2)). The variables Q and U must be independent (see, Ex. 1.10 and the subsequent exercise). This can be achieved as follows. If Q and U are defined on $[0, 1]$, then 'lift' Q and U to $[0, 1] \times [0, 1]$ by $Q(\omega_1, \omega_2) \equiv Q(\omega_1)$ and $U(\omega_1, \omega_2) \equiv U(\omega_2)$. This does not affect the associated distributions, but the 'lifted' versions of Q and U are independent random variables on the product probability space with measure P determined by $P([a, b] \times [c, d]) = (b-a)(d-c)$ for sub intervals $[a, b]$ and $[c, d]$ of $[0, 1]$. The function c times q should majorate f for some fixed c, that is,

$$f(x) \leq c\,q(x), \qquad (x \in Q) \quad \text{or, equivalently, } g \leq 1, \text{ where } g(x) \equiv f(x)/(c\,q(x)).$$

Now with realisations x_i and u_i and accept x_i as random realisation for the random variable X with distribution f only if

$$u_i \leq g(x_i). \tag{2.4}$$

Proof. Why does this work? We also view X as random variable on the product space $[0, 1] \times [0, 1]$: $X(\omega_1, \omega_2) \equiv X(\omega_1)$. As observed above, this does not affect the density function of X.

To determine the cumulative distribution function for the newly generated X, we write down the conditional probability of having $X \leq t$, given that the sample is accepted:

$$P\,(X \leq t\,|\,U \leq g(X)) = P\,(X \leq t \text{ and } U \leq g(X))\,/P\,(U \leq g(X))$$

For the equality, we used the definition of conditional probability. It suffices to focus on the numerator: with $t = \infty$, the value of the denominator is obtained.

$$\begin{aligned} P\left(X \leq t \text{ and } U \leq g(X)\right) &= \int_{-\infty}^{t} \int_{0}^{g(x)} 1 \; du \, q(x) \, dx \\ &= \int_{-\infty}^{t} g(x) \, q(x) \, dx = \frac{1}{c} \int_{-\infty}^{t} f(x) \, dx. \end{aligned}$$

For any pdf we know that $\int_{-\infty}^{\infty} f(x)\, dx = 1$, whence $P\left(U \leq g(X)\right) = \frac{1}{c}$ and

$$P\left(X \leq t \,|\, U \leq g(X)\right) = \int_{-\infty}^{t} f(x)\, dx.$$

We see that X has pdf f as required, which concludes the proof. □

We conclude this section with the observation that random numbers can be used to randomly generate objects other than numbers, objects as permutations. This observation may become handy in Chapter 4.

Exercise 2.10: *Random permutations of n points.*
Let n be a positive integer. Put $\mathbf{n} \equiv \{1, 2, \ldots, n\}$. π is a *permutation* of the n numbers $1, 2, \ldots, n$ if $\pi : \mathbf{n} \to \mathbf{n}$ is such that $\pi(i) \neq \pi(j)$ whenever $i \neq j$: a permutation of n numbers 'shuffles' these numbers. For a given sequence $(\omega_1, \omega_2, \ldots, \omega_n)$ of ω_i in $[0, 1]$ let the map π from $\mathbf{n}$ to $\mathbf{n}$ sort the ω_i in non-decreasing order: $\omega_{\pi(i)} \leq \omega_{\pi(i+1)}$ all $i = 1, \ldots, n-1$ (if $\omega_{\pi(i)} = \omega_{\pi(j)}$, then $\pi(i) < \pi(j)$ if $i < j$). Show that π is a permutation.
Now generate permutations as follows. Draw randomly $\omega_1, \omega_2, \ldots, \omega_n, \omega_{n+1}, \ldots$ from $[0, 1]$ according to a uniform distribution. Let π_j sort the j-th set $(\omega_{nj+1}, \ldots, \omega_{n(j+1)})$ of n numbers in non-decreasing order. Check that the π_j are randomly generated permutations. Prove, for $n = 2$ and $n = 3$, that the π_j are random according to a uniform distribution (i.e., on average the permutation (123) occurs as often as the permutation (231), etc.). ■

2.5 Statistical tests

We use our RNGs to produce sequences $\omega_1, \omega_2, \ldots, \omega_n$ of numbers (samples) in $[0, 1]$. The numbers should be random and uniformly distributed in $[0, 1]$. We could use the following tests to gain confidence in our RNGs:

- **The period should be large:** A way of visualising a random sequence is by generating a two dimensional random walk. We start at coordinate $(0, 0)$ and at step i we go up (u) if $\omega_i \in [0, 1/4)$, we go right (r) if $\omega_i \in [1/4, 1/2)$, etc. If the walk repeats its structure after a number of steps then we have detected a period in the sequence. Note that we actually use the ω_i to generate d_i with d_i in the set of four characters $\{u, r, d, l\}$ (up, right,down, left).
- **Uniformity:** Every distinct value ω_i should occur with equal chance. We test this by creating M '*bin*s'. While increasing i from 1 to n, we increase the value y_j of bin j if sample ω_i is in I_j. Here $\{I_1, \ldots, I_M\}$ is a partitioning of $[0, 1]$ into M disjoint sub intervals, i.e., $I_j \equiv [a_j, a_{j+1}) \equiv [a_j, a_j + p_j)$ with $a_1 = 0$, $p_j > 0$ and $\sum_{j=1}^{M} p_j = 1$, for instance, $p_j = 1/M$ all $j = 1, \ldots, M$. Initially the bins are empty, that is, all $y_j = 0$ for $i = 0$. The jth bin acts as a counter.

  As in §1.3.1,

$$y_j \equiv \sum_{i=1}^{n} B_j(\omega_i) \quad \text{with} \quad B_j(\omega) \equiv 1 \text{ if } \omega \in I_j \text{ and } B_j(\omega) \equiv 0 \text{ otherwise.} \qquad (2.5)$$

If the bin acts on random numbers (as it should), y_j is the realisation of a random variable Y_j (as discussed in §1.3.1). The bar-plot of the bins should be approximately flat, and get flatter with increasing sequences of random numbers. Note that this test tests uniformity of the distribution of the ω_i rather than randomness. Pearson's χ^2-test (see below) also tests the randomness.

Note that, with $d_i \equiv j$ if $\omega_i \in I_j$, the test actually tests the distribution of the d_i in the sequence $d_1, d_2, \ldots, d_n$ rather than of the ω_i. For instance, if $M = 10$ and all bins are equally large ($p_j = 1/M$ all j), then d_i is a decimal digit: $d_i \in \{0, 1, \ldots, 9\}$. We can use the ω_i to produce (randomly?) decimal digits ($M = 10$). As a variant, with $M = 26$, we can represent the d_i as lower case letters in our alphabet of 26 letters: we can use the ω_i to produce (randomly?) letters of the alphabet ($M = 26$), etc.. This observation that we are actually not testing ω_i, but d_i, also applies to the above 'test on periodicity' as well as to Pearson's χ^2-test below: it does not test randomness of the ω_i but of the d_i. The tests may have to be repeated for several bin sizes $1/M$ (and may have different outcome).

- **Correlations:** Seemingly random numbers can have subtle correlations in, for instance, only the last few bits. A nice way of visualising that sometimes allows finding correlations is by plotting points (vectors) (ω_1, ω_2), (ω_2, ω_3), (ω_3, ω_4), $\ldots$ (with overlapping coordinates) in a plane, or similarly in 3D. Clustering or some structure hints at correlations in the numbers.

- **χ^2-test:** The *Pearson's chi-square test* measures the deviation v of the observed bin counts from the statistical expectation using n samples: the deviation is defined by

$$v \equiv \sum_{j=1}^{M} \frac{[y_j - E_j]^2}{E_j}, \tag{2.6}$$

  where y_j is the observed number of ω_i in bin j (as defined in (2.5)), E_j is the expectation for bin j (see §1.3.1) ($E_j = np$ if $p_j \equiv 1/M$ all j).
  When v is large, the observed values differ too much from the expected counts, so the samples ω_i were probably not uniform. When v is very close, or equal, to zero, (almost) all observed counts in each bin were correct, which is in line with a uniform distribution, but which is very unlikely for a random sequence.
  Note that, by continuing the sequence $\omega_1, \ldots, \omega_n$ to, say, $\omega_1, \ldots, \omega_n, \omega_{n+1}, \ldots, \omega_{kn}$ we can obtain k (hopefully) independent values $v_1, \ldots, v_k$ of v. We also can obtain k independent values of v by restarting $(k-1)$-times with a new random initial ω_1 as soon as we have generated $\omega_1, \ldots, \omega_n$ (how do you make sure that the k initial ω_1s are independently random?).
  The next subsection, §2.5.1, will give more information on the χ^2-test and its implementation.

- **Higher dimensional χ^2-tests:** suppose we use the sequence of ω_i to produce a sequence $d_1, \ldots, d_{n+2}$ of decimal digits d_i as explained above under the heading 'Uniformity'. Now we may ask ourselves how often does the string $d_1 d_2 \ldots d_{n+2}$ contain, say, the substring 000 of three decimal digits and what should we expect if the production of the digits d_i would truly random with uniform distribution? (Or similarly, when producing randomly n letters of the alphabet, how often do we find the substring 'ape': what is the probability that a monkey types 'ape'?) We are actually asking for the randomness and distribution of the sequence $d_1d_2d_3$, $d_2d_3d_4$, $d_3d_4d_5$, $\ldots$ of 'vectors' (with 'overlapping' coordinates). As before make bins 000, 001, 002, $\ldots$, 999 corresponding to $j = 0, 1, 2, \ldots, 999$ and count the number of occurrences in bin j of $d_id_{i+1}d_{i+2}$ in the vector sequence of length n (requiring $n + 2$ ω_i). Let ${}^3y_j$ be the count for bin j. If the distribution is uniform, all bins have equal probability to be 'visited'. Hence, the expectation ${}^3E_j$ for bin j equals $n/1000$ (100 if $n = 10^5$). Unfortunately, we can not simply copy formula (2.8)

$${}^3v \equiv \sum_{j=0}^{999} \frac{[{}^3y_j - {}^3E_j]^2}{{}^3E_j}, \tag{2.7}$$

  and apply ${}^3v$ to a χ^2-test: the dependency between ${}^3Y_j$ and ${}^3Y_k$ (where, random variables are denoted by capitals, corresponding realisations by lower case letters; ${}^3y_j$ corresponds to ${}^3Y_j$, etc.) is even more complicated than in case of one digit (as in (2.8), see §1.3.1). If $d_i d_{i+1} d_{i+2}$ is, say 000, then $d_{i+1} d_{i+2} d_{i+3}$ can only hit one of the bins 0, 1,..., 9 and no other one. However, if we correct for a two digit count,

$$v \equiv {}^3v - {}^2v \quad \text{with} \quad {}^2v \equiv \sum_{j=0}^{99} \frac{[{}^2y_j - {}^2E_j]^2}{{}^2E_j}, \tag{2.8}$$

then the χ^2-test is applicable: for large n, the random variable V $(= {}^3V - {}^2V)$ approximately has a χ^2-distribution for $900 (= 1000 - 100)$ degrees of freedom, if the d_i are truly from a random variable with uniform distribution (over $\{0, 1, \ldots, 9\}$). For $n = 10^5$, ${}^3E_j = 10^2$ and ${}^2E_j = 10^3$.

Exercise 2.11: *Higher dimensional χ^2-tests.*
We continue the discussion on 'Higher dimensional χ^2-tests'. Consider the sequence $d_1d_2d_3$, $d_4d_5d_6$, $d_7d_8d_9$, ... of vectors with 'non-overlapping' coordinates. To what quantity (relevant for this sequence) would one apply a χ^2 test? What are the number of freedoms? For testing purposes: what are the advantages of using sequences of vectors with overlapping coordinates? ■

2.5.1 A more quantitative χ^2-test

A 'χ^2-test' checks whether the available data is likely to be in line with data that would be obtained if the theoretical assumptions are fulfilled. The value for v, as defined above in 'χ^2-test', should not be too large, but also not too close to zero. In this section we will elaborate on this, and introduce the notion of a *confidence level.*

The test relies on χ^2 distributions. As we learnt in §1.3.1 from Pearson's theorem, for large n,

$$V \equiv \sum_{j=1}^{M} \frac{[Y_j - E_j]^2}{E_j}$$

will approximately have a χ^2_ν-distribution for $\nu \equiv M - 1$ (see (1.4)):

$$\chi^2_\nu(x) = \frac{1}{\kappa_\nu} x^{(\nu-2)/2} \exp\left(-\frac{x}{2}\right) \quad (x \in (0, \infty)) \quad \text{and} \quad \chi^2_\nu(x) = 0 \quad (x \in (-\infty, 0]).$$

Recall that the constant κ_ν scales χ^2_ν such that $\int_{-\infty}^{\infty} \chi^2_\nu(x)\, dx = 1$. Note that the value v of (2.8) is a realisation of V. Let F be the cumulative distribution associated with χ^2_ν:

$$F(x) \equiv \int_{-\infty}^{x} \chi^2_\nu(t)\, dt \qquad (x \in \mathbb{R}).$$

For a real number x, the value $F(x)$ gives the probability that a realisation v of V (a "measurement of V") is less than x, while $1 - F(x) = \int_x^\infty \chi^2_\nu(t)\, dt$ is the probability that v is larger than x.

We can set values for q_1 and q_2 and solve $F(x_1) = q_1$ for x_1 and $1 - F(x_2) = q_2$ for x_2. Rather than solving, these values x_1 and x_2 are usually taken from a (statistical) table. At the *confidence level* q_2 we then expect x_2 to be exceeded by v only $100 \cdot q_2$ percent of the cases that we generate our (hopefully) random numbers $\omega_1, \ldots, \omega_n$ (each time with different seed). Since we do not want $v \approx 0$ either, a similar quality assessment can be made at the left side: at the confidence level q_1 we expect $v < x_1$ only $100 \cdot q_1$ percent of the cases. We sometimes speak of a *confidence interval* $[x_1, x_2]$. Figure 2.3 shows a χ^2 distribution and the interval $[x_1, x_2]$ in which we would most likely expect values for v. If a value is outside of this interval, we have $1 - q$ 'confidence' that V is not χ^2_ν distributed, where $q \equiv q_1 + q_2$. Note that $1 - q = \int_{x_1}^{x_2} \chi^2_\nu(t)\, dt$.

As an alternative to determining the confidence interval $[x_1, x_2]$, the value $1 - F(v)$ can be computed. This is the so-called *p-value* of our 0-hypothesis, the hypothesis that the sequence of computed $\omega_1, \ldots, \omega_n$ is random from a uniform distribution. Recall that Pearson's χ^2-test requires n to be large. If this p-value is less than q_2, with say $q_2 = 0.025$, or larger than 0.975 ($q_1 = 0.025$), then we do not believe (with 95%

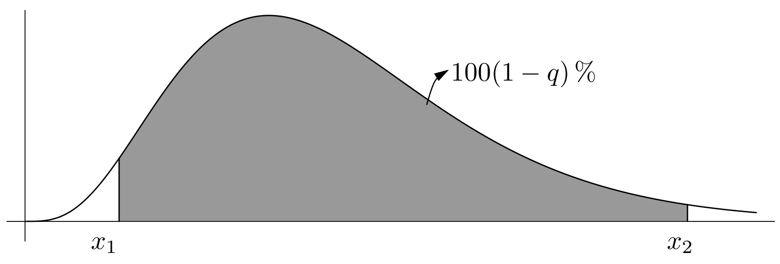

Figure 2.3: The χ^2 distribution with a confidence interval for V.

confidence) the hypothesis to be correct (the hypothesis is rejected).
Of course, it is a bit fishy to rely on only one measurement for V. Therefore, if you want 95 percent confidence ($q_1 = 0.025, q_2 = 0.025$), it may be better to generate independently a number of values for v and check if less than 2.5 percent of these values exceeds x_2, and similar for x_1. If this condition is failed, you have 95 percent confidence that your random number generator is not sufficiently random or not sufficiently uniform. The reverse (95 percent confidence that the RNG is random when the condition is met) need not be true (why not?).
Note that with many measurements for V you can also make an histogram that should approximate the appropriate χ^2-distribution. More reliable statements require more computational work.

★ **Exercise 2.12:** *Testing RNGs*.
Test the generators you wrote in Exercise 2.6 with the criteria of the previous section. Try various confidence levels for several generators. Also try some really bad generators for comparison. Do you prefer some RNG over another? Sample MATLAB code for visualisation can be found at the course web page [36]. Test also for efficiency!
The UNIX generator is the standard random number generator of the Unix operating system. It is known that the least significant bits have bad randomisation. Verify this.
(It might also be interesting to compare the behaviour of "your" random number generators to one that is generally being accepted as being excellent, cf., §2.6, e.g., §2.6.5 and the references in that subsection, and also to one that is clearly not random [but uniformly distributed], as the one in the next exercise, Exer. 2.13.) ■

Exercise 2.13: *The* van der Corput *sequence: a deterministic uniform distribution.*
The van der Corput sequence is uniformly distributed in $[0,1]$, yet it is not random.
This sequence is best defined using binary expansions of positive integers. Let π be the map from $\mathbb{N}$ to $[0,1]$ defined by

$$\pi\left(\sum_{i=0}^{n-1} b_i\,2^i\right) \equiv \sum_{i=0}^{n-1} b_i\,2^{-i-1} \qquad (n \in \mathbb{N}, b_i \in \{0,1\})$$

or, in binary representation $\pi(b_{n-1}\dots b_1 b_0) = 0.b_0 b_1 \dots b_{n-1}$. Then $(\pi(1), \pi(2), \pi(3), \dots)$ is the so-called *van der Corput sequence*.
Draw the first 15 points of the van der Corput sequence in the interval $[0,1]$ in order to visualise how a uniform distribution is achieved. Can this sequence pass your tests for randomness with a uniform distribution? (This sequence will be used in §3.3. A hint for a code to generate this sequence is included in the material for that section, `discrep.cc`; restrict to the case of only one prime number p, $p = 2$; `_dim=1;`.) ■

2.5.2 State-of-the-art RNG tests

Randomness is a vague concept, and various applications require numbers to be 'random' in different senses. Even, for an RNG that succeeds a range of tests, it is always possible to design a new test that this

RNG will fail.

George Marsaglia has collected a range of sophisticated tests in the Diehard package [19, 2]. The testsuite reads in large files that your RNG has produced, and performs selected tests on them.

☞ **Exercise 2.14:** *The Diehard testsuite for RNGs.*
Study some literature on the Diehard testsuite and decide which tests make sense to use in our upcoming application of Monte Carlo integration. Subject the various RNGs you implemented to the selected tests.
■

2.6 More general random number generators

More general (and more powerful) random number generators have been developed than the ones that we saw up to here. We will not implement them in this course, but for completeness and for the interested reader, we discuss a few of the ideas that have been pursued and that lead to (very long) sequences of ω_i in $[0, 1]$ that perform well in statistical tests that assume randomness with uniform distribution in $[0, 1]$. Obviously, more demanding tests (requiring longer sequences of random numbers) require generators with longer period. A glance on the generator '$x_{i+1} = x_i + 1 \bmod m$' reveals that the converse is generally not true. However, in all the generators that we mention below, the converse is true: the longer the period of the generators, the better its sequences of ω_i that it produces passes more demanding tests. Recall that in the generators of §2.2 the period is at most m and, therefore, for the ones that we considered, at most 2^{32}.

The generators in the next two subsections are straight-forward modifications of the linear congruential random number generators (LCRNG) of §2.2, though they are not of the form $x_{i+1} = f(x_i)$. Obviously, if we use 64-bits integer arithmetic (`unsigned long long int`), then the period can be increased (cf., §2.6.1). Averaging the outcome of our familiar type of generators for 32-bit arithmetic may also increase the period (cf., §2.6.2). Then we discuss two less obvious generalisations of LCRNG. Finally, in §2.6.5, we discuss a class of generators that rely on completely different ideas.

Pure mathematicians like to point out that from the decimal expansion of real numbers as π and e, sequences $\omega_0, \omega_1, \omega_2, \ldots$ can be extracted that will pass the tests for randomness with uniform distribution (for instance, if $\pi = d_0.d_1d_2d_3\ldots$ with $d_i \in \{0, 1, \ldots, 9\}$ ($d_0 = 3$, $d_1 = 1, \ldots$), and $m \in \mathbb{N}$ (the number of decimals required for the ω_i) and j is a randomly selected positive integer (to determine the seed), then, with $k \equiv j + im$, $\omega_i \equiv 0.d_{k+1}d_{k+2}\ldots d_{k+m}$ will do the job). Unfortunately, though reproducible, the decimal expansion (from the jth decimal on) cannot efficiently be generated.

2.6.1 Linear congruential generators

The basic generator (see [26]) in the NAG library (Numerical Analysis Group) [27] is a multiplicative linear congruential one,

$$x_{i+1} = a\,x_i \bmod m, \qquad \omega_{i+1} = \frac{x_i}{m},$$

with $a = 13^{13}$ and $m = 2^{59}$. It reaches a period of $\approx 2^{57}$. Implementation needs some care since it does not match a 32-bit integer computation.

$(11\,600, 2\,147\,483\,579), (47\,003, 2\,147\,483\,543), (23\,000, 2\,147\,483\,423), (33\,000, 2\,147\,483\,123)$

Table 2.2: Four pairs (a_j, m_j) for one set of parameters for the Wichmann and Hill combination (as discussed in §2.6.2) of LCRNGs to be used with Schrage's trick. Note that the m-values are the same except for the last three decimal digits.

2.6.2 Combining linear congruential generators

A combination of four multiplicative LCRNG (by Wichmann and Hill in 1989, see [18]) appears to lead to sequences of ω_i in $[0, 1]$ with high period, while allowing calculations with 32-bit integers:

$$\begin{aligned} v_{i+1} &= a_1\, v_i \bmod m_1, \quad x_{i+1} = a_2\, x_i \bmod m_2, \\ y_{i+1} &= a_3\, y_i \bmod m_3, \quad z_{i+1} = a_4\, z_i \bmod m_4 \\ \omega_{i+1} &= \left(\frac{v_{i+1}}{m_1} + \frac{x_{i+1}}{m_2} + \frac{y_{i+1}}{m_3} + \frac{z_{i+1}}{m_4}\right) \bmod 1 \end{aligned} \tag{2.9}$$

You may hope that by 'averaging' four LCRNGs, it is possible to create a random number generator with a period that is close to the product of the periods of the four individual LCRNGs, that is, in 32-bit integer arithmetic, close to $\approx (2^{32})^4 = 2^{128}$. The NAG library offers 273 sets of $(a_1, m_1, a_2, m_2, a_3, m_3, a_4, m_4)$ of pairs (a_j, m_j) that generate sequence each with a period of at least 2^{80}. The generators for different sets are essentially independent, which makes them attractive, for instance, for generating sequences of random high dimensional vectors (for each coordinate another generator can be used). The idea here is that if 2^{80} 'random numbers' are not enough, take other generators to generate other sequences of random numbers. The numbers a_i and m_i that NAG uses are small enough to allow straight-forward 32-bit integer calculations. The a_j are in the range $[112, 127]$ while the m_j are primes relatively close to $2^{24} = 16\,777\,216$.
Appropriate larger values of a_j and m_j, as in Table 2.2, have also been identified (by Wichmann and Hill in [38]) that allow 32-bit integer calculation when combined with Schrage's trick and that lead to a high period of $\approx 2^{121}$. Note that this period is very close to the trivial upper bound of 2^{128} on the period of the generator that can be obtained by averaging four LCRNGs each with period $\leq 2^{32}$.

2.6.3 Two dimensional more term linear congruential generators

More complicated multiplicative linear congruential generators, as two-dimensional four term generators have been investigated (by l'Ecuyer and Simard, cf., [17]), as well:

$$\begin{aligned} x_{i+1} &= (a_{11}\, x_i + a_{12}\, x_{i-1} + a_{13}\, x_{i-2}) \bmod m_1 \\ y_{i+1} &= (a_{21}\, y_i + a_{22}\, y_{i-1} + a_{23} y_{i-2}) \bmod m_2 \\ \omega_{i+1} &= \frac{1}{m_1}\left((x_{i+1} - y_{i+1}) \bmod m_1\right) \end{aligned}$$

The values for the a_{ij} and m_j that the NAG library provides allow 32-bit integer calculation and lead to a period of $\approx 2^{191}$.

2.6.4 Linear congruential generators with variable carry

Marsaglia (see [20] and its references) proposed the so-called *multiply-with-carry* (MWC) variant of LCRNG, where the carry c in the LCRNG (2.1) is subject to the iteration as well: for given a and m, and *seeds* x_0 and c_0, generate x_i, c_i, and ω_i as

$$y_i = a\, x_i + c_i, \quad x_{i+1} = y_i \bmod m, \quad c_{i+1} = y_i \operatorname{div} m, \quad \omega_{i+1} = \frac{x_{i+1}}{m} \quad (i = 0, 1, 2, \ldots). \tag{2.10}$$

It is easily verified that the y_i satisfy $y_{i+1} = a\,y_i \bmod (am-1)$ (see also Exer. 2.3). Therefore, while with fixed carry (i.e., $c_i = c$ all i), the period of the LCRNG is at most m, the period of this variant with variable carry is determined by $am-1$. The following result can be proved: if $p \equiv \frac{am}{2} - 1$ is a safe prime, that is, if p as well as $am-1$ is prime,[8] then the period is p. For instance, the pair $m = 2^{32}$ and $a = m-178$ forms a safe prime and leads to a period of $p \approx 2^{63}$. Note that with this value of 2^{32} for m, x_{i+1} are the lower 32 bits of y_i, while c_{i+1} are the upper 32 bits in case y_i has been computed in 64 bits (as a `unsigned long long int`); a, x_i, c_i can be represented as 32 bits integers. Since the c_i vary, Schrage's trick will not be applicable. (Can a MWC RNG with period of $\approx 2^{64}$ be efficiently realised in 32 bits integer arithmetic?)

By introducing a delay (or lag) in the use of x_i, the period can be much higher: for given a, m, and *delay* k, and now with *seeds* $x_0, \ldots, x_{k-1}$ and c_{k-1}, generate x_i, c_i, and ω_i as

$$y_{i-1} = a\,x_{i-k} + c_i, \quad x_i = y_{i-1} \bmod m, \quad c_i = y_{i-1} \operatorname{div} m, \quad \omega_i = \frac{x_i}{m} \qquad (i = k, k+1, \ldots). \quad (2.11)$$

With $k = 1$ we have the variant (2.10). If $p \equiv \frac{1}{2}am^k - 1$ is a safe prime, then the period is at least p. For number theoretical reasons, it seems to be easier to find primes of the form $am^k + 1$ rather than $am^k - 1$. These are the primes associated to the period of the so-called *complementary* MWC variant where x_i is computed as $x_i = m-1-[y_{i-1} \bmod m]$. Wikipedia gives a (four line) C-code for a complementary MWC RNG with period $\approx 2^{131\,104}$ ($m = 2^{32}$, $a = 18705$, $k = 2^{14}$).

2.6.5 Generalised feedback shift register

In this subsection we briefly discuss so-called *generalised feedback shift register* methods. The RNGs in this class of methods rely on ideas that are completely different from the ones used in LCRNG.

These methods generate sequences $b_0, b_1, b_2, \ldots$ of bits, that is, sequences of $b_i \in \{0,1\}$. A method is defined by a finite number of positive integers, called *taps*. For ease of explanation, consider the case of two taps, say $p, q \in \mathbb{N}$ with $p > q$. If a sequence $b_0, \ldots, b_{n-1}$ of n bits is available, then the method computes a new bit b_n as $b_n \equiv b_{n-p} \otimes b_{n-q}$ (this is the 'feedback'), where and $\otimes$ is a bit-wise 'exclusive or' operation[9]. Note that the method requires a sequence $b_0, \ldots, b_{p-1}$ of p bits as initialisation, for instance $b_i = 1 \ (i < p)$. The method could be modified to generate a sequence of integers in binary representation of constant length, say $m > p$: if $y_i \equiv a_{m-1} \ldots\ldots a_1 a_0$ is the ith integer with $a_i \in \{0,1\}$, then y_{i+1} can be formed by first extending y_i with a new bit using the feedback approach and then reducing to a length m binary integer by a '*register shift*', where each bit is shifted one position to the right: $y_{i+1} = [a_{m-p} \otimes a_{m-q}]a_{m-1} \ldots\ldots a_1$. Unfortunately, for interesting numbers p and q, the integers are too long (p will be larger than 32). However, sequences of integers $x_0, x_1, \ldots$ can also be generated in the same way as the sequences of bits are generated: if each x_i is represented by m binaries, then compute x_n as $x_n \equiv x_{n-p} \otimes x_{n-q}$, where now the $\otimes$ operator is applied bit wise to all m bits of x_{n-p} and x_{n-q}. To start, the sequence, $x_0, x_1, \ldots, x_p$ is required. In practice, these initial integers are formed from the bit sequence $b_0, \ldots, b_{mp}$ that is generated as explained above. The magic numbers $(p,q) = (250, 103)$ seem to work well.[10]

The *Mersenne twister generator* [23] that is available in the NAG library, combines register shifts, bit-wise 'exclusive or' and also bit-wise 'and' with a twisted type of feedback. It has a period of $\approx 2^{19\,937}$, it is highly efficient, and has been shown to lead to random numbers with uniform distribution in 623 dimensions. See [6] for a C++ implementation (mt19937) and [21] for a MATLAB version (mt19937ar).

[8] p is said to be a *safe* prime if p as well as $2p+1$ is prime.

[9] Defined, for each individual bit, by $0 \otimes 1 = 1$, $1 \otimes 0 = 1$, $1 \otimes 1 = 0$, $0 \otimes 0 = 0$

[10] PRNG, or, more specific, generalised feedback shift register methods, are used to determine the time difference between the broadcast of a signal from a GPS satellite and the arrival of the signal at a GPS receiver. This time difference times the speed of light equals the distance between satellite and receiver. Satellite and receiver continuously produce numbers, the same numbers at the same time with the same bit rate (bits per second). The satellite broadcasts these numbers at the given bit rate. The receiver is able to determine the time delay by comparing the bit string received from the satellite and the one produced by itself; the receiver delays his bit string until it exactly matches the bit string received from the satellite. The "randomness" of the consecutively produced numbers allows a sure match between the satellite numbers and the ones of the receiver (a sequence as 012301230123... would match for many delays). For the used bit rate, the PRNG that are used in GPS technology has a period of seven days.

Chapter 3

Monte Carlo Integration

This chapter discusses three numerical strategies for computing integrals. One of these strategies, *Monte Carlo integration*, uses statistical averages, i.e., it uses random samples and statistical statements to obtain a numerical value for the integral. We can put the RNGs we developed in the previous chapter to good use here.

All three strategies, *quadrature*, variants of Monte Carlo sampling, and *low-discrepancy sampling*, obtain approximate integrals by weigthed averages of function values:

$$\int_{\Omega} f(\mathbf{x})\, d\mathbf{x} \approx \sum_{\mathbf{i}\in E} w_{\mathbf{i}} f(\mathbf{x}_{\mathbf{i}}). \tag{3.1}$$

Here, $\Omega \subset \mathbb{R}^d$ is the integration domain, f is the integrand, a real-valued function on Ω, E is a finite subset of indices, $\mathbf{x}_{\mathbf{i}}$ are points in Ω, so-called *integration points*, and the scalars $w_{\mathbf{i}}$ (in $\mathbb{R}$) are weights. The methods that we consider differ in the way the points are selected at which the functions are to be evaluated. Basically, quatrature formulas use points on rectangular grids, thus, in higher dimensions, putting more points near the boundary of the integration domain than in the iterior. Monte Carlo selects points randomly (hence the naming). On average the set of these points is then more or less uniformly dense over the whole domain. However, Monte Carlo methods rely on large numbers to achieve this uniform spread of points. It is conceivable that a careful selection of the points can achieve a sufficiently dense and uniform coverage of the domain with less points. This is what low-discrepancy methods persue.

The first section below, §3.1, considers a conventional integration technique using quadrature formulas. It also introduces the general integration problem that will be handled by Monte Carlo in the subsequent section, §3.2. The discussion of low-discrepancy integration in §3.3 and the exercise on multi-dimensional integration, Exercise 3.8, is taken from [7].

3.1 Conventional integration: quadrature formulas

Why would we use Monte Carlo integration if we have a vast number of efficient quadrature schemes (like Simpson, Gauss, Romberg, etc.)? This is because quadrature schemes take time that is exponential with respect to the dimension of the integral. As a consequence, for higher-dimensional problems, quadrature (then called: *cubature*) can be very time consuming. In addition, the domains can be complicated and possibly non-convex then.

3.1.1 Quadrature schemes in one dimension

The general problem of integrating a univariate function is described as follows:

Let $f : [a, b] \to \mathbb{R}$ be a continuous function. Here, $a < b$. Find the solution of the definite integral:

$$\int_a^b f(x)\,dx. \tag{3.2}$$

The primitive function of f may be hard (or even impossible) to determine analytically. Therefore, we try to find a numerical approximation for the integral.

One of the simplest methods is to approximate f by a function p that is easily integrated. The *trapezoidal rule* uses a linear interpolation on a and b:

$$p(x) \equiv \frac{x-b}{a-b} f(a) + \frac{x-a}{b-a} f(b) \qquad (x \in [a, b]),$$

This leads to an approximation of the integral by:

$$\int_a^b f(x)\,dx \approx \int_a^b p(x)\,dx = \frac{b-a}{2}(f(a) + f(b)). \tag{3.3}$$

The error in this approximation equals the remainder

$$R \equiv \int_a^b f(x)\,dx - \frac{b-a}{2}(f(a) + f(b)). \tag{3.4}$$

If $f \in C^2$, i.e., is twice continuously differentiable, then, for some point $\eta \in [a, b]$, this remainder can be expressed as

$$R = \frac{1}{2}\int_a^b (x-a)(x-b) f''(\xi(x))\,dx = -\frac{(b-a)^3}{12} f''(\eta), \tag{3.5}$$

Here, for each $x \in [a, b]$, $\xi(x)$ is some point in $[a, b]$.

☞ **Exercise 3.1:** *The error of the trapezoidal rule.*
Derive the expressions for the remainder value in (3.5).
HINT: Use the fact that the integrand in (3.5) is the *error of a linear interpolation*. Use *Rolle's Theorem* to complete the derivation. To complete the proof, remember that f'' is continuous, and use the *Intermediate Value Theorem* to solve the last integral in (3.5). ■

The error (3.5) of the trapezoidal rule is usually too big. We can easily decrease it by dividing $[a, b]$ into k subintervals. We assume they all have equal size h, i.e., $kh = b - a$. Define the integration points $x_i \equiv a + ih$ $(i = 0, \ldots, k)$. On each subinterval, we apply the trapezoidal rule (3.3). The approximation for the integral (3.2) by the *repeated trapezoidal rule* then becomes:

$$T(h) \equiv h \cdot \left[\frac{1}{2} f(x_0) + \left[\sum_{i=1}^{k-1} f(x_i)\right] + \frac{1}{2} f(x_k)\right] \tag{3.6}$$

Figure 3.1 shows the repeated trapezoidal rule schematically. The overall error is obtained by summing all remainders:

$$R(h) \equiv -\frac{1}{12} h^3 \sum_{i=1}^{k} f''(\eta_i), \text{ with } \eta_i \in [x_{i-1}, x_i]. \tag{3.7}$$

Assuming that f is again a C^2 function, this error has an upper bound:

$$|R(h)| \le \frac{b-a}{12} h^2 \max_{x \in [a,b]} |f''(x)|, \tag{3.8}$$

or, using the fact that the expression $h \sum_{i=1}^{k} f''(\eta_i)$ represents a Riemann sum approximation of $\int_a^b f''(x)\,dx$, we also have that

$$R(h) = \frac{b-a}{12} h^2 \left[f'(b) - f'(a)\right] + \mathcal{O}(h^3) \quad \text{for} \quad h \to 0. \tag{3.9}$$

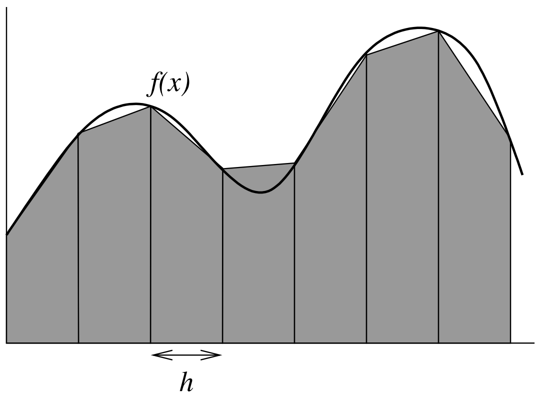

Figure 3.1: Schematic representation of the repeated trapezoidal rule for approximating the integral of a function f. All subintervals have equal size h. Notice the relatively large errors that are made by the linear approximation on some intervals (the ones where f'' is large).

☞ **Exercise 3.2:** *Error of the repeated trapezoidal rule.*
Derive the expression for the remainder value of the repeated trapezoidal rule in (3.7). Next give proof for the upper bound on the remainder value, as given in (3.8).
HINT: For the proof, again use the *Intermediate Value Theorem* and the fact that f'' *is continuous*.
Note that the expressions in both (3.8) and (3.9) show that the accuracy of the repeated trapezoidal is of order h^2. However, (3.9) gives more information: it shows that the accuracy is not of higher order (provided that $f'(b) \neq f'(a)$) and actually is proportional to h^2. ■

The expression in (3.9) shows that the accuracy of the repeated trapezoidal is proportional to h^2. Suppose we want to approximate an integral with a given accuracy. Automatic integration schemes use this proportionality of the error to h^2 to successively refine the discretisation (i.e. increase k, the number of points along an axis) until the requested accuracy is obtained.

Exercise 3.3: *The repeated trapezoidal rule in higher dimensions.*
Let f be a real-valued C^2 function on $\Omega \equiv [a_1, b_1] \times [a_2, b_2]$.
Derive an expression for the repeated trapezoidal rule for the computation of the 2-dimensional integral

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y)\,dy\,dx \qquad \left(= \int_{a_1}^{b_1} F(x)\,dx \quad \text{with} \quad F(x) \equiv \int_{a_2}^{b_2} f(x,y)\,dy \quad (x \in [a_1,b_1]) \right)$$

by first applying the one dimensional trapezoidal rule with stepsize $h_1 \equiv (b_1 - a_1)/k_1$, to compute $\int_{a_1}^{b_1} F(x)\,dx$ and then by applying the one dimensional trapezoidal rule with stepsize $h_2 = (b_2 - a_2)/k_2$ to to compute the $F(jh_1)$ for $j = 0, \ldots, k_1$.
Derive an expression for the associated error in terms of (h_1, h_2) and partial derivatives of f. Conclude that the error is of order $\mathcal{O}(h^2)$ for $h \equiv \max(h_1, h_2) \to 0$.
Adapt the above arguments and argue that the repeated trapezoidal rule for a d-dimensional integral for an C^2 function also leads to an error of order $\mathcal{O}(h^2)$ for $h \equiv \max_j h_j \to 0$. ■

Let n denote the total number of integration points needed to obtain an error less than, say, ε. In one dimension, $n = k \propto h^{-1}$. If in two dimensions we take k points in the x-direction, and for each of those also k points in the y-direction, then we need a total of $n = k^2 \propto h^{-2}$ points. In general, in a d-dimensional domain, the total number n of needed integration points is proportional to h^{-d}: $n \propto h^{-d}$. The error in the repeated trapezoidal rule is of order h^2 regardless the dimension d, cf., Exercise 3.3. Therefore, the total number of points to achieve an accuracy ε, i.e., an error $\leq \varepsilon$, is of order $\varepsilon^{-d/2}$:

$$n \propto \frac{1}{\varepsilon^{d/2}} \quad \text{and} \quad \text{error} \propto \frac{1}{\sqrt[d/2]{n}}. \tag{3.10}$$

In other words: an increased accuracy, say to an error less than $\frac{1}{2}\varepsilon$, requires $(\sqrt{2})^d$ as many discretisation points (for $d = 12$ a difference between 1 second of computational time versus 1 minute). This shows that the costs of this quadrature formula are exponential in d. Other quadrature rules that are obtained from simpler ones by repetition (as the repeated Simpson rule) are also exponential in d (the increase in the required number of points will be by a factor α^d for some $\alpha > 1$ rather than by $(\sqrt{2})^d$; α might be $< \sqrt{2}$). Exponential costs are about the worst that can happen to a computational scientist. Therefore quadrature schemes are often said to suffer from the *curse of dimensionality*.

Note that for higher dimensions, an application of the one dimensional formula to each of the coordinates (as in Exercise 3.3) leads to a high number of discretisations points already for small values of k_j. For instance, with $k = k_1 = k_2 = \ldots = k_d$, the number of points required, for $k = 1$, $k = 2$, $k = 3$, and $k = 4$, with the repeated trapezoidal rule on $[0,1]^{12}$ (i.e., $d = 12$) equals $2^{12} \approx 4\,000$, $3^{12} \approx 500\,000$, $4^{12} \approx 17\,000\,000$, and $5^{12} \approx 240\,000\,000$, respectively (and no other number of points in between).

Apart from the enormous costs, quadrature schemes can also become complicated in higher-dimensional spaces. In one dimension, the domain usually is an interval; in more dimensions, the domain can easily take any shape and integration is especially complicated when the domain is not convex.

When using a Monte Carlo simulation for approximating integrals instead, the error is of a statistical nature. As we will learn in §3.2.4 below, it decreases with the square root of the sample size, *independently of the dimensionality of the integral*:

$$\text{error} \propto \frac{1}{\sqrt[2]{n}} \quad \text{and} \quad n \propto \frac{1}{\varepsilon^2}. \tag{3.11}$$

This error decreases (much) faster for $d > 4$ than in case of the repeated trapezoidal rule (cf., (3.10)). This observation is the main justification for studying Monte Carlo integration.

3.2 Alternative integration: Monte Carlo

3.2.1 Hit-or-miss Monte Carlo

Perhaps the simplest version of Monte Carlo integration is 'hit-or-miss'.

Suppose that we want to integrate a function $f : [0,1] \to [0,1]$,

$$I(f) \equiv \int_0^1 f(x)\,dx.$$

The hit-or-miss algorithm generates a sequence $(x_1, y_1), (x_2, y_2), \ldots$ of random 'shots' (x_i, y_i) in the square $S \equiv [0,1] \times [0,1]$.[1] The number of shots out of n that 'hit' the area A in S below the graph of f,

$$A \equiv \{(x,y) \in S \mid y \leq f(x)\},$$

are counted:

$$N_A(n) \equiv \#\{i \leq n \mid (x_i, y_i) \in A\} = \#\{i \leq n \mid y_i \leq f(x_i)\}.$$

The fraction $N_A(n)/n$ of hits in the set of the first n shots is taken as an approximation of the size of the area A, and therefore of the integral $I(f)$. A more accurate approximation is expected for larger n.

We now use the terminology and results of Chapter 1 to formally justify this approach.
With respect to the standard product measure P, $\Omega \equiv S$ is a probability space; see Example 1.4. Consider the random variable B_A on Ω defined for $(\omega, \tilde{\omega}) \in \Omega$ by $B_A(\omega, \tilde{\omega}) = 1$ if $(\omega, \tilde{\omega}) \in A$ and $B_A(\omega, \tilde{\omega}) = 0$ else: $B_A(\omega, \tilde{\omega})$ equals 1 in case $(\omega, \tilde{\omega})$ 'hits' the area A and is 0 in case of a 'miss'. B_A is a Bernoulli

[1]That is, each subsquare of S of size $h \times h$ has equal probability of being 'hit' (namely, probability h^2), or, in other words, the x_i and y_i are independent and random in $[0,1]$ from a uniform distribution.

variable (see Example 1.9) with expectation value $E(B_A)$ equal to $P(A)$ (see Exercise 1.10.c) which is equal to $I(f)$: $E(B_A) = I(f)$. According to Theorem 1.7, the law of large numbers, the average

$$\overline{B}_n \equiv \frac{1}{n}\sum_{i=1}^{n} b_i$$

of realisations b_i of B_A converges to $E(B_A)$ with probability 1 ($n \to \infty$). Since realisations b_i of B_A are obtained as $b_i = B(\omega_i, \tilde{\omega}_i)$ with $(\omega_i, \tilde{\omega}_i)$ random samples from Ω, and 'shots' (x_i, y_i) are such random samples, we see that $N_A(n)/n$ is an average of realisations of B_A, whence

$$\frac{N_A(n)}{n} \to I(f) \quad \text{with probability 1} \quad (n \to \infty).$$

The expectation of the error $|\overline{B}_n - E(B_A)|$ can be bounded in terms of Var(B_A) (see (1.15)):

$$E(|\overline{B}_n - E(B_A)|) \leq \frac{1}{\sqrt{n}}\sqrt{\text{Var}(B_A)}.$$

Since Var$(B_A) = P(A)(1 - P(A))$ (see Exercise 1.10.c), $E(B_A) = P(A) = I(f)$, and $N_A(n)/n$ is a $\overline{B}_n$, this leads to the following statement on the convergence of 'hit-or-mis':

$$E\left(\left|\frac{N_A(n)}{n} - I(f)\right|\right) \leq \frac{1}{\sqrt{n}}\sqrt{\text{Var}(B_A)} = \sqrt{\frac{I(f)(1 - I(f))}{n}} \leq \frac{1}{2\sqrt{n}}. \tag{3.12}$$

Recall from the end of §1.4 that Var(B_A) can be estimated from realisations b_i of B_A, yielding

$$\text{Var}(B_A) \approx \frac{N_A(n)}{n}\left(1 - \frac{N_A(n)}{n}\right) \quad \text{and} \quad \text{Var}(B_A) \approx \frac{N_A(n)}{n-1}\left(1 - \frac{N_A(n)}{n}\right),$$

a biased estimator (use (1.24) and the fact that $b_i^2 = b_i$) and an unbiased estimator (use (1.25)), respectively.

3.2.2 The Monte Carlo method using simple sampling

In this subsection we devise an alternative Monte Carlo method to compute the following integral to some accuracy

$$\int_a^b f(x)dx. \tag{3.13}$$

If Q is a *random variable* which is uniformly distributed over $[a, b]$, then the *density function* q of Q is

$$q(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b, \\ 0 & \text{otherwise,} \end{cases} \tag{3.14}$$

and the expectation of the random variable $f(Q)$ is (see Theorem 1.3)

$$E(f(Q)) = \int_a^b f(x)\, q(x)\, dx = \int_a^b \frac{f(x)}{b-a}\, dx = \frac{1}{b-a}\int_a^b f(x)\, dx. \tag{3.15}$$

Which gives

$$\int_a^b f(x)dx = (b-a)\, E(f(Q)). \tag{3.16}$$

So if we have an *estimate* for $E(f(Q))$ we also have an estimate for the integral of f. According to the law of large numbers (cf., (1.23) in §1.4), we obtain such an estimate by *simple sampling*, where we average f over realisations $q_1, \ldots, q_n$ of Q:

$$E(f(Q)) \approx \overline{f(Q)}_n \equiv \frac{1}{n}\sum_{i=1}^{n} f(q_i). \tag{3.17}$$

This leads to the following approximation of the integral

$$\int_a^b f(x)\,dx \approx \frac{1}{n}(b-a)\sum_{i=1}^{n} f(q_i) = (b-a)\overline{f(Q)}_n. \tag{3.18}$$

Here, the ω_i are randomly sampled from $[0,1]$ with uniform distribution and

$$q_i \equiv Q(\omega_i) = \omega_i(b-a)+a \qquad (i=1,\ldots,n). \tag{3.19}$$

As in (1.23), we added the subscript n in $\overline{f(Q)}_n$ to indicate the number of samples taken to calculate the average.

Theorem 3.1 (The error of the Monte Carlo method)
The error of the simple sampling Monte Carlo approximation to an integral is estimated to decrease with $\mathcal{O}(n^{-\frac{1}{2}})$, where n is the number of samples used. ♦

Proof. According to (1.15), we have

$$E\left(\left|\overline{f(Q)}_n - E(f(Q))\right|\right) \le \sqrt{\frac{1}{n}\operatorname{Var}(f(Q))}. \tag{3.20}$$

$\operatorname{Var}(f(Q))$ is unknown, but nonetheless we can conclude that the error (probably) decreases as $n^{-\frac{1}{2}}$. □

Though the variance of $f(Q)$ is unknown it can be estimated from the $f(q_i)$: as explained at the end of §1.4, the following expressions are both estimators for $\operatorname{Var}(f(Q))$.

$$\left(\frac{1}{n}\sum_{i=1}^{n} f(q_i)^2\right) - \left(\overline{f(Q)}_n\right)^2 \quad \text{or} \quad \frac{1}{n-1}\sum_{i=1}^{n}[f(q_i) - \overline{f(Q)}_n]^2. \tag{3.21}$$

Note the following features of Monte Carlo approximation

- The error is probabilistic but
- the error does not depend on the dimensionality of the integral (to be shown below in §3.2.4).
- The error does not depend on the smoothness of the function f.[2]

3.2.3 Importance sampling

In the previous method we chose Q to have a uniform distrubition q in the interval $[a,b]$. This can give poor results if our function f is 'concentrated' in a small part of the interval of integration. We can improve our efficiency by *importance sampling*, that is, sampling from another distribution function q chosen to be a reasonable approximation of the function we want to integrate and observing that

$$\int_a^b f(x)\,dx = \int_a^b g(x)q(x)\,dx \quad \text{with} \quad g(x) \equiv \frac{f(x)}{q(x)} \quad (x \in [a,b]). \tag{3.22}$$

See also equation (1.9) for the definition of the expectation. We draw points distributed according with q and compute the average of $g(x)$ at these points.

You may use techniques from Section 2.4 to obtain the non-uniformly distributed variables. This technique is useful if F^{-1} is readily available, enabling us to use the inversion method. Here, $F(x) \equiv \int_{-\infty}^{x} q(y)\,dy$. When we need to resort to the rejection method the computational cost becomes prohibitive and there is no advantage to using importance sampling.

[2]Since, our samples will not be truly randomly drawn, but are (pseudo) random drawn from a subset of rational numbers, some (local) smoothness of f is required in practice.

★ **Exercise 3.4:** *Integrating a function with Monte Carlo.*
Compute the integral

$$\int_0^1 \sqrt{1-x^2}\,dx,$$

first using 'hit-or-miss' Monte Carlo. Secondly do the computation using 'simple sampling' Monte Carlo, i.e., by calculating

$$E(\sqrt{1-Q^2}) \approx \frac{1}{n}\sum_{i=1}^{n}\sqrt{1-q_i^2},$$

with $Q = U_{[0,1]}$, $q_i = \omega_i$. Plot the error (you know the analytical value of the integral!) versus the sample size for both methods. Do you observe the expected convergence (see (3.12) and (3.20))? ■

3.2.4 Multi-dimensional Monte Carlo integration

In most instances, it is possible to generalise the Monte Carlo procedure for (approximately) computing the integral (3.1) in multiple dimensions, i.e., $d > 1$. Hit-or-miss Monte Carlo of §3.2.1 is particularly simple. Suppose you need to integrate $f : \Omega \to [0,1]$ over a domain Ω subset of, say, $[0,1]^d$. In each step you generate realisations $x_1, \dots, x_{d+1}$ of $d+1$ random variables $X_1, \dots, X_{d+1}$, all with uniform distribution on $[0,1]$, treat the first d variables as coordinates of a point $\mathbf{x} \equiv (x_1, \dots, x_d)$ in $[0,1]^d$ and test whether $x_{d+1} < f(\mathbf{x})$. Here, we define f to be zero on points outside Ω.

☞ **Exercise 3.5:** *Multi-dimensional hit-or-miss.*
Formulate the multi-dimensional hit-or-miss procedure. Why does this approximate $\int_\Omega f(\mathbf{x})\,d\mathbf{x}$? Can you derive the convergence speed in terms of the number of random variables needed? Does this depend on the dimension d? ■

In case the integration domain Ω is an *(hyper) rectangle* $\mathbf{J} \equiv [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ in $\mathbb{R}^d$ the Monte Carlo method of §3.2.2 using simple sampling for one dimension generalises to

$$\int_{\mathbf{J}} f(\mathbf{x})\,d\mathbf{x} \approx \frac{1}{n}(b_1 - a_1)\cdot(b_2 - a_2)\cdot \dots \cdot (b_d - a_d)\sum_{i=1}^{n} f(\mathbf{q}_i), \tag{3.23}$$

where the $\mathbf{q}_i$ are randomly selected from $\mathbf{J}$ with a uniform distribution: with a sequence $(\omega_1, \omega_2, \dots, \omega_d)$ of d (random) samples ω_i in $[0,1]$ (from a uniform distribution on $[0,1]$), we have a randomly selected point $\mathbf{q}$ in $\mathbf{J}$ from a uniform distribution when taking $\mathbf{q} \equiv (\omega_1(b_1 - a_1) + a_1, \dots, \omega_d(b_d - a_d) + a_d)$.

Obviously, this method with simple samples can also be also be used if the integration domain Ω in $\mathbb{R}^d$ is more complicated but contained in an rectangle $\mathbf{J}$, $\Omega \subset \mathbf{J}$: simply define f to be zero in points of $\mathbf{J}$ outside Ω and integrate over $\mathbf{J}$. However, points outside Ω do not give information on f: they give information on Ω only. Therefore, if f varies significantly on Ω, this approach (as well as hit-or-miss) might be not be optimal efficient. The idea of 'stretching' Ω to match $\mathbf{J}$ exactly may lead to better results. The approach below of linearly 'stretching' along the axis can be viewed as a realisation of this idea. The effect of stretching is reflected in the fact that the weigths w_i depend on the position of $\mathbf{q}_i$ in Ω; see (3.25) below.

To derive the appropriate values for the weights, we assume that the integration can be carried out by means of repeated integration (i.e. the d-dimensional integral can be written as d nested 1-dimensional integrals). With $\mathbf{x} = (x_1, \dots, x_d)$, the integration may then be written as

$$\int_\Omega f(\mathbf{x})\,d\mathbf{x} = \int_{a_1}^{b_1}\int_{a_2(x_1)}^{b_2(x_1)} \cdots \int_{a_d(x_1,\dots,x_{d-1})}^{b_d(x_1,\dots,x_{d-1})} f(x_1, \dots, x_d)\;dx_d\,dx_{d-1}\dots dx_1.$$

In principle, the Monte Carlo method may now be applied to each one dimensional integral, starting at the inner integral and continuing up to the outer one. In the first step we would consider

$$x_1 \rightsquigarrow \int_{a_2(x_1)}^{b_2(x_1)} \cdots \int_{a_d(x_1,\dots,x_{d-1})}^{b_d(x_1,\dots,x_{d-1})} f(x_1, \dots, x_d)\;dx_d\,dx_{d-1}\dots dx_2,$$

a function of x_1 only and apply the Monte Carlo approximation of (3.18),

$$\int_\Omega f(\mathbf{x})\,d\mathbf{x} \approx \frac{1}{k}\sum_{i_1=1}^{k}[b_1-a_1] \int_{a_2(x_1^{(i_1)})}^{b_2(x_1^{(i_1)})} \cdots \int_{a_d(x_1^{(i_1)},x_2\ldots,x_{d-1})}^{b_1(x_1^{(i_1)},x_2,\ldots,x_{d-1})} f(x_1^{(i_d)},x_2,\ldots,x_d)\;dx_d\,dx_{d-1}\ldots dx_2$$

where k is the number of random points $x_1^{(1)},\ldots,x_1^{(k)}$ taken from $[a_1,b_1]$: $x_1^{(i_1)}$ is the i_1th realisation of the random variable X_1 with uniform distribution on $[a_1,b_1]$. Now, what's inside the remaining outer integral is, for each $x_1^{(i_1)}$, a function of x_2 on the interval $[a_2(x_1^{(i_1)}),b_2(x_1^{(i_1)})]$ and we can again apply Monte Carlo of (3.18), again taking k points $x_2^{(1)},\ldots,x_2^{(k)}$ now in $[a_2(x_1^{(i_1)}),b_2(x_1^{(i_1)})]$:

$$\int_\Omega f(\mathbf{x})\,d\mathbf{x} \approx \frac{1}{k^2}\sum_{i_1=1}^{k}\sum_{i_2=1}^{k}[b_1-a_1]\cdot[(b_2(x_1^{(i_1)})-a_1(x_1^{(i_1)})] \int_{a_3(x_1^{(i_1)},x_2^{(i_2)})}^{b_3(x_1^{(i_1)},x_2^{(i_2)})} \cdots \int_{a_d(x_1^{(i_1)},x_2^{(i_2)},x_3,\ldots,x_{d-1})}^{b_d(x_1^{(i_1)},x_2^{(i_2)},x_3,\ldots,x_{d-1})} f(x_1^{(i_1)},x_2^{(i_2)},x_3,\ldots,x_d)\;dx_d\,dx_{d-1}\ldots dx_3.$$

Continuing in this fashion we end up with the generalisation (3.25) below of (3.18). First, to simplify notation, let us define the 'unscaled weight' $w(\mathbf{x})$ associated to a point $\mathbf{x}=(x_1,\ldots,x_d)\in\Omega$:

$$w(\mathbf{x}) \equiv [b_1-a_1]\left(\prod_{j=2}^{d}[b_j(x_1,\ldots,x_{j-1})-a_j(x_1,\ldots,x_{j-1})]\right) \tag{3.24}$$

Then, with multiple index $\mathbf{i}\equiv(i_1,\ldots,i_d)$ and $\mathbf{x_i}\equiv(x_1^{(i_1)},\ldots,x_d^{(i_d)})$, we obtain

$$\int_\Omega f(\mathbf{x})\,d\mathbf{x} \approx \frac{1}{|\mathcal{I}|}\sum_{\mathbf{x}\in\mathcal{I}} w(\mathbf{x})\,f(\mathbf{x}). \tag{3.25}$$

Here, $\mathcal{I}$ is the (finite) set of selected points $\mathbf{x_i}$ in Ω, $|\mathcal{I}|$ is the number of points in $\mathcal{I}$, and we sum over all $\mathbf{x}$ in $\mathcal{I}$. The derivation leads to $\mathcal{I}=\{\mathbf{x_i}\mid\mathbf{i}=(i_1,\ldots,i_d),i_j=1,\ldots,k\}$, $|\mathcal{I}|=k^d$ and the sum is actually a repetition of d sommations. But the pair of formulas (3.24) and (3.25) is more generally correct. Scaling $w(\mathbf{x})$ gives the *weight* $w(\mathbf{x})/|\mathcal{I}|$.

There is now one final consideration: the distribution of the random variable $\mathbf{X}\equiv(X_1,\ldots,X_d)$ of which the $\mathbf{x}$ in (3.25) are realisations. The domain Ω can be complicated and the boundaries come into play. The uniformity of the distribution is according to the boundaries in the integration, thus

$$\begin{array}{rcl} X_1 & \propto & U_{[a_1,b_1]} \\ X_2 & \propto & U_{[a_2(X_1),b_1(X_2)]} \\ X_3 & \propto & U_{[a_3(X_1,X_2),b_3(X_1,X_2)]} \\ & \vdots & \\ X_d & \propto & U_{[a_d(X_1,\ldots,X_{d-1}),b_d(X_1,\ldots,X_{d-1}]}. \end{array}$$

Here, $U_{[a,b]}$ is as defined in Example 1.8.

To obtain a realisation $\mathbf{x}=(x_1,\ldots,x_d)$ of the $\mathbf{X}$ and the associated unscaled weight $w(\mathbf{x})$, it is easiest to generate $\omega_1, \ldots, \omega_d$ from $U_{[0,1]}$ and then set, in the given order (cf., (3.19)),

$$\begin{aligned} \ell_1 &\equiv b_1 - a_1, & x_1 &\equiv \omega_1 \ell_1 + a_1 \\ \ell_2 &\equiv b_2(x_1) - a_2(x_1), & x_2 &\equiv \omega_2 \ell_2 + a_2(x_1) \\ &\vdots & &\vdots \\ \ell_d &\equiv b_d(x_1, x_2, \ldots, x_{d-1}) - a_d(x_1, x_2, \ldots, x_{d-1}), & x_d &\equiv \omega_d \ell_d + a_d(x_1, \ldots, x_{d-1}). \end{aligned} \tag{3.26}$$

Note that $w(\mathbf{x}) = \ell_1 \ell_2 \ldots \ell_d$. For some specific boundary functions a_i and b_i you may be able to work out the above relations and insert them in formula (3.25). Note that the realisations $\mathbf{x}$ of $\mathbf{X}$ are random and, in contrast to integrations points in the higher dimensional repeated trapezoidal rule, do not lie on a rectangular grid.

Exercise 3.6: *Monte Carlo on a rectangle.*
Check that Formula (3.25) simplifies to (3.23) in case Ω is the rectangle $\mathbf{J} = [a_1, b_1] \times \ldots \times [a_d, b_d]$. Note that, in this case, the points $\mathbf{x}$ of (3.25) are according to a uniform distribution (in what sense?). ■

The derivation of (3.25) looks a little troubling: it led to a formula with costs k^d function evaluations and it seems that we have no gain over 'traditional' methods. However, looks deceive since we shall see in the proof of the following theorem that the error decreases as $k^{-d/2}$. It is therefore still in the order of one over the square root of the number of function evaluations, independent of the dimension, as in Theorem 3.1.

Theorem 3.2 (The error of multi-dimensional Monte Carlo)
The expectation of the error of multi-dimensional Monte Carlo of (3.24) and (3.25) is independent of dimension d, and scales as $\mathcal{O}(n^{-\frac{1}{2}})$. Here $n = |\mathcal{I}|$ is the total number of points used. ♦

Proof. We discuss the proof for the case where $\Omega = [0,1]^d$ (cf., Exercise 3.6) and follow the repeated 1-dimensional approach. In particular $n = k^d$.
As in the one dimensional method we need to work out the variance of the sample mean of f, the mean absolute error is then the square root. See also Theorem 3.1 for the rationale behind this procedure. Let us work out the error expression,

$$\begin{aligned} e_k \equiv \sqrt{\operatorname{Var}(\overline{f_k(X_1, \ldots, X_d)})} &= \sqrt{\operatorname{Var}\left(k^{-d} \sum_{i_1, \ldots, i_d = 1}^{k} f(X_1^{(i_1)}, \ldots, X_d^{(i_d)})\right)} \\ &= \sqrt{(k^{-d})^2 \operatorname{Var}\left(\sum_{i_1, \ldots, i_d = 1}^{k} f(X_1^{(i_1)}, \ldots, X_d^{(i_d)})\right)} \\ &= k^{-d} \sqrt{\operatorname{Var} \sum_{i_1, \ldots, i_d = 1}^{k} f(X_1^{(i_1)}, \ldots, X_d^{(i_d)})}, \end{aligned} \tag{3.27}$$

where, for each direction $j \in \{1, \ldots, d\}$, $X_j^{(i)}$ is the random variable with identical but independent distribution as X_j; $x_j^{(i)}$ is a realisation of X_j $(i = 1, \ldots, k)$. We use this and the rules for working with variances (see Ex. 1.9.c) to obtain from (3.27) that

$$\begin{aligned} e_k &= k^{-d} \sqrt{k^d \operatorname{Var} f(X_1, \ldots, X_d)} \\ &= k^{-d} \cdot k^{d/2} \sqrt{\operatorname{Var} f(X_1, \ldots, X_d)} = \mathcal{O}(k^{-d/2}) \qquad (k \to \infty). \end{aligned} \tag{3.28}$$

Recall that we did not use k points to approximate the integral, but k for each of the one-dimensional integrals in the nesting of integrals, i.e., $n \equiv k^d$ in total. Combining this with (3.25) yields the $\mathcal{O}(1/\sqrt{n})$. □

3.3 Sampling for low discrepancy

The methods that we consider in this section, §3.3, proceed as the Monto Carlo integration methods, but rather than selecting *random* samples from, say, $[0,1]^d$ according to the uniform distribution (cf., the introduction of §3.2.4), they select a sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots$ of points $\mathbf{x}_j$ from this (hyper) cube in a predetermined order and use the jth point where Monto Carlo would use the jth random sample.[3] In this setting, the error, the difference between the value that is obtained in this way using the first n points and the actual value of the integral, is called the *discrepancy*. The methods, the so-called *low discrepancy methods*, aim for selecting a sequence of points $\mathbf{x}_j$ that 'minimise' the discrepancy. For an exact formulation and for analysis purposes the *discrepancy* D_n is formally defined as the largest error in the volumes of rectangles in $[0,1]^d$ using the first n points.[4] Note that, discrepancy here actually focuses on integrating characteristic functions of rectangles, while the classical error analysis assumes smoothness of the integrand. Nevertheless, the approach is in line with Riemann integrability where an integral is defined as the limit of Riemann sums and each Riemann sum is the integral of a step functions, functions that are finite linear combinations of characteristics of rectangles. Summarising, we are looking for sequences of points $\mathbf{x}_1, \mathbf{x}_2, \ldots$ in $[0,1]^d$, *low discrepancy sequences*, with a distribution that is as uniform as possible. We do not worry about randomness. For ease of discussion, we focus on the cube $[0,1]^d$, but, of course, with appropriate scaling, we easily obtain points for any rectangle.

Exercise 3.7: *The discrepancy for the trapezoidal rule.*
As an example, consider the integration domain $\Omega \equiv [0,1]^d$ and a cube $\mathbf{J} = \mathbf{x} + [0,\beta]^d$ with $\beta \in (0,1)$ and $\mathbf{x} \in \Omega$ such that $\mathbf{J} \subset (0,1)^d$. Show that the error in the trapezoidal rule to integrate $\chi_{\mathbf{J}}$ with $h = 1$ and with $h = \frac{1}{2}$ equals β^d and $\beta^d - \frac{1}{2^d}$, respectively. For $\beta \approx 1$ and large d, say $d = 12$, the error is ≈ 1 even though approximately 4 000, and 500 000, respectively, integration points are required. What is the expected error with Monte Carlo integration when these many points are being used (but now randomly selected)? ■

It turns out that coefficients of prime numbers p expansions do very well for defining sample points for low-discrepancy methods.

We first explain how to write a non-negative integer i in base p. Write i as

$$i = \sum_{j=0}^{m} a_j\, p^j,$$

with a_j and m appropriate non-negative integers such that a_j in $\{0, 1, \ldots, p-1\}$. The representation of i in the p-ary number system is then

$$i = a_m \ldots a_1 a_0$$

You are already familiar with the binary system where $p = 2$. The treatment of real numbers is analogous to the way we write for example $0.321 = 3 \cdot 10^{-1} + 2 \cdot 10^{-2} + 1 \cdot 10^{-3}$ in decimal system: for a real number $x \in [0,1)$ we write

$$x = \sum_{j=1}^{\infty} a_j\, p^{-j}.$$

Again for appropriate $a_j \in \{0, 1, \ldots, p-1\}$. The representation of the number x in the p-ary number system is preceded by a dot and (sometimes) a zero:

$$x = 0.a_1 a_2 \ldots$$

[3]Of course the interval $[0,1]$ (and the corresponding coordinate of $\mathbf{x}_j = (x_{j,1}, \ldots, x_{j,d})$) has to be scaled and shifted when values in, say, $[a,b]$ are required.

[4]Let $R_n(\mathbf{J})$ denote the error in the volume of the rectangle $\mathbf{J} = [a_1,b_1] \times [a_2,b_2] \times \ldots \times [a_d,b_d]$, that is, the difference between $\Pi_{i=1}^d (b_i - a_i)$ and $\#\{j \leq n \mid \mathbf{x}_j \in \mathbf{J}\}/n$. The discrepancy D_n is the supremum of the $R_n(\mathbf{J})$, where the supremum is taken over all rectangles $\mathbf{J} \subset [0,1]^d$.

The system for positive integers i and reals $x \in [0,1)$ can be combined to represent any real number in the p-ary number system (but we will not need this here).

In the definition below, the operator π_p maps non-negative integers to reals in $[0,1)$ as follows

$$\pi_p\left(\sum_{j=0}^{m} a_j\, p^j\right) \equiv \sum_{j=0}^{m} a_j\, p^{-j-1} \qquad (a_j \in \{0,1,\ldots,p-1\},\ m \in \mathbb{N}_0) \tag{3.29}$$

or, using the representation in the p-ary number system,

$$\pi_p(a_m \ldots a_1 a_0) = 0.a_0 a_1 \ldots a_m \qquad (a_j \in \{0,1,\ldots,p-1\},\ m \in \mathbb{N}_0).$$

Here, $\mathbb{N}_0$ is the set of non-negative integers.

To find a sequence of points $\mathbf{x}_j$ in $[0,1]^d$ that match well a uniform distribution, for the purpose of computing d-dimensional integrals, we choose d primes $p_1, \ldots, p_d$. The first few primes $2,3,5,7,\ldots$ will do nicely. Then, using the operators π_{p_i}, we define our jth point $\mathbf{x}_j$ in d-space as

$$\mathbf{x}_j \equiv (\pi_{p_1}(j), \pi_{p_2}(j), \ldots, \pi_{p_d}(j)) \qquad (j = 1,2,3,\ldots). \tag{3.30}$$

For $d = 1$, one dimensional, and $p_1 = 2$, we have the *van der Corput sequence* of Exercise 2.13. The above generalisation to higher dimensions is the *Halton sequence*.

As we learnt in (3.12), (3.20) and Th. 3.2, the errors in Monto Carlo integration using n points are *expected* to be of

$$\mathcal{O}\left(\frac{1}{\sqrt{n}}\right) \qquad (n \to \infty). \tag{3.31}$$

with the constant 'in the $\mathcal{O}$' of modest size and depending only on the function that is to be integrated and not on the dimension.[5] When using using predetermined points as the ones from the Halton sequence, then the discrepancy is well-defined and not of stochastic type (no need to include the phrase 'to be expected'). Halton showed that [**?**], with the first n Halton points, the discrepancy is

$$\mathcal{O}\left(\frac{\log^d(n)}{n}\right) \qquad (n \to \infty). \tag{3.32}$$

Except for a constant, this is almost the square of the expected error in Monto Carlo integration. This bound is regarded as the best possible one for any sequence and besides the Halton sequence there are numerous other sequences for which this bound on the discrepancy actually holds.

However, a few remarks are in place. With the Halton sequence, the constant in the $\mathcal{O}$ is approximately equal to the product $\Pi_{i=1}^{d}\kappa(p_i)$, where $p_1, p_2, \ldots, p_d$ are the primes used in the definition of the sequence (cf., (3.30)) and $\kappa(p) \equiv (p-1)/(2\log p)$. The constant depends 'super exponentially' on the dimension d. To limit the size of the constant, Faure used one prime number p only. For the first coordinate of the $\mathbf{x}_j$ he took $\pi_p(j)$. He formed the other coordinates of $\mathbf{x}_j$ by specific linear combinations $(\bmod p)$ of the a_j that form the first coordinate $\pi_p(j) = 0.a_0a_1 \ldots a_m$ of $\mathbf{x}_j$. He showed that the discrepancy is then also as in (3.32), but now with a constant that tends to 0 for $d \to \infty$ (actually, the constant is approximately $\kappa(p)^d/d!$). Unfortunately, the prime p needed for this result has to be of size at least d. For large values of p, the numbers $\pi_p(j)$ for $j = 1,2,3,\ldots$ increase from near 0 to 1 in p steps: these initial numbers (for, say, $j \approx p/2$ or $j \approx 3p/2$) can not considered to be really uniformly distributed. To compensate for this initial lack of non-uniformity n has to be taken large. This remark applies to Faure as well as to Halton points when high dimensional integrals have to be computed. There are hybrid methods that mix randomness and predetermination, for instance, using the van der Corput sequence in the first coordinate and for each following coordinate a random permutation (a different one for each coordinate) of the van der Corput sequence. The Sobol sequence also relies on the van der Corput sequence (that is, on a p as small as 2). It

[5]It can be proved that for D_n as introduced in Footnote 4, $D_n = \mathcal{O}\left(\sqrt{\log(\log(n))/n}\right)$ for $n \to \infty$ with probability 1.

seems that, for low values of n, for all these sequences the error (discrepancy) is not much better then with a random sequence. Purely based on theoretical bounds, improvements on random sequences are only to be expected if n is larger than e^d (because the function $t \rightsquigarrow (\log t)^d/t$ actually *increases* for $t \in [1, e^d]$); for n larger than e^{20d} (!) it can be proved that the discrepancy is less than $n^{-0.95}$. Generally, the theoretical bounds are overestimates. Nevertheless, also in practise, only for large values of n the order $1/n$ shows (large depending on d and on the specific sequence, and large can be as large as 6^d).

Another popular class of low-discrepancy points are the so-called *lattice points*: for some carefully selected (irrational) numbers $a_1, \ldots, a_d$, the sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots$ of points $\mathbf{x}_j \in [0,1]^d$ is defined by

$$\mathbf{x}_j \equiv (ja_1 \bmod 1, ja_2 \bmod 1, \ldots, ja_d \bmod 1) \qquad (j = 1, 2, 3, \ldots),$$

where for a real number x, $r \equiv x \bmod 1$ is the real number in $[0, 1)$ for which $x - r$ is an integer. A popular choice for the a_i is $a_i = \sqrt{p_i}$ with p_i the ith prime number.

For more results and an extensive discussion on errors in integration with low-discrepancy points, see [24]. A recent overview, and a thorough mathematical discussion is in Reference [8].

★ **Exercise 3.8:** *The volume of the d-dimensional unit sphere.*
In this assignment you will compare Monte Carlo integration with simple sampling (cf., §3.2.4) to 'low-discrepancy sampling', using Halton's sequence. Our test problem is finding the volume V_d of the d-dimensional unit sphere Ω. The integral can be written as

$$V_d = \int_{-1}^{1} \cdots \int_{-1}^{1} \chi_{\{x_1^2+\ldots+x_d^2 \leq 1\}}\, dx_1 \ldots dx_d.$$

Explain why applying Monte Carlo with simple sampling to this integral in fact amounts to the hit-or-miss variant of Monte Carlo (cf., the first paragraph of §3.2.4).
Alternatively, as explained in §3.2.4, you can write down the integration $\int_\Omega \mathbf{1}\, d\mathbf{x}$ (with $\mathbf{1}$ is the constant function 1) using repeated 1-dimensional integration, and apply simple sampling Monte Carlo of §3.2.2. Of course, it suffices to compute the volume of the part of the unit sphere in $[0, \infty)^d$. In order to do this, you need to generalise

$$V_2 = 4 \int_0^1 \int_0^{\sqrt{1-y^2}} dx\, dy$$

to higher dimensions.
You can use the codes `hitormiss.cc` and `discrep.cc` from the course web page [36] for the low-discrepancy coding, and `sphere.cc` for sphere-specific routines.
The analytical formula for the volume is given by

$$V_d = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2} + 1)},$$

with the gamma function defined as in (1.5). You can implement it using the code from [33], or using MATLAB's built-in gamma function.
Make graphs of the error in the Monte Carlo method and the low-discrepancy method as a function of the number of points n, for several values of the dimensionality d. The errors should behave as in (3.31) and (3.32), respectively. ■

Chapter 4

Genetic Algorithms

4.1 Introduction

This chapter gives an introduction to Genetic Algorithms. Further introductions to the subject that the reader could consult are [5] and [14], but there is also a great number of resources available on the Internet.

Genetic algorithms (GAs) aim to solve *optimisation problems*, i.e., they try to either minimise or maximise some objective, subject to a number of parameters. There may be restrictions on the set of parameters. We will assume that the objective can be cast in the form of a function[1] $f : \mathcal{D} \to \mathbb{R}$, the so-called *objective* or objective function; f takes an x from some set $\mathcal{D}$ (containing the parameters) and produces an *objective value* or *objective score*, that is, a number $f(x)$. The goal is to find

$$\max_{x \in \mathcal{D}} f(x) \tag{4.1}$$

or, actually, to find a *solution*, i.e., an $x_{\text{opt}} \in \mathcal{D}$ for which $f(x_{\text{opt}}) = \max_{x \in \mathcal{D}} f(x)$:

$$x_{\text{opt}} = \text{argmax}\{f(x) \mid x \in \mathcal{D}\}. \tag{4.2}$$

Here, $\mathcal{D}$ is the domain of the function, i.e., the allowed range of the parameters. The requirement '$x \in \mathcal{D}$' incorporates *restrictions* on the set of parameters. If, say $\mathcal{D} = \mathbb{R}^m$, then m is number of parameters. We call any $x \in \mathcal{D}$ a *candidate solution*. Some authors simply call every $x \in \mathcal{D}$ a solution and refer to a desired x_{opt} as a best solution or an optimal solution.

Note that multiplication of the objective function by -1 turns minimisation into maximisation:

$$\min_{x \in \mathcal{D}} f(x) = -\max_{x \in \mathcal{D}}(-f(x)).$$

Example 4.1
An important class of minimisation problems is the minimisation of a norm of a residual defined by a matrix vector product, $\min_{\mathbf{x} \in \mathbb{R}^m} f(\mathbf{x})$, where $f(\mathbf{x}) \equiv \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2$ with $\mathbf{A}$ an $n \times m$ matrix $(m \leq n)$ and $\mathbf{b} \in \mathbb{R}^n$. Many practical problems from, e.g., physics are stated in this form. Then f represents the (discretised) energy of a system in state $\mathbf{x}$ and the objective is to minimise energy. ♦

Example 4.2
a) Let $\gamma_0, \gamma_1, \ldots$ a sequence of numbers (Fourier coefficients) in $[0, 1]$ with $\sum_k |\gamma_k| < \infty$ and $\sum_k |k\gamma_k| =$

[1]This function need neither be continuous nor differentiable. The function might be defined piecewise, or even take function values from a table. This generality makes GAs suitable for a wide range of problems. Alternative methods (steepest descent, conjugate gradients, Newton iteration) require the function to have some smoothness.

∞. For instance $\gamma_k = k^{-1.2}\alpha_k$ with α_k random numbers uniformly distributed in $[0,1]$. Define the objective function f on $\mathcal{D} \equiv [0,1]$ by

$$f(x) \equiv \sum_{k=0}^{\infty} \gamma_k \sin(2\pi x k) \qquad (x \in [0,1]).$$

This optimisation problem is not hard nor does it model a serious practical problem, but since this function has many local extrema it makes the problem of finding the global maximum an interesting test case for optimisation methods as genetic algorithms.
b) For $\alpha, \delta \in \mathbb{R}, \delta < 0$ consider the function

$$f(x) \equiv \delta x^2 + \sin(20\pi(x+\alpha)) \qquad (x \in [0,1]).$$

This function has a number of local maxima on $\mathcal{D} \equiv [0,1]$ with, in case $-\delta \ll 1$, almost equal objective value, which, as the previous example, turns this problem into an interesting test case. ♦

Example 4.3 (The Knapsack Problem)
Suppose we have a collection of m objects $O_0, O_1, \ldots, O_{m-1}$ with weights $w_0, w_1, \ldots, w_{m-1}$, respectively, and we have a knapsack that can carry a total weight of at most M; weights are positive real numbers. We want to carry objects from this collection in the knapsack with a total weight that is as high as possible without destroying the knapsack.
Each selection of objects can be described by a string $\mathbf{b} = b_{m-1}b_{m-2}\ldots b_1 b_0$ of m-bits, i.e., $b_j \in \{0,1\}$; for instance, 01101001 is a bit string of length $m = 8$. Object O_j is selected to be carried in the knapsack if and only if $b_j = 1$ $(j = 0, \ldots, m-1)$. The total weight $W(\mathbf{b})$ of selection $\mathbf{b}$ is

$$W(\mathbf{b}) \equiv b_0 w_0 + b_1 w_1 + \ldots + b_{m-1} w_{m-1}.$$

The assignment (the *Knapsack Problem*) is to find the $\mathbf{b}$ for which $W(\mathbf{b})$ is as large as possible, but such that $W(\mathbf{b}) \le M$.
In this example, $\mathcal{D}$ is the set of bit strings $\mathbf{b}$ of length m for which $W(\mathbf{b}) \le M$ and W is the objective function. ♦

Example 4.4 (The Travelling Salesman Problem)
Suppose $n+1$ cities $S_0, S_1, \ldots, S_n$ (or offices of companies) are to be visited (by a salesman) in one tour (round trip) that starts and ends in S_0, the 'home city of the salesman'. The shortest distance $d(i,j)$ between any pair of two cities S_i and S_j is known (either from a distance table, as will be the case in practise, or can easily be computed). Except for the start and end, the cities can be visited in any order. The assignment, the *Travelling Salesman Problem* (TSP), is to find the shortest round trip.
Any round trip can be described by a permutation x of $(1, 2, \ldots, n)$.[2] And any such a permutation x defines a round trip:

$$S_0 \to S_{x(1)} \to S_{x(2)} \to \ldots \to S_{x(n)} \to S_0. \tag{4.3}$$

The length $f(x)$ of a 'tour x' is given by

$$f(x) \equiv \sum_{i=0}^{n} d(x(i), x(i+1)) \tag{4.4}$$

where, to ease notation, we took $x(n+1) \equiv x(0) \equiv 0$: we have to find the permutation x that minimises f. Here, $\mathcal{D}$ is the collection $\mathcal{P}_n$ of all permutation of $(1, 2, \ldots, n)$. ♦

The objective function in Example 4.1 is smooth while the ones in Examples 4.3 and 4.4 are not even continuous (actually, the 'natural' topology for the space of bit strings and the space of permutations for

[2] x is a *permutation* of $(1, 2, \ldots, n)$ if $x(i) \in \{1, 2, \ldots, n\}$ for all $i = 1, \ldots, n$ and $x(i) \neq x(j)$ if $i \neq j$. A permutation can be viewed as a shuffling of carts: put the name of each city on a card (one card per city); shuffle the cards and visit the cities in the order of the cards.

these examples are discrete).
The function f of the first example has a lot of structure. Genetic algorithms do not take advantage of any structure that an optimisation problem may have. These algorithms are very general which makes them applicable to many optimisation problems, but, on the other hand, it makes them inefficient as well: generally, algorithms that exploit (some) structure are (much) more efficient than general algorithms as the genetic ones. Genetic algorithms are therefore only recommended in cases where there is no alternative solution method that exploits structure. There are many algorithms in the field of Numerical Linear Algebra that efficiently solve the problem of Example 4.1 even for values of m and n for which the problem is to large to be handled at all by GAs. For details on methods exploiting structure, see, for instance, [3, Ch.1] and [29].
The functions in Example 4.2 are continuous (the one in a) is not differentiable). In practice, the functions will be discretised, i.e., only finitely many terms will be summed (in case a)) and $\mathcal{D}$ will be a set of sample point as $\{jh \mid j = 0, \ldots, N\}$ with $h \equiv 1/N$ (Note that in case b), depending on α and h, the maximiser of the discrete problem can differ from the one of the 'continuous' problem). Even with values as large as $N = 10^5$, computers will not have trouble in finding the maximum by 'brute force', that is, by simply checking all function values in the sample points (Fast Fourier Transform can be used in a) for computing the function values). Nevertheless, as 'toy' problems, the problems are of interest: they have many local maxima and many optimisation methods that rely on smoothness of the objective function tend to get stuck in local maxima. For instance, in example b), a method as Newton–Gauss, i.e., basically Newton for solving $f'(x) = 0$, easily detects a local maximum, but is not able to 'leave' a local maximum in a search for the global maximum.
A brute force approach can not be applied to solve 'serious' problems as the ones in the Examples 4.3 and 4.4: for instance, to solve the TSP for 35 cities by simply checking all possibilities, more than 10^{40} tours have to be investigated. This would take more than 10^{17} year on the fastest computer (n numbers can be permuted in $n!$ different ways).
For a more extensive discussion on optimisation problems for which genetic algorithms are considered to be most suitable, see §4.4.2.

Since solving the optimisation problem exactly is usually impossible, one settles for a *near optimal solution*, i.e., for an $x_{\text{nopt}} \in \mathcal{D}$ for which $f(x_{\text{nopt}}) + \varepsilon \geq \max_{x \in \mathcal{D}} f(x)$ for some small $\varepsilon > 0$, or even for 'candidate' solutions in $\mathcal{D}$ that are *likely* to be near optimal.

The graph of f is known as the *fitness landscape*, inspired by two-dimensional graphs where you have hills and valleys. Finding the global maximum (minimum) means finding the highest point on the highest hill (lowest point in the lowest valley) without getting stuck on another hill (in another valley).

Genetic algorithms mimic evolution as it happens in nature. A set of candidate solutions, called a generation, is evolved through several generations. In each generation, members of that generation are selected, combined and mutated to form the next generation. One hopes that good candidate solutions combine with other good candidates, creating better candidate solutions and thereby evolving towards the optimal state.

Let us discuss the algorithm in some more detail. We denote a *generation* by

$$\mathcal{G} = (x_1, \ldots, x_n).$$

The elements $x_j \in \mathcal{D}$ are the *members* or *chromosomes* of generation $\mathcal{G}$.[3] Note that a situation where $\mathcal{D}$ is a set of bit strings of length m, as in in Example 4.3, nicely fits the biological point of view: a bit string represents a chromosome, the bits in the string correspond to genes on the chromosome, and the value of a bit is an allele. $\mathcal{G}$ may be interpreted as a matrix with columns the vectors x_j. The number of columns of $\mathcal{G}$, the size of the generation, is denoted by $|\mathcal{G}| \equiv n$. The first generation needs to be initialised. Usually its members are generated randomly. From this point onward the genetic algorithm is an iterative process, where in each step operators may be applied to $\mathcal{G}$ to form a next generation. The three main operations are *selection*, *combination*, and *mutation*. *Crossover* is a popular way of combining two candidate solutions to form new candidate solutions. Before describing these operations we need a way of evaluating how well

[3]In some situations, one member is one chromosome, in other situations, one member may consist of several chromosomes, cf., §4.1.3.

a member optimises our objective function. In genetic algorithm terminology: we want to determine the *fitness* for members of a generation. Intuitively the 'biological' term 'fitness' fits best to maximisation. We therefore focus on maximising our objective function.

4.1.1 Determining the fitness

There are countless measures of fitness. For example, if the true maximum value t to (4.1) is known, then one may evaluate $f(x)-t$. Unfortunately, in general, we do not know the true maximum. However, we can consider the x_j for which $f(x_j)$ is largest in generation $\mathcal{G} = (x_1,\dots,x_n)$ as the best candidate solution in that generation. We also have a second-best candidate solution, etc. Now, we can assign values to x_j according to its place in the resulting ordering. Let p be a permutation of $(1,\dots,n)$ such that

$$f(x_{p(k)}) \geq f(x_{p(k+1)}) \quad \text{for} \quad k = 1,\dots,n-1.$$

Then the *fitness function* $\mathcal{F}$ for this generation could be defined by

$$\mathcal{F}(x_{p(j)}) = n - j + 1 \qquad (j = 1,\dots,n). \tag{4.5}$$

In this 'ranking' approach it does not matter much how much better a candidate solution is with respect to another candidate solution, i.e., it does not matter how much larger its objective value is. As an alternative, we can define the fitness function of generation $\mathcal{G} = (x_1,\dots,x_n)$ by

$$\mathcal{F}(x_j) = a + (b-a)\left(\frac{f(x_j) - f_-}{f_+ - f_-}\right) \qquad (j = 1,\dots,n). \tag{4.6}$$

Here, $f_- \equiv \min_j f(x_j)$ and $f_+ \equiv \max_j f(x_j)$ and a and b are some preselected positive real numbers with $b > a$. In this way, the objective values are scaled to values in $[a,b]$. The scalars a and b can be used to control the impact of differences in fitness. For instance, the condition $a > 0$ ensures that a fitness score of zero will not occur, thus giving all members of the generation a positive fitness. If $b = a + 1$ and a is very large, then all members of the generation will have approximately the same fitness (relative to each other). Consider the influence of the parameters a and b on the fitness of one member of generation $\mathcal{G}$ as compared to the fitness of its fellow generation members. What will happen in the limiting cases where $b \gg a$ or $b \approx a$?

4.1.2 Selection and reproduction

Selection is the process of determining which individuals from the current generation $\mathcal{G} = (x_1,\dots,x_n)$ will survive to the next generation. Selection is random favouring the fittest individuals. The most common form of selection is *fitness proportional selection* where the probability for x_j for surviving is given by the fitness $\mathcal{F}(x_j)$ of (4.6) devided by the 'total' fitness $T \equiv \sum_{j=1}^{n} \mathcal{F}(x_j)$ of the current generation. There is a convenient way of implementing this procedure, called the *roulette wheel*. In this approach, we first draw a random number S between zero and T (random with respect to the uniform distribution; denoted as $S = \text{Random}(T)$) and then start adding the numbers $\mathcal{F}(x_1)$, $\mathcal{F}(x_2)$, ..., until S is exceeded. Then, the x_k for which this happens is selected. Symbolically we represent the selection as

$$\mathcal{S}_S(\mathcal{G}) \equiv x_k \qquad (\text{with } k \text{ such that } \sum_{j<k} \mathcal{F}(x_j) \leq S, \quad \sum_{j\leq k} \mathcal{F}(x_j) > S):$$

the operator $\mathcal{S}_S$ selects a member of $\mathcal{G}$ to be member of the next generation.[4] The analogy with the roulette wheel is obvious if one imagines a roulette wheel to be partitioned into sections with angle-width equal to the relative fitnesses $\frac{2\pi}{T}\mathcal{F}(x_j)$ of the members x_j of $\mathcal{G}$ $(j = 1,\dots,n)$. The probability that the roulette ball falls in the section of the roulette wheel associated to a certain member x_j is proportional to the size

[4]Putting the fitness of a member to zero upon selection can be convenient as a flag to prevent the member from reselection.

of the section, whence to the fitness of x_j. The variable S marks the spot where the ball rests on the wheel. The drawing of S followed by operating $\mathcal{S}_S$ on $\mathcal{G}$ will be performed a number of times to form the set of 'survivers'.

A number of survivers will be selected for *reproduction*. How many will be selected depends on the specific reproduction procedure that is to be used. The most common practice is that survivers all have the same probability R of being selected: if x is a survivor then x is selected for reproduction if $S < R$ with $S =$ Random(1.0) newly drawn for each survivor x. Here, $R \in (0, 1]$ is determined in advance.

Whenever two members have been selected for reproduction, say x_{j_0} and x_{j_1}, two offspring, say y_0 and y_1 are produced (see Section 4.1.4 for various strategies). Symbolically this may be represented by

$$(y_0, y_1) = \mathcal{C}(x_{j_0}, x_{j_1}).$$

$\mathcal{C}$ is a *combination operator*.

If we adopt *steady state* reproduction, where each generation has the same number of members, then we arrive at Algorithm 4.1.

Algorithm 4.1 The main GA loop with selection and steady state reproduction.

```
Select $R \in (0,1)$. Initialise $\mathcal{G}$   // Form the initial generation.
while no convergence do
  $p \leftarrow 0$,   $\mathcal{G}^+ = ( )$   // The next generation is initially empty.
  while $|\mathcal{G}^+| < n$ do
    while $p < 2$ and $|\mathcal{G}^+| < n$ do
      $S \leftarrow$ Random(totalFitness),   $y \leftarrow \mathcal{S}_S(\mathcal{G})$,
      Add $y$ to $\mathcal{G}^+$   // 'Fit' members of the current generation 'survive' to the next generation.
      if Random(1.0) $< R$ then
        $x_p \leftarrow y$,   $p \leftarrow p + 1$   // Select parents from the survivers.
      end if
    end while
    if $|\mathcal{G}^+| < n$ then
      $(y_0, y_1) \leftarrow \mathcal{C}(x_0, x_1)$,   // Form two offsprings.
      $p \leftarrow 0$,   Add $y_0$ to $\mathcal{G}^+$,   // Add the first offsprings to the next generation.
      if $|\mathcal{G}^+| < n$ then
        Add $y_1$ to $\mathcal{G}^+$,   // Add the second offsprings to the next generation if there is room.
      end if
    end if
  end while
  $\mathcal{G} \leftarrow \mathcal{G}^+$.   // Replace the current generation by the next generation.
  Perform analysis on $\mathcal{G}$
  Write data to file
end while
```

There are variations on the above selection procedure. For instance, if $y = \mathcal{S}_S(\mathcal{G})$ is selected, then y can be excluded from reselection: a member can not occur twice in the same generation or, as another alternative, can not mate twice.

A *tournament strategy* is a popular alternative to the roulette wheel approach. In a tournament, $y = \mathcal{S}(\mathcal{G})$ is selected as follows. Randomly two members of $\mathcal{G}$ are selected (say, x_{j_0} and x_{j_1}) for a 'tournament'. The fittest of them 'wins' the tournament and is selected as y (i.e., $y = x_{j_0}$ if $\mathcal{F}(x_{j_0}) > \mathcal{F}(x_{j_1})$). Instead of tournaments with two participants, tournaments can be hold with three (or more). Another variant arises by letting the fittest win the tournament with probability W for some preselected $W \in (0, 1)$. And of course, reselection can be allowed or not.

4.1.3 Chromosomes, bit strings, numbers, vectors, and permutations

The above selection strategies depend on the objective values. The other two main operations, 'combination' (or crossover) and 'mutation' depend on the parameters x and the restrictions '$x \in \mathcal{D}$'. As observed in §4.1, parameters x that are represented as bit strings nicely fit the terminology and the 'biological' point of view taken in genetic algorithms. As we learnt in Example 4.3, a representation of the parameters as bit strings may come naturally (see also §4.4.4), other problems may require some reformulation as we will discuss below (see also §4.2).

Let $\mathcal{B}_m$ be the set of all bit string $\mathbf{b} = b_{m-1}b_{m-2}\ldots b_1 b_0$ of length m, i.e., $b_j \in \{0, 1\}$.

Bit strings can be viewed as binary representations of integers. To be more precise, the map ι defined by

$$\iota(\mathbf{b}) \equiv \sum_{j=0}^{m-1} b_j\, 2^j \qquad (\mathbf{b} = b_{m-1}\ldots b_0 \in \mathcal{B}_m)$$

identifies bit strings $\mathbf{b}$ of length m with non-negative integers $k = \iota(\mathbf{b})$ of size $< 2^m$.

A combination of ι with the map $k \rightsquigarrow \alpha + \gamma k$ with $\gamma \equiv (\beta - \alpha)/(2^m - 1)$, maps bit strings to a subset of reals in the interval $[\alpha, \beta]$. Here, $\alpha, \beta \in \mathbb{R}$, $\alpha < \beta$. With m sufficiently large this subset might be sufficiently dense for accurately solving problems (with continuous objective function) in case $\mathcal{D} = [\alpha, \beta]$ or $\mathcal{D} = \mathbb{R}$ and α sufficiently small and β sufficiently large. Note that, here, (as always when pursuing numerical solutions) we actually aim for solving a 'discretised' version of the optimisation problem.

A string $\mathbf{b}_1\mathbf{b}_2\ldots\mathbf{b}_n$ of n bit strings $\mathbf{b}_k$ each of length m can be mapped to an n-vector $(\iota(\mathbf{b}_1), \ldots, \iota(\mathbf{b}_n))^{\mathrm{T}}$ with integer coordinates (of limited size) and further with a map as $\vec{k} \rightsquigarrow \alpha + \gamma\,\vec{k}$, , $\gamma \equiv (\beta - \alpha)/(2^m - 1)$, to an n-vector of reals. Obviously, the string $\mathbf{b}_1\mathbf{b}_2\ldots\mathbf{b}_n$ of bit strings can be merged into one huge bit string of length mn. However, depending on the problem, it may be more effective to let one member of a generation consist of n bit strings, or, if one wishes, of n chromosomes of length m, rather than of one huge chromosome.

Clearly a permutation x of $(1, \ldots, n)$ can be represented by an n-vector $(x(1), \ldots, x(n))^{\mathrm{T}}$ of non-negative integers. On the other hand, an n-vector $(k_1, \ldots, k_n)^{\mathrm{T}}$ of positive integers (possibly with multiple occurrence of certain integers and possibly with some k_i larger than n) defines a permutation: let x be such that $k_{x(1)} \leq k_{x(2)} \leq \ldots \leq k_{x(n)}$. We take $x(i) < x(j)$ if $k_i = k_j$ and $i < j$. This allows us to map a string of n bit strings each of length m to a permutation of $(1, \ldots, n)$. (See also Exer. 2.10.)

Above we suggested maps from $\mathcal{B}_m$ to $\mathcal{D}$. Note that, if

$$g : \mathcal{B}_m \to \mathcal{D} \quad \text{and } g \text{ is surjective,}^{5}$$

then solving the optimisation problem

$$\mathbf{b}_{\text{opt}} = \text{argmax}\{f \circ g(\mathbf{b}) \mid \mathbf{b} \in \mathcal{B}_m\}$$

solves the original problem, because $x_{\text{opt}} = g(\mathbf{b}_{\text{opt}})$. The inverse of g need not exist (or may be hard to evaluate), but is not needed.

Many (all?) optimisation problems can be formulated as optimisation problems for bit strings with bit strings as chromosomes. However, the appropriate set of bit strings might be much larger than $\mathcal{D}$, and, more importantly, by the bit string representation of the problem, we may loose (view on the) structure in the problem. Generally, the more structure that can be exploited in a solution method, the more efficient the method is. In some cases, it already helps to find another way of representing chromosomes as bit strings. In §4.3.2 below, we will discuss an alternative map from $\mathcal{B}_m$ to the set of non-negative integers of size $< 2^m$. This map is suppose to better preserve 'nearness' of integers in bit string representation than ι does and may allow to better exploit continuity of the objective function in case the xs in $\mathcal{D}$ have real

[5] g is surjective on $\mathcal{D}$ if for each $x \in \mathcal{D}$ there is an $\mathbf{a} \in \mathcal{B}_m$ such that $g(\mathbf{a}) = x$

entries. Nevertheless, it is common practice to represent a chromosome by a bit string.[6] But this is by no means mandatory. One can also let a chromosome be represented by an integer, a vector of integers, a vector of reals, a permutation or whatever fits your problem and allows best to exploit structure. For instance, if you have a weighted graph and aim to find the shortest path between two given points, then the chromosomes would represent different paths, not numbers. There are also problems that do not seem to allow a formulation in terms of bit strings. However, by using the theory of *schemata* (see [15]) it is possible to show that a bit string formulation is effective in searching the fitness landscape.

In the discussion below on 'crossover' and 'mutation', we assume that the chromosomes x are represented by bit strings. If they are represented differently, then the selection and mutation operators should be redefined to be suitable for the particular representation of the xs (and problem).

4.1.4 Crossover

Crossover is the mechanism that does the actual optimisation. Selected members are combine with the crossover technique to make new members for the next generation. The hope is that the result of mixing two good chromosomes creates a new good chromosome and hopefully even a better one. One often used type of crossover for bit strings is *one point crossover*. Two parent bit strings are selected, for instance,

$$\begin{array}{c|c} 0\,0\,1\,1\,0 & 0\,1\,1 \\ b\,b\,b\,a\,b & b\,b\,b \end{array}.$$

For clarity of presentation, we replaced 1 by b and 0 by a in the second parent. At a certain randomly chosen position k in the strings we swap all elements of the chromosome that lie to the right of it. With $k = 6$ in the example, this leads to the offsprings

$$\begin{array}{c|c} 0\,0\,1\,1\,0 & b\,b\,b \\ b\,b\,b\,a\,b & 0\,1\,1 \end{array}.$$

One can also try other crossover strategies, with for example *two point crossover*,

$$\begin{array}{c|c|c} 0\,0\,1 & 1\,0 & 0\,1\,1 \\ b\,b\,b & a\,b & b\,b\,b \end{array} \qquad \rightsquigarrow \qquad \begin{array}{c|c|c} b\,b\,b & 1\,0 & b\,b\,b \\ 0\,0\,1 & a\,b & 0\,1\,1 \end{array}.$$

or, more generally, *multiple crossover points*, and uniform crossover. In *uniform crossover*, a sequence of positions is randomly selected (say, $(2, 3, 5, 7)$) and crossover is performed at these positions:

$$\begin{array}{c} 0\,0\,1\,1\,0\,0\,1\,1 \\ b\,b\,b\,a\,b\,b\,b\,b \end{array} \qquad \rightsquigarrow \qquad \begin{array}{c} 0\,b\,b\,1\,b\,0\,b\,1 \\ b\,0\,1\,a\,0\,b\,1\,b \end{array}.$$

You should adjust your crossover strategy to reflect the properties of your specific problem at hand.

4.1.5 Mutation

In nature, mutation is a spontaneous slight change in a chromosome. Mutation is included in order to keep the population diverse. One important effect of mutation is to allow the population to get out of a *local optimum*, in order to find the *global optimum*. The simplest method of applying mutation is to firstly set the probability of mutation to a (small) fixed value $M \in (0, 1]$. Next, each element within each chromosome is

[6] Internally, this bit string is implemented by a data structure such as a *linked list* or *array*.

visited (for example the bits in the bit string representing the chromosome) and a random number $p \in [0, 1]$ is drawn. If $p < M$, then the corresponding element of chromosome is slightly changed, for example by a bit flip if the chromosomes are represented by bit strings. Take care when choosing the parameter M. It should be chosen in such a way that candidate solutions are not destroyed prematurely (by having M too high), but on the other hand diversity in the population should be maintained. Typically values for M seem to be of order $M = 0.1\%$.

4.1.6 Stopping criteria

One needs to think about when to stop the genetic algorithm. Since we do not know the true solution (in practical circumstances) we need a way to detect ‘convergence’. A practical way of measuring convergence is by monitoring the chromosome with the highest value of the objective function and check whether it has changed significantly during the last few iterations. The chromosome with the highest objective score is also a relevant value to write to file during the GA iteration. A plot of this value versus generation gives insight in the convergence of the algorithm. Alternatively, you could plot the average objective values.

☞ **Exercise 4.1:** *The genetic algorithm.*
Modify Algorithm 4.1 to include mutation and a stopping criterion based on convergence. ■

4.2 Implementation

In this section several implementation issues are discussed. Here we will discuss how to represent chromosomes as bit strings, and have the bit strings represent real numbers. When using bit strings to encode chromosomes, mutation becomes a simple bit flip.

You should test the exercises in the following paragraphs by running the genetic algorithm on some suitably chosen test problems. Interesting test problems have several local minima (e.g. the continuous function minimisation exercise on the course web page)! Use the code that is available from the course web page, read the part about *classes* in appendix B if you have not done so already.

★ **Exercise 4.2:** *Representing integers.*
We need a representation of integers before we can start optimising functions. Create a derived class based on the `Chromosome` class, that interprets its bit string as an integer. Start from the provided class `IntChromosome` and define all *virtual* member functions. Recall that a binary number translates to a decimal number as follows:

$$k = \sum_{i=0}^{m-1} 2^i \, b_i,$$

where b_i is a bit, $b_i \in \{0, 1\}$ (in the chromosome) and m is the total number of bits. ■

★ **Exercise 4.3:** *Representing real numbers.*
In order to represent real numbers we firstly suppose we have an integer k represented by the chromosome. We can then transform to the real number by

$$r = \gamma \, k + \alpha,$$

where γ and α are yet to be determined real numbers. Note that $k_{\text{min}} = 0$ and $k_{\text{max}} = 2^m - 1$, where m is the number of bits in the chromosome. Now

$$\begin{aligned} r_{\text{min}} &= \gamma \, k_{\text{min}} + \alpha \\ r_{\text{max}} &= \gamma \, k_{\text{max}} + \alpha. \end{aligned}$$

These equations are easily solved,

$$r = \left(\frac{r_{\text{max}} - r_{\text{min}}}{2^m - 1}\right) k + r_{\text{min}}.$$

Of course we cannot represent every real in $[r_{\text{min}}, r_{\text{max}}]$, but the higher the value of m, the better we do. For example, 4 bit numbers in $[1, 3]$ yield the transformation $r = 2k/15 + 1$. Create a derived class based on the `Chromosome` class, that interprets its bit string as a real number in $[r_{\text{min}}, r_{\text{max}}]$. ■

★ **Exercise 4.4:** *Mutation.*
The next step in the genetic algorithm is the implementation of mutation. The ability to mutate is a property of a chromosome. Write the member function `void mutate(double prob)`, in the class `Chromosome`. Use bit flips, invert the bits according to the probability `double prob`. (Hint: look at `getBitstringText()` for an example of how to work with *iterators*) ■

★ **Exercise 4.5:** *Selection and Crossover.*
Selection and crossover are somewhat more complicated to implement. The most straightforward part is writing a member function `selectByRoulette` of the class `Population` which determines if a member becomes selected using the roulette wheel methodology. You may want to set the fitness of selected members to zero, to avoid future reselection. Test whether the selection function works before implementing crossover. For crossover (which comes before mutation!) you need to create a new empty population and fill it as outlined in Algorithm 4.1. Implement this in the member function `selectAndReproduce`. Look at the methods `selectAndReproduce()` in `Population` and `crossover` in `Chromosome`. Do not forget to delete the old population after the new population has been generated. ■

★ **Exercise 4.6:** *Stopping criteria.*
We now have a genetic algorithm working on real numbers and integers, with crossover and mutation. In the main program, the population evolves through a fixed number of generations. Change the stopping criterion such that the genetic algorithm loop is terminated when there is convergence (within a specified tolerance). ■

4.3 Improvements

4.3.1 Elitism

The genetic algorithm mutates every element in the chromosome with a certain small probability. This implies that also the best candidate solutions in the current population could be mutated and this may be disadvantageous. A solution to this problem is to use *elitism*, in this case the best or the few best individuals are always selected for surviving and are not mutated.

☞ **Exercise 4.7:** *Implementation of elitism.*
Modify Algorithm 4.1 to include elitism. ■

★ **Exercise 4.8:** *Elitism.*
Implement elitism and measure the gain in efficiency. Present your results for a few test functions as a graph of generation number versus the error. If you have time it is also interesting to see if more than one elite member gives significant speed-up to the algorithm. ■

4.3.2 Gray codes

The representation of integers by bit strings $\mathbf{b} \equiv b_{m-1} \dots b_0$ using $\iota(\mathbf{b}) \equiv \sum_{j<m} b_j \, 2^j$ may have an important drawback. Integers can be nearby, while the corresponding bit strings can be very different, for

instance, the integers 31 and 32 have bit string representation 011111 and 100000, respectively. Conversely, bit strings can be nearby for very different integers, as the integers 0 and 32 with bit string representation 000000 and 100000 illustrate. Note that this observation extends to real numbers if they are represented by bit strings using ι and the map $k \rightsquigarrow \alpha + \gamma k$ with $\gamma \equiv (\beta - \alpha)/(2^m - 1)$. As a consequent, a mutation by flipping one bit, that is supposed to be a small change, may actually be a huge change for (the objective value, whence for the fitness of chromosomes) the problem at hand.

It is favourable if small changes in the representation imply a small change in the represented number. One way to partially solve this is by using *Gray codes*. Such a code is a bijection χ from the set $\mathcal{B}_m$ of bit strings of length m to this set $\mathcal{B}_m$:[7] $\mathbf{g} \equiv \chi(\mathbf{b})$ is the Gray code bit string representation of the bit string $\mathbf{b}$. The map χ should have the property that incrementing or decrementing an integer k will change only one bit of the corresponding 'Gray coded' bit string $\mathbf{g}$: $k = \iota(\chi^{-1}(\mathbf{g}))$. Gray codes for a given bit length are not unique, there are many different valid Gray codes. Note that, though incrementing or decrementing the number implies one bit flip, *the opposite is not true*. The probability that a bit flip leads to a smaller increase in the represented number has however become greater.

There exist efficient algorithms [33], as Alg. 4.2, to convert bit strings to Gray coded bit strings. Alg. 4.2 converts bit strings (or binary numbers, if you wish) $b_{m-1} \dots b_0$ to Gray coded bit strings (Gray code) $g_{m-1} \dots g_0$ (Alg. 4.2, left; this defines χ) and vice versa (Alg. 4.2, right; this defines χ^{-1}).[8]

Algorithm 4.2 Converting binary numbers to Gray codes and visa versa.

```
// Converts a binary number $b_{m-1} \dots b_0$ to a Gray code $g_{m-1} \dots g_0$ and visa versa.

// From binary to Gray code
$g_{m-1} \leftarrow b_{m-1}$
for $i = m - 2$ to 0 do
  if $b_{i+1} = b_i$ then
    $g_i \leftarrow 0$
  else
    $g_i \leftarrow 1$
  end if
end for

// From Gray code to binary
$b_{m-1} \leftarrow g_{m-1}$
for $i = m - 2$ to 0 do
  if $b_{i+1} = g_i$ then
    $b_i \leftarrow 0$
  else
    $b_i \leftarrow 1$
  end if
end for
```

☞ **Exercise 4.9:** *Gray code generation.*
Use algorithms 4.2 the verify that the following two numbers are the same:

Binary:	10100111100111
Gray code:	11110100010100

Consider the binary number $a = 1001111$. Create two other binary numbers by adding 1 and by subtracting 1 from a. Give the Gray code representation of these three numbers. Inspect the effect on the number of bit flips in both representations of adding and subtracting 1 from a.
How much differ 110000 and 010000 when these numbers represent integers in the Gray code? ■

☞ **Exercise 4.10:** *Validity of algorithms 4.2.*
Prove that the bit string $c_n \dots c_0$, obtained from first applying algorithm 4.2 left and then algorithm 4.2 right to a bit string $b_n \dots b_0$, satisfies $c_i = b_i$ for $i = 0, \dots, n$. Conclude that algorithm 4.2 right is an inverse of algorithm 4.2 left. In the same way, show that algorithm 4.2 left is an inverse of algorithm 4.2 right. ■

[7] χ is a *bijection* if it is injective, $\chi(\mathbf{b}_1) \neq \chi(\mathbf{b}_2)$ if $\mathbf{b}_1 \neq \mathbf{b}_2$, as well as surjective.
[8] The quickest way to implement these algorithms is using the exclusive or operation.

★ **Exercise 4.11:** *Gray codes.*
Add Gray codes to the genetic algorithm. This means that the bit string in a chromosome now has the interpretation of a Gray code. You will need to rewrite the `GetValue` member function, where instead of converting the bit string to decimal you should now convert the bit string (Gray code) to binary before converting to decimal.

You can add Gray code functionality to the genetic algorithm by either adding a flag `isGray` to a chromosome (as done with elitism), or defining a new chromosome type (for example a `RealGrayChromosome` derived from a `RealChromosome`).

Test your Gray code-enabled algorithm for efficiency improvement. ■

4.3.3 Local search methods

Like GA, *local search methods* try to maximise (or minimise) an objective function f. Unlike GA, they work with only one canditate solution, say x_{k-1}, in each step and they try to find a better candidate solution by a local search, i.e., by searching a neighbourhood of x_{k-1}.

The global structure of a local search algorithm is outlined in Algorithm 4.3. Note that we use supscripts for iteration indices. Since $\mathcal{D}$ usually is multi-dimensional, we also have to indicate the coordinates of iterates. For this, we will use the function value notation: $x_k(j)$ is the jth coordinate of the iterate $x_k = (x_k(1), \dots, x_k(d))$ (i.e., we view a sequence or a vector as a map on $\{1, \dots, d\}$). The $\mathcal{N}(x_{k-1})$ in line 3 is a subset of $\mathcal{D}$ consisting of candidate solutions which are, in some respect, close to x_{k-1}. If the candidate solutions x are elements of $\mathbb{R}^d$, then for some small $\delta > 0$, such a 'neighbourhood' $\mathcal{N}(x)$ could, for example, consist of all vectors y in $\mathbb{R}^d$ that are equal to x except at at most one coordinate, say the jth, at which we have that $|x(j) - y(j)| \le \delta$. Commonly, $\mathcal{N}(x)$ is constructed such that the values $f(y)$ for $y \in \mathcal{N}(x)$ do not vary too much, but also such that the set $\mathcal{N}(x)$ is 'small' enough so that choosing an x_k in $\mathcal{N}(x_{k-1})$ (as in line 4), can be implemented efficiently. Algorithms from this family thus iteratively improve a given candidate solution, and if the initial guess x_0 was close to a local maximum in f, it is expected that local search methods will converge to that maximum. A very simple method would repeatedly construct an $\mathcal{N}(x_{k-1})$, proceed with calculating $f(y)$ for all $y \in \mathcal{N}(x_{k-1})$, and finally choose the largest and make that the next candidate solution: $x_k = \text{argmax}\{f(y) \mid y \in \mathcal{N}(x_{k-1})\}$. This variant is called *greedy local search*.

Algorithm 4.3 Outline of a local search algorithm.

```
1: Let $x_0 \in \mathcal{D}$ be an input guessed solution (ideally such that $f$ is large at $x_0$)
2: **for** $k = 1, 2, \dots$ **do**
3:     Let $\mathcal{N}(x_{k-1})$ be the *neighbourhood* of $x_{k-1}$
4:     Choose an $x_k \in \mathcal{N}(x_{k-1})$ such that $x_k$ is expected to be an improvement on $x_{k-1}$
       (hence expecting that $f(x_k) \ge f(x_{k-1})$)
5:     **break if** $f(x_k)$ is sufficiently large or if $k$ is too large
6: **end for**
7: **return** $x_k$.
```

Local search methods differ from each other mostly on how an improvement on x_{k-1} (line 4) is selected, and, as such, on which conditions they require on f. For example, the *steepest descent* method, used in Algorithm 4.7, requires that the partial derivatives of f are known (and can be efficiently evaluated). As the name implies, the candidate solution x_{k-1} is shifted a small step in the direction the gradient $\nabla f(x_{k-1})$ of f leading to an increase of f (we are actually taking the 'steepest ascent' direction). In the case of $f : \mathcal{D} \to \mathbb{R}$, $\mathcal{D} \subset \mathbb{R}^d$, with known derivatives, a simplified algorithm would look as in Algorithm 4.4. Here the δ in line 5 is a positive real number such that $f(x_k) > f(x_{k-1})$. The construction of an appropriate δ (the best?) is called a *line search*. Often a δ is constructed by *back tracking*: select a $\delta > 0$ and repeatedly reduce δ by a factor, say, 2 until $f(x_k) > f(x_{k-1})$. *Conjugate gradients* (CG) is a modification of steepest descent that combines x_{k-1} and $\nabla f(x_{k-1})$ with x_{k-2} to form the new iterate x_k. CG is very effective in cases where the optimal δ can be efficiently computed. Steepest decent and CG rely on linear approximations of f around the iterates ($f(x_{k-1}+h) \approx f(x_{k-1}) + (\nabla f(x_{k-1}))^{\mathrm{T}} h$). *Gauss–Newton* uses quadratic approximations with the Hessians of f and leads locally to quadratic convergence, whereas steepest descent and CG can only converge linearly. Generally faster algorithms require smoother objective functions. For more details, [3, Ch.1] and [29] are good sources.

A variant of local search, called *Tabu Search*, accepts also worsening the candidate solution (for instance, when the search is stuck at a local maximum), but, as explained in [40], 'in addition, prohibitions (henceforth the term tabu) are introduced to discourage the search from coming back to previously-visited solutions'. Tabu search can be effective also as a *global* optimiser.

Algorithm 4.4 A simplified steepest descent algorithm.

```
// $f : \mathcal{D} \to \mathbb{R}$, $\mathcal{D} \subset \mathbb{R}^d$, with known derivatives.
Let $x_0$ be the input guessed solution.
for $k = 1, 2, \ldots$ do
    Select a $\delta > 0$ such that $f(x_{k-1} + \delta\,\nabla f(x_{k-1})) > f(x_{k-1})$
    $x_k = x_{k-1} + \delta\,\nabla f(x_{k-1})$
    break if $f(x_k)$ is sufficiently large or if $k$ is too large
end for
return $x_k$.
```

The *simulated annealing* algorithm of Algorithm 4.5 is a well-known *global* search algorithm that can also be used for local search. This algorithm does not need any special requirements on f. It simply selects a candidate solution y from the neighbourhood $\mathcal{N}(x_{k-1})$ of x_{k-1} at random, and then decides whether or not to accept this y as the next candidate solution. When $f(y) \geq f(x_{k-1})$, no further questions are asked and y is accepted as x_k. If y is worse than x_{k-1}, then simulated annealing will still accept y as the next iterate with probability α, where, for some $T > 0$,

$$\alpha = \exp\left(\frac{1}{T}[f(y) - f(x_{k-1})]\right).$$

The possibility to move to worse candidate solutions avoids to get stuck in a (very) local maximum. As the iteration progresses, the candidate solution should stabilise in some local maximum; this is attained by decreasing T (per step with some fixed small portion: $T \leftarrow (1-\delta)T$). The reduction of T during the process makes it less likely that the process moves to candidate solutions that are much worse. An extra is to always store the best attained candidate solution and return the best (in Algorithm 4.5 indicated as comment), instead of simply returning the last obtained candidate solution x_k, when the algorithm terminates. Theoretically the method is likely to converge towards a global maximum, but, depending on the size of the initial T, it is more likely that a local maximum close to the initial x_0 will be detected in an early stage of the process. Therefore, simulated annealing may do well as a local search method, where only a (fixed) limited number of steps of simulated annealing is performed. The random choices in the algorithm (of y in $\mathcal{N}(x_{k-1})$ and of r) are with respect to a uniform distribution.

Local search algorithms can enhance the efficiency of genetic algorithms greatly by applying local search on individual chromosomes: with local search and genetic algorithms we hopefully follow complementary goals. With local search we aim for efficiently finding the local maximum nearest to the individual chromosome, whereas with the genetic algorithm we aim for quickly detecting the global maximum among all local maxima. To phrase it differently, we want to avoid the genetic algorithm 'wasting' time on improving candidate solutions towards candidate solutions that are only local maximisers. In, for instance, Example 4.2.b, Newton–Gauss easily detects local maximisers, but is not likely to detect the global maximiser, whereas a GA algorithm would probably require many steps (as compared to Newton–Gauss) to accurately compute a local maximiser and even many more to find the global maximiser. In a combination, GA would actually be 'focussing' only on the discrete problem of detecting the global maximiser among the (limited number of) local ones.
To keep efficiency in check, it might be worthwhile to apply the extra search only once every so-many

Algorithm 4.5 Simulated annealing algorithm

```
Let $x_0$ be an input guessed solution. Let $T > 0$.    // $x_{best} = x_0$.
for $k = 1, 2, \ldots$ do
    Choose an $x_k$ from $\mathcal{N}(x_{k-1})$ randomly
    if $f(x_k) < f(x_{k-1})$ then
            // if $f(x_k) > f(x_{best})$, then $x_{best} = x_k$ end if
        Let $r$ be random in $[0, 1)$
        Compute $\alpha = \exp\left(\frac{1}{T}[f(x_k) - f(x_{k-1})]\right)$
        if $r > \alpha$ then
            Reject; reset current candidate solution: $x_k = x_{k-1}$
        end if
    end if
    Decrease $T$
end for
return $x_k$    // $x_{\text{best}}$
```

generations, or apply them only on a subset of the chromosomes in the generations. Of course strategies for improving efficiency can be combined. Which local search algorithm to use and how to construct viable neighbourhoods are, however, highly problem-dependent choices.

4.4 Projects

Three advanced families of optimisation problems are discussed in this section and it is explained why and in what sense these problems are hard (see §4.4.2).

The Travelling Salesman Problem in §4.4.1 (as introduced in Example 4.4) is a classic one: find the shortest route that visits all cities on a given list of cities.[9] There are many local minima and no algorithm for provably given a solution is known that does better than trying all possible routes. You will redefine chromosomes to represent a path between nodes. The second problem in §4.4.3 is the equidistribution of n repelling particles on a sphere. This problem is less well known than the Travelling Salesman Problem, but certainly not less challenging. You will redefine chromosomes to represent sets of vectors of real numbers. Combining GAs with traditional methods may be needed to obtain satisfactory results. Finally, in §4.4.4, variants of the Knapsack Problem of Example 4.4 are discussed, where the challenge is to find appropriate objective functions.

★ **Exercise 4.12:** *Project.*
Read the following subsections and choose one of the three families of problems as your main topic. These are hard optimisation problems of the type as discussed in §4.4.2. Do your experiments in a systematic way and don't hesitate to consult additional literature. ■

4.4.1 The travelling salesman problem

The *Travelling Salesman Problem* (TSP) is a notoriously difficult optimisation problem.

The distance $d(i,j)$ between any two cities S_i and S_j from a list $S_0, S_1, \ldots, S_n$ of $n+1$ cities (or locations) is known. In the so-called *Euclidean* variant of the TSP, $d(i,j)$ is simply the Euclidean distance between

[9]Obviously, the Travelling Salesman Problem is not just one problem, but a family of problems: there is a problem for each finite list of cities and each distance table with such a list. A similar observation applies to the other type of problems discussed in this Section 4.4. Often, in literature, with a problem one actually refers to a family of problems and a problem from the family of problems of interest is then referred to as an instance of a problem.

city S_i and city S_j: the cities S_j are located in a plane and have coordinates (a_j, b_j) and

$$d(i,j) \equiv \sqrt{(a_i - a_j)^2 + (b_i - b_j)^2} \qquad (i,j = 0,1,\ldots,n). \tag{4.7}$$

In other variants, we may have (one way streets and) $d(i,j) \neq d(j,i)$ for some i, j.

The assignment, the TSP, is to find a permutation $x_{\text{opt}} = x$ of $(1,\ldots,n)$ for which $f(x)$ with

$$f(x) \equiv \sum d(x(i), x(i+1))$$

(see (4.4)) is smallest; $f(x)$ is the length of the round trip described by the permutation x. Here, $x(0) = x(n+1) = 0$.

The related problem of finding in the network of the $n+1$ cities the shortest route between two cities, say S_{j_0} and S_{j_1}, is simple and routinely being (exactly) solved in route planners as TomToms using *Dijkstra's algorithm*, a realisation of the so-called *dynamic programming* strategy (where, step by step, solutions of subproblems are extended to solutions of larger subproblems; see Wikipedia [39] for more details). The TSP may look equally simple. Yet despite many efforts that have been made, if the candidate solution is required that is provably the best (x_{opt}), then we still cannot do much better than trying all possible combinations of cities.[10] This gives us an algorithm running in $\mathcal{O}(n!)$ time. This is very slow, if 24 cities would require one hour of computational time, then adding only two cities would take the computer one month.

We can try to find an appropriate candidate solution of the TSP by mapping bit strings (of sufficient length) to permutations (as indicated in §4.1.3) and then simply using our genetic algorithm. However, to allow exploitation of structure it may be more effective to represent chromosomes by permutations: create a number of random permutations of $\{1,\ldots,n\}$ (see Exercise 2.10) and evolve using the objective function $-f$. Yet, there is one catch. The definition of mutation and crossover needs some attention in order to obtain feasible tours, that is, to avoid double and missing cities in the tour. We solve this obstacle by introducing new mutation and crossover procedures.

Mutation of tours

For mutation we simply select randomly two cities from the tour x, say $S_{x(j_0)}$ and $S_{x(j_1)}$ $(j_i \in \{1,\ldots,n\})$, and swap the order in which they are visited: $\tilde{x}(j) \equiv x(j)$ if $j \notin \{j_0, j_1\}$, while $\tilde{x}(j_i) \equiv x(j_{1-i})$ for $i = 0,1$. Optionally you could evaluate the fitness before and after mutation, and keep the mutation only if the result is better. For large n it may be efficient to inspect the updates of $f(x)$: check that with

$$\left.\begin{aligned} \delta_i &\equiv d(x(j_i - 1), x(j_i)) + d(x(j_i), x(j_i + 1)), \\ \delta_i' &\equiv d(x(j_i - 1), x(j_{1-i})) + d(x(j_{1-i}), x(j_i + 1)) \end{aligned}\right. \qquad (i = 0,1)$$

we have that

$$f(\tilde{x}) = f(x) - \delta_0 + \delta_0' - \delta_1 + \delta_1'.$$

As a variant, you could swap not only the selected two cities, but all cities in between: $\tilde{x}(j_0 + i) \equiv x(j_1 - i)$ for $i = 0,1,\ldots,j_1 - j_0$.

There are other alternatives. For instance, select randomly one city from the tour x, say $S_{x(j_0)}$, and put it randomly at an other position in the tour and keep the rest of the tour intact: if $x(j_0)$ is put on position j_1 with, say, $j_1 > j_0$, then $\tilde{x}(j) \equiv x(j)$ for $j < j_0$ and $j > j_1$, $\tilde{x}(j) \equiv x(j+1)$ for $j_0 \leq j < j_1$, $\tilde{x}(j_1) \equiv x(j_0)$. Or more generally, select a part of tour x, say $S_{x(j_0)} \to S_{x(j_0+1)} \to \ldots \to S_{x(j_0+k)}$ and shift this part after city $S_{x(j_1)}$.

[10]Though a dynamic programming strategy here also helps to reduce costs, costs and memory requirements are nevertheless still prohibitive.

Finally all the above can be combined in some (random?) way.

Note, however, that swapping or shifting whole part of the tour may not fit the idea that a mutation is only a small change.

Crossover

Crossover is more complicated. One nice way of doing this is *greedy crossover* (from Grefenstette). This procedure continuously selects from the next two cities from the two parenttours the one that is closest to the current city for being visited next unless this city has already been visited. In that case, the other city is selected. If both cities have already been visited, then randomly an unvisted city is selected.

Study Algorithm 4.6, which details the procedure for the case of producing one child C. Cities have been identified with numbers (in $\{0, 1, \dots, n\}$). Creating another child can be done in a similar way (travel backwards?). You need to think of the implementation of $\mathrm{nextCity}$ and distance d yourself.

There are many variants. For instance, pick the first city randomly (of course the home city has to be included in the tour, but since the tour is cyclic, the home city can again become the zerost city by cycling the permutation),[11] or choose not only between the next two cities from the parenttours for being visited next but consider the preceding cities from the parenttours as well. The random choice in “select randomly the next city if the cities that are neighbouring in the parenttours have already been visited”, can be replaced by choose a city, not visited yet, that is closest to the current city (greedy choice).

Algorithm 4.6 Greedy crossover for the TSP creating one child.

```
// Creates child C from parents $P_1$ and $P_2$.
curCity=0;
$C \leftarrow ( )$        // (C is empty)
while $C$ not full do
    $\mathcal{V} \equiv \{\mathrm{nextCity}(P_1, \mathrm{curCity}), \mathrm{nextCity}(P_2, \mathrm{curCity})\}$
    Remove the values from $\mathcal{V}$ that are already in $C$.
    if $\mathcal{V} = \emptyset$ then
        newCity $\leftarrow$ Randomly select a non-selected city.
    else
        newCity $\leftarrow$ argmin$\{d(\mathrm{curCity}, j) \mid j \in \mathcal{V}\}$
    end if
    Add newCity to $C$:  $C \leftarrow (C, \mathrm{newCity})$
    curCity $\leftarrow$ newCity
end while
```

★ **Exercise 4.13:** *The Travelling Salesman Problem.*
Implement the TSP. Try to think of symmetries appearing when representing a tour as a permutation, and exploit those symmetries. First try only mutation, when this works, proceed with crossover. You can easily generate test problems for the Euclidean travelling salesman problem by randomly locating cities in a plane. You can find a wealth of test problems, with best known solutions at [34]. ■

★ **Exercise 4.14:** *Local search in the Travelling Salesman Problem.*
To increase the efficiency of your algorithm, implement local search in the main GA loop. A simple neighbourhood to start with is the following. Consider a single tour, or chromosome x and select a randomly a city, say $x(j)$, that is city $S_{x(j)}$. The first neighbour solution then is defined by removing $x(j)$ from the tour and inserting it before $x(1)$. The second neighbour solution is similarly defined from x by removing $x(j)$

[11] We actually identify tours as $S_5 \to S_4 \to S_1 \to S_0 \to S_3 \to S_2 \to S_5$ and $S_0 \to S_3 \to S_2 \to S_5 \to S_4 \to S_1 \to S_0$, i.e, permutations as $(5, 4, 1, 0, 3, 2)$ and $(0, 3, 2, 5, 4, 1)$ represent equivalent tours. By fixing the home city, we are allowed to drop the 0 from the permutations that represent tours, thus ruling out the possibility that different permutations can represent equivalent tours. However, it might be more efficient to maintain equivalent tours in a generation: detecting equivalency is not for free.

(the same j) and inserting it before $x(2)$, et cetera. The easiest local search method is greedy selection: evaluate the objective function for all neighbour solutions and select the best candidate solution. See how this method affects the quality of the candidate solutions, *as well as the computation time required.* Try to research other local search algorithm and neighbourhoods. ■

Exercise 4.15: *A representation of permutations allowing straightforward crossover.*
According to D.H. Lehmer, sequences $d = (d(0), d(1), \ldots, d(n-1))$ with $d(j) \in \{0, 1, 2, \ldots, j\}$ (in particular, $d(0) = 0$) can be identified with permutations of n elements. Prove this claim. (Hint: consider such a sequence d. Initiate x by putting $x = d$. Then apply the *Lehmer decoding* algorithm:

```
for $j = 1, \ldots, n-1$
    for $i = 0, \ldots, j-1$
        if $x(i) \geq x(j),\ \ x(i) \leftarrow x(i) + 1,$ end if
    end for
end for
```

Show that a) the resulting x permutes $(0, 1, \ldots, n-1)$ and b) the algorithm identifies sequences d with permutations of n elements. The inverse algorithm is the Lehmer encoding of permutations.) Note that the "standard" crossover strategies for bit strings can be applied to these sequences d: the restriction $d(j) \leq j$ is preserved.
The Lehmer decoding and encoding requires approximately $\frac{1}{2}n^2$ conditional checks, where for the encoding strategy for permutations in Section 4.1.3 only $n \log_2(n)$ checks are required. ■

4.4.2 NP-complete problems

As mentioned above, the TSP is notoriously difficult. In this section we will elaborate a little on what 'difficult' means. Note that the TSP actually is a family of problems: there is a problem for each n and for each configuration of $n + 1$ cities.

There is a simpler variant of the TSP: given a configuration of $n + 1$ cities plus a length M, does there exist a tour x of length at most M, i.e., is there an x such that $f(x) \leq M$? This is a '*decision problem*': the answer is simply a 'yes' or a 'no'. Note that all optimisation problems have a decision variant. Given a specific configuration of $n+1$ cities and a specific tour x, it can easily be checked whether $f(x) \leq M$. This requires $7n$ flop (floating point operations as $+, -, \cdot, \sqrt{}$): verification can be done in an amount of time that is proportional to the number n of unknowns. Computer scientist will state that verification is *polynomial in time*: the time needed is proportional to $P(n)$, with n the number of unknowns and P a polynomial with coefficients that are independent of n and of the configuration of cities (the same polynomial for all instances of the TSP), and where the proportionality only depends on the computer (that is, at its speed of performing flops and not on the type of problem). There are (other) families of problems for which verification is polynomial in time and for which the solution can be computed in polynomial time as well. For instance, solving a non-singular square linear systems $\mathbf{Ax} = \mathbf{b}$ of dimension n, requires at most $\approx \frac{2}{3}n^3$ flop. Here, $\mathbf{A}$ is a non-singular $n \times n$ matrix, $\mathbf{b}$ is a given n-vector and the equation has to be solved for an n-vector $\mathbf{x}$. There are $n \times n$ systems that can be solved with less flop: if, for instance, $\mathbf{A}$ is the identity matrix then no flop at all is needed to determine $\mathbf{x}$. However, the statement is that regardless the matrix $\mathbf{A}$ and the vector $\mathbf{b}$, it can be solved in at most $\frac{2}{3}n^3$ flop (neglecting terms of $\mathcal{O}(n^2)$) with n the number of unknowns (the number of coordinates of $\mathbf{x}$). That is in polynomial time ($P(n) = \frac{2}{3}n^3$). Verification, 'given $\mathbf{x}$, do we have that $\mathbf{Ax} = \mathbf{b}$?', can also be done in polynomial time (now $P(n) = 2n^2$).

There are families of problems for which verification is polynomial in time;[12] the family is said to be of

[12]To be precise, a decision problem is *verifiable in polynomial time* if there is a proof/algorithm, or more precise, a code that, for any candidate solution x, shows in polynomial time using a deterministic Turing machine that x satisfies the decision problem provided that x is a 'yes instance', that is, if x is a solution indeed (in case of the decision variant of the TSP this would mean $f(x) < M$. If $f(x)$ happens to be larger than M, then, for TSP, that can be checked in polynomial time as well. But in the definition of verifiable in polynomial time, verification of a 'no instance' is not required to be possible in polynomial time). A deterministic Turing machine is an idealised type of computer that, for instance, can handle integers of any size.

class $\mathbf{NP}$.[13] But for which it is *unknown* whether there exists an algorithm that can solve any problem of this family also in polynomial time: is the family of class $\mathbf{P}$?[14] Clearly $\mathbf{P} \subset \mathbf{NP}$.[15] Actually, it is one of the outstanding problems in Mathematics, whether for any family of decision problems with integer input,[16] solution is polynomial in time if verification is polynomial in time: is $\mathbf{P} = \mathbf{NP}$? There are one million dollars waiting for the person who can answer this question (with a proof or with a counter example). As explained above, the decision variant of the TSP (dTSP) is in $\mathbf{NP}$, but it is unknown whether it is in $\mathbf{P}$. (Actually, if dTSP is provable not in $\mathbf{P}$, then this would solve the '$\mathbf{P} = \mathbf{NP}$?' question.) It is considered to be among the hardest problems of class $\mathbf{NP}$. The class of problems in $\mathbf{NP}$ from the hardest kind is called $\mathbf{NP}$-complete.[17]

The optimisation variant of the TSP may even not be in $\mathbf{NP}$: it is unknown whether verification can be done in polynomial time. This makes the problem $\mathbf{NP}$-*hard*.

In the above discussion, we did not worry about the size of proportionality constant that is hidden in the statement that an algorithm needs polynomial time. It might be debatable whether, say, $10^{15}n^4$ is better than, say, $(1.02)^n$ for practical values of n.[18] Nevertheless, it will be clear that problems that can not be solved in polynomial time will be hard ones.

The conclusion is that an algorithm that can exactly solve any problem from a family of problems that is $\mathbf{NP}$-hard, as the TSP, or $\mathbf{NP}$-complete, as the dTSP, will require unacceptable long time. Therefore, we turn to stochastical methods and settle for a method that wil find relatively efficiently candidate solutions with function values that are probably (close to) the maximum. Note that genetic algorithms as well as simulated annealing only use the decision variant of the optimisation problem that is to be solved: in each step there is only to decide whether a member of a generation is 'fitter' than its fellow generation members. A disadvantage of these algorithms is the lack of viable convergence proofs and statements on convergence

[13]If, for an undetermined number of candidate solutions, checking can start at the same time and in parallel, and the decision problem happens to have a 'yes' answer, then that will be established in polynomial time (if the 'yes instance' is one of the candidate solutions that are to be checked, then that will give the 'yes' in polynomial time. At a 'yes' for one of the candidate solutions, checking on the others can be stopped.). An extension of this idealised deterministic Turing machine, a so-called nondeterministic Turing machine, can handle an undetermined number of candidate solutions at the same time. Therefore, if any 'yes' instance can be checked in polynomial time with a deterministic Turing machine, then the decision problem can be *solved in polynomial time* with a nondeterministic Turing machine. $\mathbf{NP}$ is an abbreviation of "nondeterministic polynomial time".

[14]Can the decision problem be solved in polynomial time using a deterministic Turing machine rather than a nondeterministic one?

[15]If the 'nondeterministic option' of a nondeterministic Turing machine is not used, then the machine performs as a deterministic one. Therefore, if a decision problem can be solved in polynomial time on a deterministic Turing machine then that can also be done on a nondeterministic Turing machine even without using its 'nondeterministic option'.

[16]Our TSP takes real inputs: the $d(i,j)$ are real numbers. But we would be equally happy if we could efficiently solve it for rational input: take the rationals that are up to machine precision equal to the reals. If the input is rational, then we can turn the TSP into an equivalent problem with integer input simply by rescaling the $d(i,j)$ by multiplication with a common multiple of the denominators of the $d(i,j)$, which can be done in polynomial time.

Note that the linear algebra problem of solving $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ may only fit in the integer setting if we allow errors (in the solution or in the residual: find an $\mathbf{x}$ such that the absolute value of all coordinates of the residual $\mathbf{b} - \mathbf{Ax}$ is less than ε). Then we are allowed to find a vector with rational coordinates close to $\mathbf{x}$. This also 'solves' the somewhat puzzling observation that, according to Galois, determining the eigenvalues λ of an $n \times n$ matrix $\mathbf{A}$ ($n > 5$) can not be done with finitely many flop, whereas checking whether a specific pair $(\lambda, \mathbf{x})$ of scalar λ and vector $\mathbf{x}$ is an eigenpair, i.e., $\mathbf{Ax} - \lambda\mathbf{x} = \mathbf{0}$, requires $\mathcal{O}(n^2)$ flop: it suggests that $\mathbf{P} \neq \mathbf{NP}$. However, the '$\mathbf{P} \neq \mathbf{NP}$?' question is on integer problems, while the eigenvalue problem is on real or complex data. Indeed, accurate approximations of eigenpairs can be computed in polynomial time; actually with approximately less than $n^3(6 + \log_2 \log_2(1/\varepsilon))$ flop if the absolute value of all coordinates of the residual $\mathbf{Ax} - \lambda\mathbf{x}$ of all approximate eigenpairs are only required to be less than ε times the absolute largest coordinate of $\mathbf{x}$, rather than being 0.

[17]The actual definition of a decision problem being in $\mathbf{NP}$-complete is different. Different formulations can be used for the same family, say Q, of problems. As a simple illustration consider the 'maxint problem' of solving $\max\{x \mid x \in \mathbb{N}, x < M\}$: for each $M \in (0, \infty)$ find the largest integer smaller M. This problem can also be reformulated as a special type of Knapsack Problem: for weights $w_j \equiv 2^{j-1}$ find the bit string $\mathbf{b}$ for which $W(\mathbf{b}) < M$ with $W(\mathbf{b})$ as large as possible. The family Q of problems is reformulated as a subfamily of a large family $\mathbf{Q}$ of problems. If the larger family is in $\mathbf{NP}$ (or in $\mathbf{P}$) and the operators that are needed for the reformulation can act in polynomial time, then, clearly Q is in $\mathbf{NP}$ (or in $\mathbf{P}$, respectively) as well. A family $\mathbf{Q}$ of problems in $\mathbf{NP}$ is said to be in $\mathbf{NP}$-*complete* if *any* family of problems in $\mathbf{NP}$ can be reformulated as a subfamily of $\mathbf{Q}$ using reformulation operators that can act in polynomial time. In particular, any family of two families of problems in $\mathbf{NP}$-complete can be reformulated as a subfamily of the other of the two: all families of problems in $\mathbf{NP}$-complete are, in some sense, equivalent. In particular, if one family of problems in $\mathbf{NP}$-complete would shown to be in $\mathbf{P}$ (or to be not in $\mathbf{P}$), then that would settle the '$\mathbf{P} = \mathbf{NP}$?' question.

[18]The eigenvalue problem of 'ε-accuracy', cf., Footnote 16, needs less than $n^3(6 + \log_2 \log_2(1/\varepsilon))$ time. Even with $\varepsilon = 10^{-16}$, the $\log_2 \log_2(1/\varepsilon)$ is modest. There are problems (cf., e.g., at the end of Section 4.4.4) where the ε of the floating point error shows up in the polynomial timing statement as $\mathcal{O}(n/\varepsilon)$. With $\varepsilon = 10^{-16}$ we have $\mathcal{O}(10^{16}\, n)$.

speed. To illustrate this observation, let us discuss a 'convergence statement' for simulated annealing (the global variant). Let $p(x)$ be the probability that, in the long run, x is visited by the simulated annealing algorithm ($x \in \mathcal{D}$). This means that, if $x_0, x_1, \ldots$ are the candidate solutions in the consecutive steps of simulated annealing, then, by the law of large numbers,

$$\#\{k < N \mid x_k = x\}/N \to p(x) \qquad (N \to \infty).$$

When T, with $T > 0$, is fixed throughout the process, then it can be proved that

$$\frac{1}{p(x)} = \sum_{y \in \mathcal{D}} \exp\left(\frac{f(y) - f(x)}{T}\right).$$

If, for instance, $\mathcal{D}_{\text{opt}} \equiv \{x_{\text{opt}} \in \mathcal{D} \mid x_{\text{opt}} = \text{argmax}_{y \in \mathcal{D}} f(y)\}$ contains one point only, say x_{opt}, and T is very small (such that $\exp([f(y) - f(x_{\text{opt}})]/T) \ll 1$ for $y \notin \mathcal{D}_{\text{opt}}$), then $p(x_{\text{opt}}) \approx 1$ and $p(x_{\text{opt}})$ will be approximately $\frac{1}{\ell}$ if $\mathcal{D}_{\text{opt}}$ contains exactly ℓ points: in the long run, it is very likely that we will find (one of) the bests x in $\mathcal{D}$. Unfortunately, in the 'long run' can be very very long (longer than $n!$ in case of TSP?), specifically if T is very small, while for larger values of T, it is less likely that the best x is visited.

4.4.3 Charges on a sphere

The problem in this section is about finding the optimal distribution of repelling point charges on a spherical shell. Suppose we have an ideal conducting spherical shell in $\mathbb{R}^3$ and we place n electrically charged point particles on this shell. Because of the repelling forces between the particles, they will all push each other away.

For higher values of n this is an extremely hard problem, the fitness landscape has many local minima. Actually, this problem is really a *sphere packing* problem in a spherical universe. Suppose the points are optimally separated. We can draw the largest possible circle around each point (on the sphere), such that no circles overlap and all radii are equal, then we are really trying to pack circles on a spherical shell. The Sphere Packing Problem in a rectangle is notoriously hard, so we can expect a challenging problem if we try to do this on a sphere.

We will now make the problem more mathematical, and point to several variations on the problem. Since it is conceptually not much more difficult we select an $d \geq 3$ and we pose the problem in $\mathbb{R}^d$ rather than in $\mathbb{R}^3$, where we consider the spherical shell

$$\mathcal{S}^{d-1} \equiv \{x \in \mathbb{R}^d \mid \|x\|_2 = 1\}.$$

Here,

$$\|x\|_2 = \|(x(1), \ldots, x(d))\|_2 \equiv \sqrt{|x(1)|^2 + \ldots + |x(d)|^2}$$

is the Euclidean norm of the point $x = (x(1), \ldots, x(d)) \in \mathbb{R}^d$. The distance between two points x and y on $\mathcal{S}^{d-1}$ is given by

$$d(x, y) \equiv \|x - y\|_2.$$

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a sequence of n points on the sphere $\mathcal{S}^{d-1}$: $x_i \in \mathcal{S}^{d-1}$ $(j = 1, \ldots, n)$. For some $s \geq 1$, we want to minimise the *total Riesz s-energy* $E_s(\mathbf{x})$,

$$E_s(\mathbf{x}) \equiv \sum (d(x_i, x_j))^{-s}, \tag{4.8}$$

where we sum over all different pairs (i, j), $i, j = 1, \ldots, n$, $j > i$. For $s = 1$, this is just the *Coulomb energy*.

There are some features that make this problem very hard to solve. First of all, you have a large number of symmetries, both rotational and reflectional. The symmetries in the problem imply that there is no unique solution. By fixing one particle at $(1, 0, \ldots, 0)$ you have eliminated a lot of symmetry, yet incurred no loss of generality. More generally, for $i < d$, you may take x_i of the form $(*, \ldots, *, 0, \ldots, 0)$ with $d - i$ zeros in the tail. Moreover, you may assume the ith coordinate of x_i to be non-negative for $i < d$ and you can reorder the x_j such that for $j \geq d$ the dth coordinates $x_j(d)$ are non decreasing.[19] Furthermore, for large values of n the fitness landscape has many local minima (increasing exponentially in n), which is problematic for any optimisation procedure.

In our definition of the distance we may choose from several values of s. In the case of $s = 1$ the problem is known as the *Thomson problem*. The related problem

'find a sequence $\mathbf{x} = (x_1, \ldots, x_n)$ of points in $\mathcal{S}^{d-1}$ that maximises $E_\infty(\mathbf{x})$',

where

$$E_\infty(\mathbf{x}) \equiv \min_{i,j} d(x_i, x_j) \tag{4.9}$$

can be viewed as a limit case for $s \to \infty$ of finding the minimal total Riesz s-energy. Here, we are interested in a situation where for every particle we want the closest particle to be as far away as possible. Where would you put the particles in the case of $n = 1, 2, 3, \ldots$? This 'ultrarepulsive' optimisation problem is the *Tammes problem*.

Implementation

From a genetic algorithmic point of view you need an encoding of points on S^{d-1}. You have a few possible options to do this. You could create a string that encodes n vectors of length d. In this case you make sure that the mutation and crossover procedures keep the norm of the vectors equal to one, or keep mutation and crossover as they are and rescale all vectors to length one afterwards.

Another possibility is to have an encoding that respects the constraint of the point being on the shell. For example, have the first $d-1$ coordinates arbitrary and calculate the dth as $\pm\sqrt{1 - \|(x_i(1), \ldots, x_i(d-1)\|_2^2}$, where i just denotes the ith point on the spherical shell. You then need one extra bit to choose plus or minus. The advantage is that mutation and crossover do not need to be modified to ensure that the requirement $\|x_i\|_2 = 1$ always holds. An example of a genetic algorithm approach to the Thomson problem may be found in [25].

Perhaps the best option is to employ hyperspherical coordinates. In this generalisation of polar and spherical coordinates we use one radial coordinate r, with $r = 1$ for our purposes, and $d - 1$ angular coordinates ϕ_j. The coordinates of a point $x = (x(1), \ldots, x(d)) \in \mathbb{R}^d$ may then be represented as

$$\begin{aligned}
x(1) &= r\cos(\phi_1) \\
x(2) &= r\sin(\phi_1)\cos(\phi_2) \\
x(3) &= r\sin(\phi_1)\sin(\phi_2)\cos(\phi_3) \\
&\vdots \\
x(d-1) &= r\sin(\phi_1)\sin(\phi_2)\ldots\sin(\phi_{d-2})\cos(\phi_{d-1}) \\
x(d) &= r\sin(\phi_1)\sin(\phi_2)\ldots\sin(\phi_{d-2})\sin(\phi_{d-1}),
\end{aligned} \tag{4.10}$$

where ϕ_1 is the angle between vector $\vec{x} \equiv (x(1), \ldots, x(d))^{\mathrm{T}}$ and the positive $x(1)$-axis, ϕ_2 is the angle between the positive $x(2)$-axis and the plane through $\vec{x}$ and the $x(1)$-axis, et cetera. Note that vectors $\vec{x}$ and points x can identified. Each coordinate is thus defined by $d - 1$ parameters $\phi_1, \phi_2, \ldots, \phi_{d-1}$. Think about appropriate ranges for the $d - 1$ angles.

★ **Exercise 4.16:** *Energies for the Thomson problem.*
In [12] you will find tables of the energy $E_1(n)$ corresponding to an optimal n-particle configurations for

[19]The zeros can be obtained in any candidate solution $\mathbf{x}$ by a proper rotation of the sphere. The ordering into non-decreasing last coordinate follows by switching the order of the x_j in $\mathbf{x} = (x_1, \ldots, x_n)$. Note that rotations of the sphere to not affect the Euclidean norm, while permutations of of the j-index does not affect $E(\mathbf{x})$.

the Thomson problem. Here,

$$E_s(n) \equiv \min\{E_s(\mathbf{x}) \mid \mathbf{x} = (x_1, \ldots, x_n), x_i \in \mathcal{S}^{d-1}\} \qquad (s \in [1, \infty]).$$

Verify some entries using the genetic algorithm. ■

Also try one of the following exercises:

Exercise 4.17: *Relaxation.*
For high values of n the problem becomes extremely hard to solve. One way to speed up the GA process is to insert a second optimisation procedure which makes sure that every member is at least a local minimum. In [12] a *steepest descent* procedure is described. The resulting algorithm is then as shown in Algorithm 4.7. Of course you may also use CG, Newton iteration, or any other optimisation technique. ■

Exercise 4.18: *Thomson vs. Tammes.*
One of the fascinating aspects of this problem is that for different values of s, different solutions emerge. Yet, for some small values of n, namely $n = 1, \ldots, 6$ and $n = 12$ the optimal Thomson and Tammes configurations are equal. Verify this and explore the differences in the solutions for several values of s. ■

Algorithm 4.7 GA combined with steepest descent.

```
while no convergence do
    Evaluate
    Select
    Crossover
    Mutate
    Perform steepest descent iteration
end while
```

4.4.4 The Knapsack Problem

We return to the Knapsack Problem of Example 4.3: objects O_j with weight w_j from a set of m objects have to be carried in a knapsack that can carry a total weight of at most M, M and w_j are in $(0, \infty)$.

Strings of m-bits describe selections (subcollections) of objects: if $\mathbf{b} = b_{m-1}b_{m-2}\ldots b_1b_0$ is such a bit string with $b_j \in \{0, 1\}$, then object O_j is selected if and only if $b_j = 1$ $(j = 0, \ldots, m-1)$. The total weight $W(a)$ of selection a is

$$W(\mathbf{b}) \equiv b_0w_0 + b_1w_1 + \ldots + b_{m-1}w_{m-1}.$$

If we want to maximise the total weight in knapsack (the *knapsack problem*), then we have to find the bit string $\mathbf{b}$ for which $W(\mathbf{b})$ is as large as possible, but such that $W(\mathbf{b}) \leq M$.[20]

The above problem is an instance of a more general variant, where each object O_j has a value, say v_j with $v_j \in (0, \infty)$, and the assignment is to find the subcollection of objects with largest value that the knapsack can carry without being destroyed:

$$\text{maximise} \quad V(\mathbf{b}) \equiv \sum_{i=0}^{m-1} b_i v_i \quad \text{such that} \quad W(\mathbf{b}) \leq M. \tag{4.11}$$

V is the objective function: the object is to maximise V under the restriction $W(a) \leq M$.

The (candidate) solutions $\mathbf{b}$ are bit strings. They naturally fit in our setting of genetic algorithms: crossover and mutation have been explained precisely for this situation. However, it is a bit tricky to define fitness

[20]For testing purposes, it might be convenient to simply define $M \equiv W(\mathbf{b})$ for some specific $\mathbf{b}$.

functions: $V(\mathbf{b})$ has to be maximised, but there should also be a severe 'punishment' on violating the restriction $W(\mathbf{b}) \leq M$. For this purpose, we may introduce a so-called *barrier function*, or *penalty function*, ϕ to prevent $W(\mathbf{b})$ from crossing M (or to make it hard to cross M); for instance,

$$\phi(t) \equiv 0 \ \text{ if } \ t \leq 0 \quad \text{and} \quad \phi(t) = -\infty \ \text{ if } \ t > 0.$$

Then, we can base our fitness functions on an adapted objective function f that arises by combining V with the barrier function:

$$f(\mathbf{b}) \equiv V(\mathbf{b}) + \phi(W(\mathbf{b}) - M) \qquad \text{for all } m\text{-bit strings } a.$$

The barrier '$\phi(t) = -\infty$ if $t > 0$' may be too 'hard' (kicking out nice candidate solutions that violate the restriction just a little). 'Soft' alternatives as $\phi(t) = -\beta\, t$ for $t > 0$ with $\beta > 0$ large (or 'softer' ones as $\phi(t) = -\beta t^2$ or 'harder' ones as $\phi(t) = -\beta\sqrt{t}$ for $t > 0$) may be more effective. The β should take into account a difference in scaling between the weights w_i and values v_i. The $\beta = \beta_i$ can be selected depending on the generation index i that is considered in step i of the genetic algorithm (for instance, such that $\beta_i \to \infty$ if $i \to \infty$).

We also have to find an appropriate barrier function if $v_j = w_j$ for all j. For the related problem, 'divide the load as fair as possible between two persons', it is easier to define an adapted objective function: with M half of the total weight of all objects, the assignment is to minimise $\max(2M - W(\mathbf{b}), W(\mathbf{b}))$ over all m-bit strings $\mathbf{b}$, or, equivalently, to

$$\text{minimise } |M - W(\mathbf{b})| \quad \text{where} \quad M \equiv \frac{1}{2}\sum_{i=0}^{m-1} w_i.$$

Here, we used the fact that, for this specific value of M, if $W(\mathbf{b}) > M$ and $\mathbf{a}$ is the bit string that arises by flipping all bits of $\mathbf{b}$, then $W(\mathbf{a}) = M - (W(\mathbf{b}) - M)$. This simpler problem can be used to get some feeling on what appropriate choices for the numerous parameters in the genetic algorithm might be for the knapsack problem (4.11).

The *check-sum* variant of the Knapsack Problem is a decision problem: given integers $w_0, w_1, \ldots, w_m$, is there a subset of this set of m integers that adds to 0? Or, formulated with the above notations, is there a selection, say $\mathbf{b} = b_{m-1}b_{m-2}\ldots b_1b_0$, such that $W(\mathbf{b}) = 0$? The problem is a simpler variant of the Knapsack Problem in the sense that the w_i are integers (rather than reals, but they can also be negative!) and we are not interested in the best solution but in a 'yes' or 'no' answer. Even the Check-sum Problem is hard; it is $\mathbf{NP}$-complete: verifiable in polynomial time, yet unknown whether it is solvable in polynomial time (cf., §4.4.2).

The decision variant of the Knapsack Problem, "is there a $\mathbf{b}$ such that $M - \varepsilon \leq W(\mathbf{b}) \leq M$?" is, in case of integer weights w_i and integer M, equivalent to the check-sum problem (in this case any $W(\mathbf{b})$ is an integer and therefore, the decision variant actually asks for the existence of $\mathbf{b}$ such that $W(\mathbf{b}) = M$: take $\varepsilon = \frac{1}{2}$. With $w_0, \ldots, w_{m-1}, -M$, this is a check-sum problem). If, in addition, M is fixed (i.e., for a fixed positive integer M, consider the subfamily of problems of knapsack problems, "for positive integers $w_0, \ldots, w_{m-1}$ is there a $\mathbf{b}$ such that $W(\mathbf{b}) = M$?"), then the problems are solvable in polynomial time: put

$$\phi(j,k) \equiv \max\{W(\mathbf{b},j) \mid W(\mathbf{b},j) \leq k\}, \quad \text{where} \quad W(\mathbf{b},j) \equiv \sum_{i=0}^{j-1} b_i\, w_i.$$

Then a table of the $\phi(j,k)$ values, for all $j = 1, \ldots, m$ and $k = 0, \ldots, M$, can easily be computed by recursive use of the relation

$$\phi(j,k) = \max(\phi(j-1,k), \phi(j,k-w_{j-1}) + w_{j-1}) :$$

fill the table by checking (i,ℓ) in a lexicographical way. This dynamic programming approach requires $\mathcal{O}(mM)$ flop. In case of rational weights and rational M, proper scaling leads to integer weights. Unfortunately, the scaled version of M can be extremely large. For instance, if the rationals have a decimal expansion of, say 16 digits, the scaling factor may be as large as 10^{16} and the scaled M will be like $10^{16}\,M$. So, even if M is fixed, the subfamily of knapsack problems with rational (certainly, with real) weights, might not be easily solvable.

4.5 Other related optimisation solvers

The methods that we discussed so far, Genetic Algorithms and Simulated Annealing, share a few characteristics. In each step a new candidate solution or set of candidate solutions is generated from the old one(s): the methods are iterative. The new candidate solutions are selected with some randomness, but with favouring the 'better' candidates. The randomness is required in order to avoid of getting stuck in candidate solutions that are only locally optimal. The 'favouring' is based on local information (better than nearby solutions, or better in this generation), but also on global information (better with respect to all candidate solutions considered so far [elitism]). The selection criteria mimic processes in real live: the forging of iron towards a crystal structure in a state of lowest energy in Simulated Annealing, the evolution of a population towards optimal fitness for certain circumstances in Genetic Algorithms. There are other optimisation methods that rely on the same principles. We briefly discuss two of them (they have not to be implemented). As for Genetic Algorithms and Simulated Annealing, these algorithms have to be completed with a stopping criterion and can be extended with a local search.

4.5.1 Ant colony optimisation

The ant system approach is inspired by the observation that a colony of ants manages to find the shortest route from the colony to a food source as soon as some ant(s) detects the food source. Ants in search for food wander around, but in selecting their way they favour trails taken by their sisters: ants leave a trace of pheromone, that can be detected by fellow ants. This pheromone evaporates in time. When an ant has to select a trail, the one with the highest concentration of pheromone is most likely to be selected by that ant. As an effect of this 'individual' selection mechanism, most of the ants end up at the shortest route. Why is that? Suppose there are a number of routes between the colony and the food source and suppose that initially each route is taken by the same number of ants. Since, within a fixed span of time, an individual ant will be able to travel the shortest route more often than a longer one (assuming all ants travel with the same speed), the shorter route will be travelled by more ants than the longer one. Therefore, the pheromone deposing strategy will lead to a higher concentration of pheromone on the shortest route. That, in its turn, will make more ants to decide to take the shortest route, which will lead to more pheromone on the shortest route, etc..

As an example of how to turn the above observation into a numerical solution strategy, let us consider the Traveling Salesman Problem for the cities $S_0, \ldots, S_n$. For more details, see, for instance, [9]. To have an iterative process, we turn the continuous movements of ants into discrete movements, that is, all our ants visit all cities in one step of the iterative process. (Of course, we have to model the fact that the shortest route can be travelled more often by the same ant than longer routes). The amount of pheromone will also be updated only once per step.

We work with A ants ($A \in \mathbb{N}$).
In each step of the iteration process
  1) the amount of pheromone $\tau(i,j)$ between the cities S_i and S_j is updated
  2) each of the A ants (again) selects a route for the traveling salesman, i.e., selects a permutation.

At step k, let $\tau(i,j)$ be the amount of pheromone on the road between city S_i and city S_j (on the edge (i,j) between the vertices i and j) and let x_a be the route (permutation) travelled by the ath ant. The amount of pheromone is updated as

$$\tau(i,j) \leftarrow (1-\rho)\tau(i,j) + \sum_{a=1}^{A} \delta(x_a, i, j),$$

where ρ is the evaporation rate of the pheromone (ρ is a fixed number in $(0,1)$) and $\delta(x_a,i,j)$ is the amount of pheromone laid by ant a on edge (i,j). We model this amount by

$$\delta(x,i,j) \equiv Q/f(x) \quad \text{if} \quad (i,j) \in T(x) \qquad \text{and} \qquad \delta(x,i,j) \equiv 0 \quad \text{otherwise.}$$

Here, $f(x)$ is the length of the round trip described by the permutation x, $T(x) \equiv \{(x(i), x(i+1)) \mid i = 0,\ldots,n\}$ is the collection of roads (edges) taken in this round trip x, Q is a fixed scaling constant. Note that $Q/f(x)$ models the fact that, in a fixed span of time, an ant can travel a short route more often than a longer route, thus will deposit more pheromone on any unit distance in a shorter route. In case of real ants, the length $d(i,j)$ of the road (i,j) will influence the amount of pheromone as well. Below, we will model this effect in the way the ants select their next trip.

At step k, ant a will select its next trip, i.e., it's next permutation $x = x_a$, as follows.
Suppose $x(1),\ldots,x(m)$ have been selected; of course, such that $x(p) \neq x(q)$ if $p \neq q$. Let $U(x,m)$ represent the collection of cities not yet visited in this trip by this ant: $U(x,m) \equiv \{1,\ldots,n\}\backslash\{x(1),\ldots,x(m)\}$. Now, with $i \equiv x(m)$, let $\tilde{p}_j$ for $j = 1,\ldots,n$ be defined as

$$\tilde{p}_j \equiv \tau(i,j)^\alpha \frac{1}{d(i,j)^\beta} \quad \text{if} \quad j \in U(x,m) \qquad \text{and} \qquad \tilde{p}_j \equiv 0 \quad \text{otherwise.}$$

Here α and β are fixed parameters, as $\alpha = 1$ and $\beta = 2$. Scale the $\tilde{p}_j$ to have probabilities:

$$p_j \equiv \frac{1}{T}\tilde{p}_j, \quad \text{where} \quad T \equiv \sum_{j=1}^{n} \tilde{p}_j.$$

Now, $x(m+1) = j$ with probability p_j, that is, the probability that the next city to be visited by ant a is city S_j equals p_j (you can use the roulette wheel to determine j: select a random number r in $[0,T]$ from a uniform distribution; j is such that $\sum_{\ell<j} \tilde{p}_j \leq r$ and $\sum_{\ell\leq j} \tilde{p}_j > r$).

With the parameters α and β, we can balance global information (contained in the amount $\tau(i,j)$ of pheromone) against local information (assuming that it is more attractive to first visit cities that are nearer: in literature on these type of methods, this is called 'heuristics').

In a variant, in the so-called the Max-Min ant system, only the most successful ant adds pheromone to the trails:

$$\tau(i,j) \leftarrow (1-\rho)\tau(i,j) + \tilde{Q}\,\delta(x_{\text{best}},i,j),$$

where x_{best} is the shortest route either detected by the ants at step k or the overall shortest route taken by the ants so far (up to step k) and $\tilde{Q}$ is some scaling constant. To avoid large or small values of $\tau(i,j)$ a min max is taken

$$\tau(i,j) \leftarrow \min(\max(\tau(i,j), \tau_{\text{min}}), \tau_{\text{max}}).$$

This approach highly favours to best route so far. The min max limitations leave room for other routes to be selected.

4.5.2 Particle swarm optimisation

Particle swarm optimisation is inspired by the behaviour of bird flocking and fish schooling.

Initially a collection of $A \in \mathbb{N}$ so-called 'particles' is selected, each particle at random position in solution space $\mathcal{D}$ with a random velocity: the ath particle is at position x_a and has velocity v_a. The particles form the 'swarm'. Each particle of the swarm 'flies' through the solution space. The velocity is updated at each (time) step (for simplicity, all time steps are of unit size). The adjustment of the velocity has some randomness, but it favours adjusting velocity in the direction of the particle's best position, say y_a, so far ($y_a = \text{argmax}_x f(x)$, where the maximum is taken over all positions x so far of the ath particle) and the swarm's best position b so-far ($b = \text{argmax}_x f(x)$, where the maximum is taken over all positions x so far of all A particles):

$$v_a \leftarrow \omega\, v_a + \phi\, r_a(y_a - x_a) + \phi_b\, r_b(b - x_a) \quad (a = 1, \dots, A).$$

Here, $\omega \in \mathbb{R}$ is a user-defined 'behavioural' parameter, called the inertia weight: it controls the amount of recurrence in the particle's velocity. The velocity $y_a - x_a$ towards the ath particle best position and the velocity $b - x_a$ towards the swarm's best position are weighted by stochastic variables r_a and r_b, random numbers from $[0, 1]$ with uniform distribution, and user-defined behavioural parameters ϕ and ϕ_b in $\mathbb{R}$, called acceleration parameters. The random numbers r_a and r_b depend on the coordinate: actually r_a and r_b are $m \times m$ diagonal matrices with random numbers in $[0, 1]$ on the diagonal. Here m is the dimension of the space $\mathcal{D}$: $\mathcal{D} \subset \mathbb{R}^m$. The idea is that each particle keeps track of its own best position as well as the one of swarm. Note that this approach assumes that the solution space is a vector space (allowing scalar multiplication and vector addition). These operators have to be adapted for other types of solution spaces. The updated velocity and the current location of the particle determines the new location of the particle:

$$x_a \leftarrow x_a + v_a \qquad (a = 1, \dots, A).$$

This step may require to update the particle best positions y_a (if $f(x_a) > f(y_a)$ then $y_a \leftarrow x_a$) and the swarm's best position b. Depending on the solution space $\mathcal{D}$ the velocities may have to be limited to prevent the particles from leaving the solution space.

For more details, and a list of values for the parameters ω, ϕ and ϕ_b that have successfully being used, see, for instance, [37].

Appendix A

Writing Reports

A.1 LaTeX

The report should be written in LaTeX, which has become the standard for scientific typesetting of documents. Some worthwhile resources on the web include:
`http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf`
`http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/`
`http://www.cs.cornell.edu/Info/Misc/LaTeX-Tutorial/LaTeX-Home.html`

A.2 Supporting materials

Do not include long (say more than 20 lines) C++ codes in your report. A good place for source code is an appendix at the end of your report. Code listings are only useful if you refer to them in your text. Remember that you will have to send the C++ source files to the instructor anyway.

For visualisation you can use MATLAB, but this is not obligatory. MATLAB has the ability to export figures to EPS (Encapsulated Postscript), which can be easily included in a LaTeX document. If you use `pdflatex`, convert the EPS pictures to PDF using:

```
$ epstopdf example.eps --outfile=example.pdf
```

A.3 Referring to sources of information

Collecting background information to describe your research is easier than ever these days. It is tempting to include literal fragments of text, e.g. from Wikipedia. However:

Plagiarism will not be tolerated!

With the ease of finding information online through search engines such as Google, comes the same ease of verifying whether a part of your report was not copied from online sources. Also, always be critical of information found online as it may contain errors.

Write your report in your own words, if you feel that quoting some short fragments literally is really necessary, use the `\begin{quote}..\end{quote}` environment in LaTeX.

Include all sources of information in your bibliography, and cite them at the appropriate places: an entry in your bibliography that is not cited anywhere in your report makes no sense at all.

A.4 Contents

The report should be aimed at master students that have not followed this course. This means the necessary theory should be covered, but you can assume some mathematical knowledge. Write in 'article style', use formal but clear sentences. Do not refer to the exercises in the handouts directly. Hence, never write things like "*we will now proceed with exercise 3.2*", but rather something like "*we now investigate the effect of ... on ...*". You don't have to include the result of every exercise in your report. The exercises marked ☞ are intended to point you to interesting aspects. The exercises marked ★ are required, though.

A good report contains at least:

- *An introduction*, topic and scope of the report (what are you researching and which questions do you answer), relevance in mathematics or other fields, possibly point to previous results in the literature, short outline of the report.
- *Theory*, the mathematical foundation.
- *Experiments*, use tables or graphs for clarity.
- *Conclusion/Discussion*, summarise the results of your study, maybe point to possibilities for further research.
- *Bibliography*, your sources (books, articles, internet).
- *Appendices*, for example for selected C++ source code.

A.5 Common mistakes

Below is a list of hints that were collected during previous editions of this course. Take advantage from it by not making the same mistakes in your own report.

Scientific content

- Motivate your choices for methods and experiments, for example by making a 'bridge' between what you did before and are about to tell the reader now. Do not use, e.g. "*Another method is discrepancy sampling, it works like this...* ".
  Instead use, e.g. "*The error's convergence order using random points can be improved significantly by using predetermined point clouds instead. This method, called 'discrepancy sampling', was originally...* ".
- Do not use "*it can be shown*" inappropriately: when some statement is nontrivial, either prove it, or provide a reference to work by others.
- Always discuss the results in tables and figures. The reader should not have to do that himself. Do not just repeat the number in tables, but lift out the results that are strange, or the most illustrative.
- Always have a 'Conclusion' section, either at the end of each topic (here: RNG and MC), or at the end of the entire report. Do not come up with new statements in your conclusion, but summarise all important findings from the preceding sections, both theoretical and—for this course—mainly experimental.
- When referring to other sources, do not include unnecessary ones. For example: when referring to definitions of expectation value, distribution functions, and other probabilistic quantities, refer to one good book. Do not refer to Wikipedia for one thing, and to a book for another thing, etc.

- Do not refer overly much to Wikipedia. Most content comes from other, more permanent sources; at the bottom of many Wikipedia pages is often a list of references. This goes in general as well; preferably cite articles or books instead of online material.
- "Less is more". Do not include theory that you are never using in your experiments, unless you want to make a theoretical point of course.

Layout

- In graphs, *do* use labels for your axes, *do* use a legend when necessary, and *avoid coloured lines* when you are printing in black only.
- Figures and tables should always have a caption, and are preferred to float, i.e. use `\begin{figure}...`, or `\begin{table}...`.
- Algorithms are nicely formatted using the `algorithmic` package. Also, give them a caption and make them float, using the `algorithm` package.
- Refer to figures, sections, etc. with a capital letter, e.g. *The results from Section 4.3 are shown in Figure 4.5.*
- When referring to websites, include a title *and* the date (month + year) visited.

Writing style

- Be formal: for example do not use *"If you want to..."*, but instead use *"Suppose we want to..."*.
- Avoid abbreviations such as *we're*, *don't*, etc. Use the full form *we are*, *do not*, etc. instead.
- Do not use abbreviations in chapter or section titles. Also, each abbreviation should be introduced once, before any other use of it. For example: *"The use of Random Number Generators (RNGs) to ... One class of RNGs is that of...."*.
- Use a spell checker. On UNIX, this can be done by:
  `ispell -t report.tex`. Optionally use British English by adding `-d british`.
- A final hint: when using external sources, e.g. papers or the course book, it is a well-known risk that your report turns out as only a slight rewriting of the original sources. When writing your report: do not continuously look at papers or the course book: make sure you understand things first, put the books aside, and then formulate things in your own words.

Appendix B

Some Useful C++ Material

B.1 The `#define` directive

Using `#define`, a named macro can be defined, that replaces any occurrence of that name anywhere in the source by a specified expression. For example:

```
#define QUICK 2
// ...
  switch (generator) {
      case QUICK:
      // ...
```

The use of '`QUICK`' will be replaced by '`2`'. This use of `#define` gives constant values a more intuitive name, hence abstracts from some technical details in the code.

B.1.1 Macros as conditionals

A very common use of `#define` is something like the following (in e.g. `mainprogram.h`):

```
#define USESPECIALTRICK
```

Any other file that is also compiled can now make use of this macro (e.g. `mainprogram.cc`):

```
#include "mainprogram.h"

double dosomething()
{
  double d;
#ifdef USESPECIALTRICK
  d = somespecialtrick();
#else
  d = 0.0;
#endif
  return d;
}
```

If, at compile time, the `USESPECIALTRICK` macro is set[1], the `#ifdef` condition evaluates to true, and the call to `somespecialtrick` will be active in the resulting program. The advantage of this approach,

[1] `#define` 'sets' a variable, even though it may not be with an explicit value.

is that a `USESPECIALTRICK` flag only needs to be set once in some header file, and that all other files have access to it. Besides, it allows the compiler to do more optimisations than when a normal boolean variable had been used (i.e. using `if(usespecialtrick) { ...}`).

Defined macros are fundamentally different from variables in C++. The macros are processed only once at compile time, by the compiler, and in essence have nothing to do with the actual C code.

Macros can also be undefined (at compile time only, that is), using `#undef NAME`. Also, one can define macros on the command line when calling the compiler by using the 'define'-flag:
`gcc -DUSESPECIALTRICK -c mainprogram.c`.

B.2 Variables and Pointers

A computer program uses the internal memory of a computer to store all kinds of values that it has computed and might need later on. Think of computer memory as an enormous row of boxes: each box has an *address*, and can contain one *value*. An address is needed when looking up a certain value: it specifies 'which box to look in'. A 'box' has a certain size, measured in units of *bytes*.

B.2.1 Normal Variables and Memory Space

When a variable is declared in a program, a bit of memory space is reserved to contain a value for that variable. It depends on the type of the variable (e.g. `char`, `int`, `double`) how much space is needed. A `char` fits exactly in a box of 1 byte, whereas a `double` needs a box of size 8 bytes to store one entire double precision number.

The smallest 'box' in computer memory is 1 byte big and it can contain 256 ($= 2^8$) different values. A variable of type `int` uses 4 bytes (i.e. $4 \cdot 8 = 32$ bit), so the number of different values [2] that it can contain is $256^4 = 2^{32} = 4294967296$. Consider the declaration of an integer variable:

```
int i;
i = 4;
i = i + 1;
```

After the declaration, `i` is equal to the 'box' reserved for this variable. It can directly be used for reading the value or assigning a new value to the variable `i`.

Addresses and Pointers Instead of directly reading from and writing to the 'box' `i`, it is also possible to get the address of the 'box', using the *address operator* `&`:

```
int *iptr;
iptr = &i; // iptr is for example 132070
```

The declaration of `iptr` additionally contains a `*`, which specifies that `iptr` is actually a *pointer variable*, or *pointer* for short. In essence, a pointer is only the address of the 'box' for this variable, it 'points' to it.

A pointer can be initialised by assigning it the address of a normal variable, using the address operator `&`, as was just shown. The other way around is also possible: consider the pointer `iptr` to an `int i` (see code sample above). Instead of using `i`, the value to which a pointer points can be found using the unary prefix 'dereferencing' operator `*`. It should be placed in front of pointers, or more generally in front of a memory address. The following two expressions are now equivalent:

```
     i = i + 1;
*iptr = *iptr + 1;
```

[2]An `unsigned int` ranges from 0 to 4294967295, a `signed int` from -2147483648 to 2147483647. Notice how the value 0 also needs a place, hence the maximum value is always 1 less.

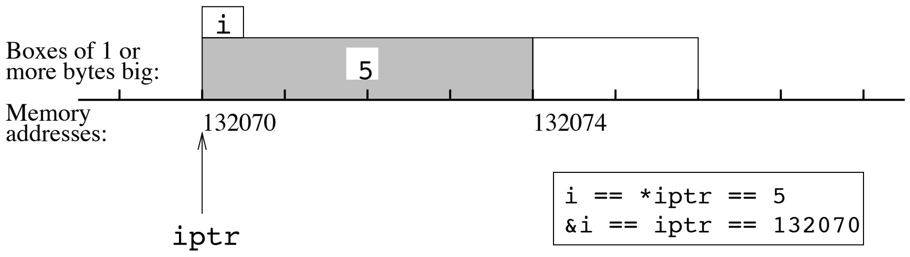

Figure B.1: Schematic representation of small piece of computer memory. A variable `int i` has an assigned value of 5, and pointer `int *iptr` points to `i`.

Note that a `*` in a declaration is different from the `*` operator in front of pointers. The first one is just part of the type specification of the declared variable, the latter accesses the value that is pointed to. To make it even more confusing, none of them has anything to do with the binary multiplication operator `*`.

An address is just a whole number, each byte has its own address, and the address of a 'box' is always the one of its first byte. Hence, in the last example, the starting address for `i` is 132070, and since `i` was declared to contain an `int` value (4 bytes), the next available address will be 132074. Bytes 132070 up to 132073 are occupied by the one and single box that contains the value of `i`. Figure B.1 shows this schematically.

B.2.2 Arrays and Addresses

Array variables are a list of 'normal' variables. They can be declared as follows:

```
int a[100];
```

A piece of memory is reserved for exactly 100 integers (i.e. $100 \cdot 4$ bytes). After this declaration, `a` is an expression containing the address of the first byte (of the first element, that is). In fact `a` is just a special pointer, pointing to the head of the array. A special construct that is used with arrays is the indexing operator `[ ]`:

```
a[0] = 3;
a[9] = 2;
a[4] = a[9] + a[0]; // a[4] is now 5
```

As shown, `a[i]` can be used for both reading and assigning array variables. Also note that counting starts at 0.

Instead of using the indexing operator, elements in the array can also be found by computing their address. This is summarised by the following two equivalences:

```
&a[i] == a + i;
 a[i] == *(a + i)
```

Remember that an `int` uses 4 bytes, so in the above first computation, to obtain the address of the i^{th} array element, actually $4i$ should be added, whereas now it states '`+ i`'. It is known, however, that `a` contains values of type `int`, so any computations that are purely performed with (pointer) addresses to `a` are *automatically* corrected to take into account the data element size.

B.2.3 Why and When to Use Pointers?

The pointer mechanism, addressing and pointer operators often seems complex to new users. Why not use normal variables as we always did?

Lasting updates of variables Imagine a function `intswap` that gets two `int`s `x` and `y` as parameters and should swap their values[3]. The following does not work as expected:

```
void intswap(int x, int y)
{
  int z;
  z = x; x = y; y = z;
}
```

Because, when calling this function, the parameters are 'passed-by-value', and become local variables in the function. Once `intswap` is finished, the swapped values are not available at the point where the function was originally called:

```
int v = 5, w = 7;
intswap(v, w);
// v and w have not changed at all!
```

Instead of getting the values to be swapped, the function should receive the addresses of the two variables that need to be swapped. It can then really swap the variables, and the effect lasts even when the function exits, as the variables are not local to the function. Here is the new working version:

```
void intptrswap(int *x, int *y)
{
  int z;
  z = *x; *x = *y; *y = z;
}
```

Notice how the variables are prefixed by the `*` operator, to assign and read their value (the contents of the 'box'). When calling the function `intptrswap`, not the variables should be used as parameters, but the pointers to them, or in other words: their addresses:

```
int v = 5, w = 7;
intptrswap(&v, &w);
// v and w have indeed been swapped.
```

'Returning' multiple values A function can return a computed value, using `return`. What if more than one value has been computed and needs to be returned? If they all are of the same type, an array could be returned, but in general this is not possible, or at least cumbersome.

When passing pointer variables to a function, the function has direct access to variables from entirely elsewhere in the program. It can change their values, which last even as the function exists. This is possible for any number of arguments, and the function now even has no need of returning anything. Consider for example the following function:

```
void grow(int *x, double *w, int *p)
{
  // Manipulate contents of x, w, and p directly...
}
```

Nothing needs to be returned, all manipulations are done directly in the memory pieces of the original parameter variables.

Save calling overhead When calling a function, the parameters are 'passed by value'. This means that a copy is made of each parameter and that this value is made available to the function. For simple types, this is no problem. When dealing with long (possibly multi-dimensional) arrays, this really starts to cost though. For example, one might work with large, dense matrices in an iterative algorithm. Instead of

[3]The example of swapping integers comes from [1].

passing the large data structures entirely and each time, a pointer to them is passed each time. Instead of passing thousands of bytes to each function call, now only several bytes are needed for the pointers. Pointers are featured in both C as well as C++.

B.2.4 Variable References

C++ features *references to variables* in addition to the pointer mechanics described earlier. Basically, references enable programmers to pass local variables to external functions *by reference*; that is, the function gets access to the original variable itself and does not copy its contents beforehand.

The C++ compiler may actually implement references by using pointers as described in the previous subsection, but it may opt to apply more advanced concepts when deemed necessary. Regardless of what the compiler decides, programmers have obtained a way to pass variables to functions without copying or using pointers, while maintaining the usual notations as if the referenced variables were just local variables; we do not need to bother with dereferencing (`z = *x` in the preceding `intptrswap` function), for example.

If we decide to pass a variable by reference, we precede it with `&` in the *function declaration*. This cannot be confused with taking an address of a variable, since that would make no sense in a function declaration. To illustrate, we repeat the examples from the previous subsection:

```
void intrefswap(int &x, int &y)
{
  int z;
  z = x; x = y; y = z;
}

...

int v = 5, w = 7;
intrefswap( v, w );
// v and w have been swapped.

...

void grow(int &x, double &w, int &p)
{
  // Manipulate contents of x, w, and p directly...
}
```

The above code will work as expected. Note again that the only thing we need to change to pass by reference is the function declaration; there is no need for dereferencing, nor for taking addresses of variables. This is now handled behind the screens, resulting in cleaner code.

This does *not* make pointers obsolete; since simple pointer arithmetic is possible, pointers are in a sense more powerful than references. Also, it is not possible to reference a non-existing variable, as can be done with pointers (`int *empty = NULL` is valid syntax). As such, some things can be done with pointers that in no way can be achieved by using references instead. In most cases where pointers and references can be used interchangeably though, references are preferred for readability.

When working on both C and C++ projects while hoping to share code between those, remember that C does not support references. Fall back to using pointers or prepare to invest in some code rewriting. On a sidenote it may prove useful to know that by contrast, the programming language Java handles *all* variables by reference.

B.2.5 Memory allocation

By declaring a variable, some memory space is automatically reserved for it. When declaring a pointer variable that is intended to point to an array, it depends on the length of the array how many memory needs to be reserved. Array variables specify this directly at the declaration:

```
int a[30]; // Automatically reserves memory for 30 integers
```

This is not always possible, since the length of an array may not always be the same, for example when it depends on user input:

```
int main(int argc, char **argv)
{
  int n = atoi(argv[1]); // Get number from command line
  int a[n];              // Discouraged; not the preferred way
}
```

In plain C89, the declaration of an array variable does not allow a variable to specify the array size. This has been made possible by the C99 standard, but most compilers still lack proper support for these so-called variable length arrays (VLAs)[4]. Use the `malloc` function in C to allocate arrays of variable length: `int *a = malloc(n*sizeof(int));` allocates an array a of length n. In C++, this can be done using the `new` operator. This operator returns a pointer to some newly reserved memory:

```
int main(int argc, char **argv)
{
  int *a;
  int n = atoi(argv[1]); // Get number from command line
  a = new int[n];        // This is 'C++ style'
}
```

All reserved memory by `new` is released when the program is ended. It is good practise to explicitly release memory before that by means of the `delete` operator. When used on array pointers, square brackets are placed behind it:

```
// Frees array variable from previous fragment
delete [] a;
```

Failure to delete allocated memory after use, is called a *memory leak*. When an application leaks a lot of memory, for example because a function which allocates but never deletes a temporary array is called many times, main memory may overflow with no longer used data causing the computer to slow down progressively as it swaps data to its hard drive. This often ultimately results in unresponsive systems.

B.2.6 What To Remember?

The preceding sections introduced pointer concepts including technical background. The main practical concepts that need to be remembered are summarised below:

[4] as of July 2008; in particular, the GNU C compiler we use still lists VLA support as broken (`http://gcc.gnu.org/c99status.html`).

`int *a`	declares a pointer `a` to an int.
`int &a`	declares a reference `a` to an int.
Operator `*`	The contents of ...
Operator `&`	The address of ...
`a = new double[n]`	Allocates ('reserves') memory for an array of `n` doubles.
`delete n`	Cleans up all allocated memory for a scalar variable pointer `n`.
`delete [] a`	Cleans up all allocated memory for an array variable pointer `a`.
For a function `void foo(int *x)`, call it with either a pointer variable: `foo(a)`,	
or the address of a normal variable: `foo(&i)`.	

B.3 Object-oriented features in C++

Here follows a quick introduction to various OO or C++ principles. More experienced OO programmers may skip this section, while others may find the information here useful when some class definition or usage seems unclear.

Plain C is a *procedural language*, which means that it consists of execution of procedures (functions) and the data flow between their executions. C++ can also be used in a procedural way, but is actually an *object oriented language*, which means that it consists of creation of objects and the message flow between these objects. This section will only shortly consider the use of objects as abstract data types. It uses a student administration system as intuitive, running example.

B.3.1 Classes are abstract data types

Built-in data types, such as `int`, are sometimes too simple to store the necessary data of a program. Consider a student, for example; we need to store personal information, such as name, date of birth, etc. Besides, we want to keep track of a student's followed courses and obtained results. The definition of a `class` for a student allows us to store the personal information in one abstract data type. An example class definition is shown on page 98.

Private and public Class members (variables and functions) can be declared as `private` or `public`[5]. Private variables can not be used from outside of the class definition, nor can private functions be called. The purpose of this access control, is to simplify the use of classes by other programmers. Given a class definition by someone, you only need to inspect all public variables and functions. The rest will consist of technical details 'under the hood' that should be of no interest to you. In the preceding code sample: we need no direct access to a person's variables, as the function `print()` can give us an informational printout (in practise though, functions such as `getName()` and `getAge()` would be very handy of course).

Constructors and destructors The *constructor function* is required in a class definition. It has the same name as the class, no explicit type in its declaration, and can have any number of function parameters. It will automatically be called upon creation of a new object of this class (see next section), and as such is a good place for initialising member variables.

[5]Class members can additionally be declared as `protected`, but we do not further discuss that here.

listing B.1 A sample class that could be part of a student administration system.

```
#include <string>
#include <iostream>
using namespace std;

class student
{
private:
  char *name;
  int age;

public:
  student(char *str, int age)
  {
    name = new char[strlen(str)];
    strcpy(name, str);
    this->age = age; // An explicit this-> is necessary, because
                     // local and member variable have same name 'age'
  }

  ~student()
  {
    delete [] name;  // Make sure *entire* array is destroyed
  }

  void print()
  {
    cout << "Student " << name << " is " << age << " years old" << endl;
  }
};
```

The *destructor function* has the name of the class prefix with a ˜ and no arguments, nor type. It will automatically be called upon destruction of an object (using `delete`, see next section). It is the best place to free any (large pieces of) memory that was reserved for member variables.

B.3.2 Using objects in a program

To use the new object oriented features we need to include the class definition and start creating objects. The code in Listing B.2 serves as running example in this section. The paragraphs hereafter discuss each part separately.

Creating objects To use the defined classes in our program, we need to call the *constructor function* of the class. The above sample shows two possibilities. The `new` operator, introduced in Section B.2.2 for creating arrays, is used to create a new object and return a pointer to it (`mike` points to a new `student` object). The constructor call `student("Mike", 30)` initialises the object. When directly declaring a variable as an object (instead of a pointer), the arguments to the constructor call are directly behind the variable name, e.g. `student cecilia("Cecilia", 25);`.

Calling functions on objects To call member functions of an object or an object pointer, use the `.` or `->` operator, respectively. This is also illustrated in the preceding code sample. Exactly the same operators

listing B.2 Example usage of the student class in Listing B.1.

```
#include "student.h"
int main()
{
  student *mike;                    // Just declare a student pointer
  mike = new student("Mike", 30); // Initialise the student object
  student cecilia("Cecilia", 25); // Declare and init. student object
  mike->print();                    // mike is an object pointer, use ->
  cecilia.print();                  // cecilia is an object, use .
  delete mike;
  delete &cecilia;                  // INVALID; pre-allocated data members
                                    //          are deleted automatically
}
```

can be used when accessing public member variables of an object.

Destroying objects Once an object is no longer necessary in a program, we would like to free the memory that it occupies. This is again done with the `delete` operator (see Section B.2.2). All extra steps that are needed for freeing member variables and more, are hidden in the destructor function, which will be automatically called when using `delete`. The argument to `delete` should be a pointer. Notice in the preceding sample how a pointer to `cecilia` is obtained with the `&` operator. This results in compiler-accepted code, but is *not* valid code; data-members in functions are pre-allocated on the *stack* while objects initialised with `new` are allocated run-time on the *heap*. Data on the stack is automatically destroyed upon function exit.

B.3.3 In-depth C++ examples

We will now illustrate the previous concepts and introduce more advanced ones using more in-depth examples. First we start off with visibility, followed by class extension, virtual functions, and finally static class members.

Class visibility

Consider the following example class:

```
class Example {
  private:
    int _id;
    Example() {};

  protected:
    string _name;
    void ProtectedFunc();

  public:
    void PublicFunc();
    Example( int id, string name ): _id(id), _name(name) {};
}; // Note the semicolon at the end of class declarations.
```

Obviously, we have declared here a class named Example, which has various *members*, such as `Example::_id` or `Example::PublicFunc()`. Globally, there are two different kinds of members; member *variables* (like `_id`) or member *functions* (like `PublicFunc()`). Class members can have different visibility settings. There are three different visibility settings; private, protected and public.

A member with private visibility setting can only be used by functions within the same class. One with protected visibility can also be used by classes *extending* the current class (see the next subsection). Public members can be used from any context. As an example we have the following.

```
int main() {                    //example program using the Example class
  Example instance1;            //INVALID: creating an instance of Example
                                //this way makes an explicit call to the
                                //default Example() constructor, which
                                //cannot be called since it is private!
  Example instance2(7,"abc");   //Valid: calls the public constructor
                                //Example( id, name ) which sets the member
                                //variables _id and _name to the supplied
                                //values 7 and "abc", respectively.
  int id2=instance2._id;        //INVALID: cannot read non-public variables.
  instance2._name="def";        //INVALID: cannot write either.
  instance2.ProtectedFunc();    //INVALID: cannot call non-public member
                                //functions either.
  instance2.PublicFunc();       //Valid: can call public member functions
                                //(and read and write to public variables).
  return 0;
}
```

Extending classes

Existing classes can be *extended* to display more specialised behaviour. Think for example of the classic case where we have a ‘Person’ class storing the usual data such as the surname, last name, address, et cetera. This can be extended by a ‘Student’ class which adds a data member storing the student number, for example. Extending in C++ works in the following way (we again use the Example class defined earlier):

```
class SubExample: public Example {
  public:
    void PublicFunc2();
};
```

This class SubExample which is a so-called *subclass* of Example, has exactly the same protected and public members as its base class Example. It does *not* inherit the private members of Example. Consider the following application code which displays some of the subtleties in inheritance.

```
int main() {                    //example program using the (Sub)Example class
  SubExample instance;          //Valid(!): While the default constructor was
                                //private in its base class, this was not ex-
                                //plicitly stated for the SubExample class;
                                //constructors are never inherited!
  SubExample inst2(7,"abc");    //INVALID(!): there is no
                                //         SubExample( int, string )
                                //constructor defined in SubExample.
  instance.PublicFunc();        //Valid; declared public in base class.
  instance.PublicFunc2();       //Valid; declared public in SubExample itself.
  instance.ProtectedFunc();     //INVALID: as expected.
  return 0;
}
```

In implementation of the `PublicFunc2()` member function of SubExample, we have the following.

```
void SubExample::PublicFunc2() { //Note the prefix SubExample:: here; C++
                                 //will not know you intended to implement
                                 //the member function of SubExample here
                                 //if this prefix is missing; it would simply
                                 //create a global function PublicFunc2().

  _id = 7;                       //INVALID: _id is private in Example.
  _name = "xyz";                 //Valid: _name is protected and thus accessible.
  ProtectedFunc();               //Valid: same as above.
}
```

Virtual functions

Consider an application where users can add students or employees to a database system. Suppose we have defined the following over-simplified classes to achieve this:

```
class Person {                   //Either a student or employee.
  public:
    std::string name;            //The person's name can be freely
                                 //read and changed.
    virtual int getNumber() = 0; //This will return either a student
                                 //or employee number.
};

class Student: public Person {
  private:
    Student() {}                 //Disable default constructor.

  protected:
    int _s_nr;                   //Student number, declared protected,
                                 //as we may later want to differentiate
                                 //between Master and Bachelor students,
                                 //for example. (Both still have a
                                 //student number and are students.)

  public:
    Student(int n): _s_nr(n) {}  //Constructor which sets _s_nr upon
                                 //Student declaration.
    virtual int getNumber();     //Return the student number here.
};

class Employee: public Person {
  private:
    Employee() {}                //Disable default constructor.

  protected:
    int _e_nr;                   //Employee number.

  public:
    Employee(int n): _e_nr(n) {} //Constructor which sets _e_nr upon
                                 //Employee declaration.
    virtual int getNumber();     //Return employee number.
};
```

and in implementation simply:

```
int Employee::getNumber() { return _e_nr; }
int Student::getNumber() { return _s_nr; }
```

We declared the `getNumber()` member function *virtual* because we are now able to do something like the following:

```
int main() {
  Person **cache = new Person*[3];      //An array where in which we will
                                        //store both Students and Employees
                                        //entries.

  cache[0] = new Student(101);
  cache[0]->name = "Jake";              //Add a Student named Jake with
                                        //student number 101.
  cache[1] = new Employee(3);
  cache[1]->name = "Jane";              //Add an Employee named Jane with
                                        //employee number 3.
  cache[2] = new Student(102);
  cache[2]->name = "Joe";               //Add another Student named Joe with
                                        //number 102.

  std::cout << cache[0]->name << ": "
            << cache[0]->getNumber()
            << ", "
            << cache[1]->name << ": "
            << cache[1]->getNumber()
            << ", "
            << cache[2]->name << ": "
            << cache[2]->getNumber()
            << std::endl;               //Print the array to screen.

  return 0;                             //Exit
}
```

The above program will print the following to screen:

```
Jake: 101, Jane: 3, Joe: 102
```

Thus it seems our application is able to determine *run-time* if a person is either a Student or an Employee, and call the right `getNumber()` function returning either `_e_nr` or `_s_nr`, whichever is appropriate. This is due to the virtual keyword. For virtual functions, the application will first check of which exact subtype a given class is, and will then call the appropriate specialised function. Of special note is the declaration

```
virtual int getNumber() = 0;
```

in the Person superclass. This defines a virtual function which is not implemented; indeed the entire point of this exercise was that we do not know what number to return since a person can be either a student or employee. We say that this function is *purely virtual*. A class, such as Person, which contains purely virtual functions is called an *abstract class*, since it cannot be instantiated; the following code is incorrect,

```
int main() {
  Person x;
  x.name="Joe";
}
```

because the compiler does know what it should do when `getNumber()` is called, regardless if this function is ever called.

B.3.4 Static members

Member functions in classes can be declared static. This results in members which are not dependent on instances of a class. This is again best illustrated with a simple example:

```
class StaticExample {
  public:
    static int StaticVariable;       //Static member variable
    static void StaticFunction();    //Static member function
    int normalVariable;              //Normal variable
    void NormalFunction();           //Normal function
};

int StaticExample::StaticVariable=10;   //Set initial value to 10
void StaticExample::StaticFunction() {} //Normal function declaration
void StaticExample::NormalFunction() {} //Normal function declaration

int main() {
  StaticExample instance;             //Create an example instance

  instance.normalVariable = 10;       //Valid.
  instance.NormalFunction();          //Valid.
  instance.StaticFunction();          //INVALID; static functions exist
                                      //independently from class instance!
  instance.StaticVariable = 7;        //INVALID; same reason as above.
  StaticExample::StaticFunction();    //Valid; static members are called
                                      //as such.
  StaticExample::StaticVariable = 7;  //Valid. Note that static variables
                                      //can have their values changed!
  instance.normalVariable =
    StaticExample::StaticVariable;  //Valid.

  return 0;
}
```

Endnote

This small introductory appendix is by no means a complete introduction to the OO-aspects of C++, nor to OO programming in general. The reader is encouraged to look up more extensive textbooks on either subject if interested. Online tutorials for C++ are in abundance, but generally are less informative than a solid textbook. Also note that the example codes given in this text may not be written in what is considered well-written C++; they mainly serve to illustrate the various subjects in an easy manner. For instance, one cannot recommend programming a database of students and employees at an university using a single array of Persons, as in Section B.3.3. Regardless, these short notes should prove useful to understand some basic OO ideas.

Good luck on your programming assignments!

Appendix C

RNG source code introduction

You will initially be implementing various types of mostly *linear congruential* (LC) Random Number Generators (*RNGs*). This is meant as a warming-up for developing more advanced applications in the area of Monte-Carlo integration and genetic algorithms, later during this course. These applications depend on a good RNG, which you will develop during the first few weeks.

You are supplied with a framework in which RNGs can freely be implemented. The code is written in *C++* and relies heavily on the object-oriented (*OO*) features of this language. While it is arguably a little more difficult to write code OO-style, its beneficial features will immediately become apparent when moving from implementing your own classes to (re)using them in other parts of your software; for example when using your RNGs to perform Monte Carlo integration.

The *header files* (`*.h`) regarding RNGs contain the class declarations; i.e., the class names, and all its local variables and functions along with their *visibility*[1], in short: the overall structure. If one wishes to *use* or *extend*[2] an existing class, one always has to include its .h-file. All functions which are not *purely virtual*[3] and all *static variables*[4] have to be implemented in its corresponding *source file* (`*.cc`).

Summary of supplied source files

RNGs

File name	Short description
`generators.h`	Global definition of various types of generators
`LC_RNGs.h`	Collection of linear congruential random number generators
`EX_RNGs.h`	Collection of external random number generators
`RNG_factory.h`	Class for easily constructing all of the above generators.
`util.h`	Collection of handy utility functions
`rand.cc`	The main program for generating random numbers.
`nonuniform.cc`	The main program for generating non-uniformly sampled random numbers.

[1]See Section B.3.3.
[2]See Section B.3.3.
[3]See Section B.3.3.
[4]See Section B.3.4.

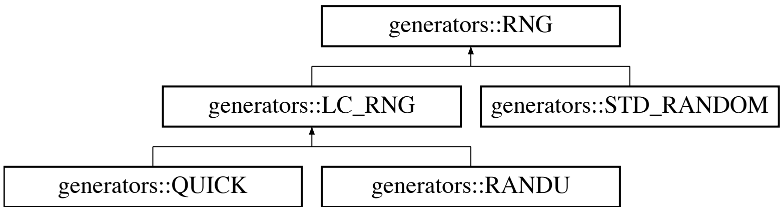

Figure C.1: Class hierarchy for several generators

```
class RNG {
    protected:
    unsigned int _seed;

    public:
    virtual unsigned int nextInt() = 0;
    virtual double nextDouble();
    virtual void setSeed( unsigned int seed );
    virtual unsigned int getMaximum();
    virtual ~RNG() {}
};
```

Figure C.2: Summarised version of generators.h

Monte Carlo

`discrep.h`	Class for multi-dimensional point generation, through discrepancy sampling
`sphere.h`	Contains a function which checks if a point is contained in an unit sphere
`hitormiss.cc`	The main program for integrating unit spheres with Monte Carlo.
`mc1d.cc`	The main program for one-dimensional Monte Carlo integration.

The following files make up the general RNG framework:

- `generators.h` and `generator.cc`
- `LC_RNGs.h` and `LC_RNGs.cc`
- `EX_RNGs.h` and `EX_RNGs.cc`

Figure C.1 shows the class hierarchy of some of the generator classes declared in these files.

generators.h/.cc

These files make up the base RNG classes, or more concrete, they define the basic RNG *interface*. Let us look at the .h-file as in Figure C.2. This simple file tells us each RNG has its own *seed*-value, and will generate a random integer when the `nextInt()`-function is called. `getMaximum()` returns the maximum integer the RNG can return and `getDouble()` returns a random number in $[0, 1] \subset \mathbb{R}$. Finally, the `setSeed( s )`-function sets the seed of an RNG to the supplied parameter s. Note that the classes defined in these files do not yet 'know' how to actually generate the random numbers; they only tell us which functionality a basic RNG should implement. Hence the RNG class is an *abstract* class.

As an extra, an abstract Random Vector Generator (RVG) is also defined (`RVG`), which provides an interface for generating random vectors $x \in [0, 1]^d \subset \mathbb{R}^d$. This can of course easily be done by generating d random numbers from an already existing RNG. This method is implemented in the form of the `RVG_RNG< X >` class[5], where `X` is an RNG class name.

LC_RNGs.h/.cc

Two implementations of an RNG are supplied in these files: a general linear congruential RNG (`LC_RNG`), and an *subclass* thereof, `LC_RNG_Schrage`. This class specialises `LC_RNG` further into a linear congruential RNG which employs Schrage's trick. As an example, a near-complete implementation of the Quick RNG is given. Your own LC RNGs should be defined in these files.

EX_RNGs.h/.cc

These files contain wrappers for externally defined RNG, such as the `rand()` function or `random()` function, commonly defined in standard C libraries. Over time, you will also be asked to implement wrappers for two of the RNGs in the GNU Scientific Library (*GSL*); these should also be declared in these files.

C.1 Application files

During the first part of the course, you will develop three applications, namely:

- `rand.cc`; An RNG front-end which produces n random numbers using the RNG `X`, where n and `X` are supplied run-time by the user. See Exercise 3.3 and 3.7 in the lecture notes.
- `nonuniform.cc`; a non-uniform RNG sampler. See Exercise 3.6 in the lecture notes.
- `mc1d.cc`; Integration of a simple function $f : [0,1] \to [0,1]$. Compares hit-or-miss and simple-sampling. See Exercise 4.3 in the lecture notes.
- `hitormiss.cc`; a hit-or-miss Monte Carlo integration application (calculates the volume of the unit sphere in $\mathbb{R}^d$). See Exercise 4.5 in the lecture notes.

The source files for these applications typically only contain the global flow of what the program should do; comment lines like `/*** ... ***/`[6] are used to indicate as to where you should write your own code to make the application work properly. All applications use the RNG classes as mentioned above; however, the RNG front-end and the hit-or-miss application will also use some additional classes which we will describe now.

RNG front-end: rand.cc

This application requires that a user can supply an RNG identification number to select the RNG to-be used. The `RNG_factory` class takes care of such an ID-to-RNG mapping, through the function `getGenerator( id )` which creates the requested RNG and returns a pointer to it. Note that the only function of `RNG_factory` is to construct and return specific RNGs based on user input; it is indeed an RNG factory. A factory is actually an example of a *design pattern* frequently used in OO. Many other design patterns exist, such as the singleton, the proxy or the iterator. The details of such patterns are not relevant to this course, but one should certainly be aware such patterns exist.

[5]This is a so-called *templated* class; a more advanced feature of C++. These will be used extensively in the second part of the course.

[6]Three stars; one star comments are used for disabling code blocks, two-star comments are used for documentation, and //-style comments are used for clarifying in-function statements.

1D Monte-Carlo integration: mc1d.cc

This application lets you compare hit-or-miss Monte Carlo with simple sampling. The sampled points simply come from the QUICK generator, or you can choose a different RNG which you expect to perform better. Here you will implement two MC integration procedures.

Monte-Carlo integration: hitormiss.cc

This application requires use of the RVGs described earlier. However, next to the RVG based on repeated use of a RNG, we may also make use of *discrepancy sampling* (Sec. 4.2.5 in lecture notes). This specialised RVG is to be found in the files `discrep.h` and `discrep.cc`.

C.2 Doxygen

All source files are heavily documented in so-called "javadoc-style". This style of commenting was originally introduced alongside the Java language, which included the javadoc utility able to convert documented source code to a HTML-based reference manual. Such a manual contains function descriptions, including information about any parameters supplied to it, and what it should return. *Doxygen* is a utility which does the same for C and C++ source. All source code supplied during this course will include HTML code documentation generated by doxygen, accessible via the course webpage [36].

Bibliography

[1] L. Ammeraal, *C++*, sixth ed., Academic Service, 2001.

[2] R. G. Brown, *Dieharder: A Random Number Test Suite*, http://www.phy.duke.edu/∼rgb/General/dieharder.php.

[3] Benoît C. Chachuat, *Nonlinear and dynamic optimization: from theory to practice*, vol. IC-31: Winter Semester 2006-2007, Laboratoire d'Automatique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2007.

[4] M. Cline, *C++ FAQ LITE*, http://www.parashift.com/c++-faq-lite/.

[5] D. A. Coley, *An introduction to Genetic Algorithms for scientists and engineers*, World Scientific, 1999.

[6] cpluscplus.com, *Reference: random*, http://www.cplusplus.com/reference/random/.

[7] R. E. Crandall, *Projects in scientific computation*, Springer, 2000.

[8] Josef Dick, Frances Y. Kuo, and Ian H. Sloan, *High-dimensional integration: the quasi-Monte Carlo way*, Acta Numerica (2013), 133–288.

[9] Marco Dorigo and Krzysztof Socha, *An introduction to ant colony optimization*, Tech. report, IRIDIA, Université Libre de Bruxelles, 2006, Technical report series, No. TR/IRIDIA/2006-10.

[10] B. Eckel, *Thinking in C++*, http://oopweb.com/CPP/Documents/ThinkingInCpp1/VolumeFrames.html.

[11] ———, *Thinking in C++*, http://oopweb.com/CPP/Documents/ThinkingInCpp2/VolumeFrames.html.

[12] T. Erber and G. M. Hockney, *Equilibrium configurations of N equal charges on a sphere*, Journal of Physics A: Mathematical and General **24** (1991), no. 23, L1369–L1377.

[13] N. Kohl et al., *C/C++ reference*, http://www.cppreference.com.

[14] W. Banzhaf et al, *Genetic Programming, an introduction*, Morgan Kaufmann Publishers, 1998.

[15] D. E. Goldberg, *Genetic Algorithms in search, optimization and machine learning*, Kluwer Academic Publishers, 1989.

[16] D. E. Knuth, *Seminumerical algorithms*, 3rd ed., The Art of Computer Programming, vol. 2, Addison Wesley, Reading MA, 1997.

[17] P. L'Ecuyer and R. Simard, *TestU01: a software library inANSI C for empirical testing of random number generators*, http://www.iro.umontreal.ca/∼lecuyer.

[18] N. M. Maclaren, *The generation of multiple independent sequences of pseudorandom numbers*, Appl. Statist. **38** (1989), 351–359.

[19] G. Marsaglia, *The marsaglia Random Number CDROM, with The Diehard Batter of Tests of Randomness*, http://www.stat.fsu.edu/pub/diehard.

[20] ———, *Random number generators*, Journal of Modern Applied Statistical Methods **2** (2003), 2–13.

[21] Mathworks, *Documentation center, MATLAB, mathematics, statistics and random numbers, random numbers generators, RandStream.list*, http://www.mathworks.nl/help/matlab/ref/randstream.list.html.

[22] The Mathworks, *Matlab*, http://www.mathworks.com.

[23] M. Matsumoto and T. Nishimura, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modelling in Computer Simulations (1998).

[24] Willam J. Moreokoff and Russel E. Caflisch, *Quasi-random sequences and their discrepancies*, SIAM J. Sci. Comput. **15** (1994), no. 6, 1251–1279.

[25] J. R. Morris, D. M. Deaven, and K. M. Ho, *Genetic-algorithm energy minimization for point charges on a sphere*, Physical Review B: Condensed Matter **53** (1996), no. 4, R1740–R1743.

[26] NAG, *NAG Library Manual: G05 - Random Number Generators*, http://www.nag.co.uk/numeric/fl/manual/html/G05/g05_conts.html.

[27] ———, *NAG Numerical Libraries*, http://www.nag.co.uk/numeric/numerical_libraries.asp.

[28] C++ Resources Network, *C++ tutorial*, http://www.cplusplus.com/doc/tutorial/.

[29] Jorge Nocedal and Stephen J. Wright, *Numerical optimization*, Springer series in operation research, Springer Verlag, New York, Berlin, Heidelberg, 1999.

[30] S. Park and K. Miller, *Random Number Generators: Good ones are hard to find*, Communications of the ACM **31** (1988), no. 10, 1192–1201.

[31] A. C. Planz, *C quick reference*, Que Corporation, 1988.

[32] R. Pooley, *C and C++ course*, http://www.macs.hw.ac.uk/∼rjp/Coursewww/.

[33] W. Press, W. Teukolsky, S. andVetterling, and B. Flannery, *Numerical recipes in C: the art of scientific computing*, second ed., Cambridge University Press, 1992.

[34] G. Reinelt, *TSPLIB*, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[35] K. Sigmon, *Matlab primer*, http://math.ucsd.edu/∼driver/21d-s99/matlab-primer.html.

[36] Scientific Computing Group Utrecht University, *Laboratory Class Scientific Computing course web page*, http://www.staff.science.uu.nl/∼sleij101>Lectures>Lab. Class Sc. Comp., 2015.

[37] F. van den Bergh and A.P. Engelbrecht, *A study of particle swarm optimization particle trajectories*, Information Sciences **176** (2006), no. 8, 937 – 971.

[38] B. A. Wichmann and I. D. Hill, *Generating good pseudo-random numbers*, Computational Statistics and Data Analysis **51** (2006), 1614–1622.

[39] Wikipedia, *Dynamic programming*, http://en.wikipedia.org/wiki/Dynamic_programming, 2015.

[40] ———, *Tabu search*, http://en.wikipedia.org/wiki/Tabu_search, 2015.