

Laboratory Class Scientific Computing
course book

Arno Swart¹

Arthur van Dam¹

March 22, 2006

¹Mathematical Institute, Utrecht University, P.O. Box 80.010, NL-3508 TA Utrecht, The Netherlands.
[http://www.math.uu.nl/people/\[swart,dam\]](http://www.math.uu.nl/people/[swart,dam]) Email: [swart,dam]@math.uu.nl, .

Contents

Contents	3
List of Exercises	5
1 Introduction	9
2 Common Techniques	11
2.1 System usage	11
2.1.1 Use a good shell!	11
2.1.2 Organise your files!	11
2.1.3 Use multiple workspaces!	12
2.1.4 Use a good editor!	12
2.2 Developing and running C++ programs	12
2.2.1 Programming in C++	12
2.2.2 Compiling C++ programs	13
2.2.3 Running the compiled program	14
2.2.4 Debugging your program	14
2.2.5 Writing your results to a file	15
2.3 Data visualisation with MATLAB	15
3 An Introduction to Random Number Generators	17
3.1 Survey of probability theory	17
3.1.1 Discrete random variables	18
3.1.2 Continuous random variables	18
3.1.3 Mean and Variance	19
3.1.4 The law of large numbers	20
3.1.5 Examples	20
3.2 Random Number Generators	21
3.3 Linear congruential generators	22
3.4 Implementation issues	23
3.4.1 Negative numbers (optional)	24
3.5 Non-Uniform distributions	24
3.5.1 The Rejection Method	25
3.5.2 Proof of the rejection method	26
3.6 Statistical tests	27
3.6.1 A more quantitative χ^2 test	28
4 Monte Carlo Integration	31
4.1 Conventional integration: quadrature formulas	31
4.1.1 Quadrature schemes in one dimension	31
4.2 Alternative integration: Monte Carlo	33
4.2.1 Hit-or-miss Monte-Carlo	33

4.2.2	The MC method using simple sampling	34
4.2.3	Importance sampling	34
4.2.4	Multi-dimensional Monte Carlo integration	35
4.2.5	Discrepancy Sampling	37
5	Genetic Algorithms	39
5.1	Introduction	39
5.1.1	Determining the fitness	40
5.1.2	Selection and reproduction	40
5.1.3	Crossover	41
5.1.4	Mutation	41
5.1.5	Stopping criteria	42
5.2	Implementation	42
5.3	Improvements	43
5.3.1	Elitism	43
5.3.2	Gray Codes	44
5.4	Projects	45
5.4.1	The travelling Salesman	45
5.4.2	Charges on a Sphere	47
A	Writing Reports	49
B	Some Useful C++ Material	51
B.1	The <code>#define</code> directive	51
B.1.1	Macros as conditionals	51
B.2	Variables and Pointers	52
B.2.1	Normal Variables and Memory Space	52
B.2.2	Arrays and Addresses	53
B.2.3	Why and When to Use Pointers?	54
B.2.4	Memory allocation	55
B.2.5	What To Remember?	55
B.3	Object-oriented features in C++	56
B.3.1	Classes are abstract data types	56
B.3.2	Using objects in a program	57
	Bibliography	59

List of Exercises

2.1	Writing and reading two dimensional data	16
3.1	20
3.2	Periods of poorly chosen iterations	22
3.3	Implementation of RNGs	24
3.4	25
3.5	25
3.6	26
3.7	Testing RNG's	29
4.1	Error of the trapezoidal rule	32
4.2	Error of the repeated trapezoidal rule	33
4.3	Integrating a function with Monte Carlo	35
4.4	Multi-dimensional hit-or-miss	36
4.5	The volume of the d -dimensional unit sphere	38
5.1	The genetic algorithm	42
5.2	Representing integers	42
5.3	Representing real numbers	43
5.4	Mutation	43
5.5	Selection and Crossover	43
5.6	Stopping criteria	43
5.7	Implementation of elitism	44
5.8	Elitism	44
5.9	Gray Code generation	45
5.10	Gray Codes	45
5.11	Project	45
5.12	The travelling salesman	47
5.13	Energies for the Thomson prblem	48
5.14	Relaxation	48
5.15	Thomson vs. Tammes	48

List of Algorithms

1	The main GA loop with selection and steady state reproduction.	41
2	Generating Gray codes of bitlength n	44
3	Converting Gray codes g to binary codes b	44
4	Crossover for the TSP, creates child one C_1 using parent one P_1	46
5	GA combined with steepest descent.	48

Chapter 1

Introduction

This book contains the material for the course ‘Scientific Computing Laboratory Class’ which is part of the Master of Science program in Scientific Computing at Utrecht University. This 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 subjects of Monte Carlo simulation and genetic algorithms.¹ The aim of the course is to learn about a number of important scientific computing 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: RNGs and Monte Carlo, 2: genetic algorithms) you have to write a report. Your final grade is the average of the grades for the two reports. You can work in teams of two persons, but every student is individually responsible for his/her *own* report. We have included some hints on writing a good report in Appendix A. The deadlines for the reports are published on the course web site [17]. You should also send your source code by email to the instructor, it must compile with the gcc compiler. If you have little experience with C++, then we recommend you team up with someone who has used C++ before.

This book The course book is set up as follows, Chapter 2 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 final report as well.

☞ A paragraph with a ☞ in the side margin is recommended. It usually contains hints to make life easier. Exercises marked as such are optional, but are interesting to have a look at.

Chapter 3 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 by writing small programs. The first main subject is Monte Carlo integration, presented in Chapter 4, 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 (e.g. trapezoidal rule).

¹This course is modelled after the 2003 course by Marciá Alves de Inda. Section 3.1 was taken from the course notes. We also thank Eduard Belitser for allowing us to use material from his course notes. In particular section 3.5 and Chapter 4 benefited from this.

The second subject is genetic algorithms, covered in Chapter 5. This is a technique for solving optimisation problems. We represent a set of possible solutions as a population of genes where some have greater probability of survival. Simulation of ‘natural’ evolution then yields a (possibly locally) optimal solution.

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

Chapter 2

Common Techniques

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

2.1 System usage

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

2.1.1 Use a good shell!

To get things done on Unix-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 ‘Terminal’.

☞ The program that handles the commands you enter, is called a ‘shell’. We suggest making `bash` your default shell from now on. To do so, you only need to run the following command just once:

```
$ 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.

2.1.2 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.
$ mkdir labsci
$ cd labsci
$ mkdir rng      ▷ First topic is random number generators.
```

2.1.3 Use multiple workspaces!

The CDE environment that runs on our lab machines offers you multiple workspaces, use them. If you don't, you will end up with ten or more windows cluttered on top of each other.

☞ At the centre bottom of your desktop is the **workspace manager** with buttons 'One', 'Two', etc. 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.

2.1.4 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:

```
Preferences > Default Settings > Statistics Line
Preferences > Default Settings > Show Line Numbers
Preferences > Save Defaults...
```

2.2 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.

2.2.1 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. They are not used for defining entire functions, but only assure that certain functions will be available. 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.

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:

- *C++*, Leen Ammeraal (in Dutch) [1].
- *De kleine C gids* (in Dutch), a handy quick reference for all standard C libraries. Also available in English (original) as ‘*C Quick Reference*’ [11]
- 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/Courseww/> [12].
- The ‘C++ Resources Network’ also has an excellent tutorial on C++. <http://www.cplusplus.com/doc/tutorial/> [9].

2.2.2 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 example2.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 example2.cc      ▷ example2.o is the default name
$ g++ example1.o example2.o -o exampleprogram
```

The advantage of this is that when one source file has changed (e.g. `example2.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 beginning. 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)'
```

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.

2.2.3 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.

2.2.4 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, probably something 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 program `gdb`. When necessary, this can be demonstrated during the laboratory sessions.

2.2.5 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 `print2file.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 `example.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, 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 generators.cc normalise.cc print2file.cc -o rand
$ ./rand 100
```

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

2.3 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 [16] for more information on the general usage of MATLAB.

You may invoke an instance of MATLAB using

```
$ matlab701 &
```

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 [7].

In the following we assume that you have a file `example.txt` containing some numbers, created as explained in section 2.2.5. 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('example.txt')
```

You will see two new windows named `Figure 1` and `Figure 2` with in them numbers from `example.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.

★ **Exercise 2.1: Writing and reading two dimensional data**

In this exercise you will expand on the one dimensional plotting function by adding 2D functionality. The printing routine is found in `print2file.cc`, as yet there is only a function `print2file1d` in the file. Add your own function `print2file2d` which outputs two columns of random numbers to a text file. Don't forget to update the header file `print2file.h`. When this works, write a MATLAB function that reads the 2-dimensional data and interprets each line as a coordinate. Use the old routine as a basis and expand on this! In a later exercise you will use this visualisation to locate correlations between random numbers. ■

Chapter 3

An Introduction to Random Number Generators

In this chapter we look at how to implement and test good random number generators. The topics covered here include: linear congruential generators, non-uniform random variables, and statistical tests. The generators developed in this chapter will be used in the Monte Carlo simulations as described in the next chapter.

Before starting with random number generators, the next section will review some basic probability theory.

3.1 Survey of probability theory

Probability theory is about the study of random processes. If we throw a dice, we don't 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 of points representing the possible outcomes of an experiment.
2. *Event*: A subset of a sample space.
3. *Probability measure* P : Is a real valued set function defined on a sample space Ω that satisfies
 - (a) $0 \leq P(A) \leq 1$, for every event $A \subseteq \Omega$.
 - (b) $P(\Omega) = 1$.
 - (c) $P(A_1 \cup A_2 \cup \dots) = P(A_1) + P(A_2) + \dots$ for every finite or infinite sequence of disjoint events A_1, A_2, \dots
4. *Complementary event*: For an event $A \subseteq \Omega$, its complementary event A^c is defined as: $A^c \equiv \Omega/A$. Its probability is $P(A^c) = 1 - P(A)$.
5. *Addition rule*: $P(A_1 \cup A_2) = P(A_1) + P(A_2) - P(A_1 \cap A_2)$.
 - (a) *Addition rule (a)*: $P(A_1 \cup A_2) = P(A_1) + P(A_2)$ if A_1 and A_2 are mutually exclusive events.
 - (b) *Addition rule (corollary)*: $P(A_1 \cup A_1^c) = 1$.
6. *Conditional rule*: The probability of A_1 given A_2 is defined as $P(A_1|A_2) = \frac{P(A_1 \cap A_2)}{P(A_2)}$.

7. *Multiplication rule*: $P(A_1 \cap A_2) = P(A_1)P(A_2|A_1)$.
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 $P(A_1|A_2) = P(A_1)$.

3.1.1 Discrete random variables

A *random variable* is a real valued function defined on a sample space. (We denote random variables by upper case letters from the end of the alphabet, e.g., X, Y, Z). A *discrete random variable* is a random variable that can take a finite or at most a countably infinite number of values. Let X be a discrete random variable. Then

1. the *frequency function* of X (also called *discrete density function* or *probability mass function*) is the function f defined by

$$f(x) = P(X = x).$$

2. The *cumulative distribution function (cdf)* of X is defined by

$$F(x) = P(X \leq x) = \sum_{t \leq x} f(t).$$

The cdf satisfies

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

3.1.2 Continuous random variables

A *continuous random variable* is a random variable that can take a continuum of values. Let X be a continuous random variable. Then

1. a *density function* of X is a function f that possesses the following properties

- (a) $f(x) \geq 0$,
- (b) $\int_{-\infty}^{\infty} f(x)dx = 1$, and
- (c) $\int_a^b f(x) = P(a \leq X \leq b)$.

Note that

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

2. The *cumulative distribution function (cdf)* of X is defined by

$$F(x) = P(X \leq x) = \int_{-\infty}^x f(t)dt.$$

The cdf satisfies

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

If f is continuous at x then

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

The cdf can be use to compute probabilities

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

Example 3.1 (Uniform random variable)

The cdf of the uniform random variable on the interval $[a, b]$ defined above is

$$F(x) = \begin{cases} 0, & \text{if } x \leq a, \\ \frac{x-a}{b-a}, & \text{if } a \leq x \leq b \\ 1, & \text{if } x \geq b. \end{cases} \quad (3.1)$$

♦

3.1.3 Mean and Variance

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

$$E(X) = \sum_{i=0}^n X_i P(X = X_i), \quad (3.2)$$

where the X_i range over all events in the sample space and possibly $n \rightarrow \infty$. The mean is not defined for distributions 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 die then $E(X) = 1/6(1 + 2 + 3 + 4 + 5 + 6) = 3.5$, which nicely illustrates that the expectation is not necessarily a realisable event. Do not confuse the mean with the *average* defined by

$$\bar{X}_N = \frac{1}{N} \sum_{i=0}^N X_i,$$

where the X_i are realisations of a random variable, generated according to some distribution. There are however theorems that relate the average to the mean. For continuous random variables, distributed according to the distribution function $f(x)$ we define the mean as

$$E(X) = \int_{-\infty}^{\infty} x f(x) dx, \quad (3.3)$$

if the integral exists. We now collect several facts on expectations in one theorem,

Theorem 3.1

If X is distributed according to $f(x)$ and $Y = g(X)$ then

$$E(Y) = \sum_x g(x) f(x), \text{ or,} \quad (3.4)$$

$$E(Y) = \int g(x) f(x) dx, \quad (3.5)$$

in the case that X is discrete respectively continuous. Furthermore, for independent random variables X and Y it holds that

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

Finally we mention that the expectation is linear, if $Y = b + \sum_i a_i X_i$ then,

$$E(Y) = b + \sum_i a_i E(X_i).$$

♦

Of course, the above statements only hold if the sums and integrations involved converge.

We now turn to the variance defined by

$$\text{Var } X = E([X - E(X)]^2). \quad (3.6)$$

The variance measures the deviation from the average. Often the standard deviation σ is used instead of the variance σ^2 . The variances is *not* linear,

$$\text{Var}(aX + b) = a^2 \text{Var}(x),$$

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

Theorem 3.2

The variance exists if $E(X^2) < \infty$, for finite sample space, and can then also be written $\text{Var}(X) = E(X^2) - [E(X)]^2$. \blacklozenge

\clubsuit **Exercise 3.1**

a) The k -th moment is defined as

$$m_k(X) = E(|X|^k),$$

show that the k -th moment is equivalent to the k -norm $\|x\|_k^k$ of the vector x containing the X_i .

b) Use equivalence of norms to show that

$$E(|X|) \leq cE(|X|^2),$$

for finite c . Can you find c explicitly?

c) Conclude that the variance exists if $E(|X|^2)$ exists and show that $\text{Var}(X) = E(X^2) - [E(X)]^2$. \blacksquare

3.1.4 The law of large numbers

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

Theorem 3.3 (Law of Large Numbers)

Let X_1, \dots, X_i be independent random variables with given mean $E(X_i) = \mu$. Let the average $\bar{X}_n = n^{-1} \sum_{i=1}^n X_i$. Then we have, for any $\epsilon > 0$,

$$P(|\bar{X}_n - \mu| > \epsilon) \rightarrow 0,$$

when $n \rightarrow \infty$. \blacklozenge

Or, phrased in words, the probability that the average differs from the mean tends to zero if the sample becomes large.

3.1.5 Examples

Example 3.2 (Bernoulli random variables)

A Bernoulli random variable takes the values 0 and 1 with probability $1 - p$ and p respectively. Its frequency function is

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

This can be written as

$$f(x) = \begin{cases} p^x(1-p)^{1-x}, & \text{if } x = 0 \text{ or } x = 1, \\ 0, & \text{otherwise.} \end{cases} \quad (3.7)$$

The *indicator random variable* of an event A , χ_A ,

$$\chi_A(\omega) = \begin{cases} 1, & \text{if } \omega \in A, \\ 0, & \text{otherwise.} \end{cases} \quad (3.8)$$

is a Bernoulli random variable. (Note the similarity with the Kronecker delta). \blacklozenge

Example 3.3 (Binomial random variables)

The binomial distribution arises as the number of successes of an experiment where there is a fixed probability of failure $1 - p$ and a fixed probability of success. Let n be the number of trials and let X be the number of successes, then X is a random variable with

$$f(x) = P(X = x) = \binom{n}{k} p^k (1 - p)^{n-k}. \quad (3.9)$$

The expectation is then

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

which may be evaluated by noting that X can be written as a sum of Bernoulli random variables Y_i ,

$$X = \sum_{i=0}^k Y_i,$$

and $E(X) = np$ follows directly from the above formula since $E(Y_i) = p$. The variance is computed using the same trick, we know that $\text{Var}(Y_i) = p(1 - p)$ and we use a theorem stating that

$$\text{Var}\left(\sum_{i=0}^n Y_i\right) = \sum_{i=0}^n \text{Var}(Y_i),$$

if the Y_i are independent to arrive at $\text{Var}(X) = np(1 - p)$. \blacklozenge

3.2 Random Number Generators

A random number generator (RNG) is a means for obtaining (uniformly distributed) random numbers. A roulette table, for example, is a 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 3.6).
- **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*).
- **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 number to minimum.

A computer based RNG is often called a 'pseudo random number generator' (PRNG), since the computer is completely deterministic and can therefore never produce true random numbers. Usually a PRNG uses an iteration 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*. In the next section we examine a particular choice for f .

3.3 Linear congruential generators

A popular choice for f leads to the *linear congruential generator*,

$$f(x) = (ax + c) \bmod m. \quad (3.10)$$

The modulo operator is a binary operator on two numbers: $a \bmod b$. The result is the remainder after division of a by b . For example $7 \bmod 2 = 1$, since $7 = 3 * 2 + 1$ and $5 \bmod 3 = 2$ since $5 = 1 * 3 + 2$. What is this good for? Let's count modulo three to see what happens:

$$\begin{aligned} 0 \bmod 3 &= 0, \text{ since } 0 = 0 * 3 + 0, \\ 1 \bmod 3 &= 1, \text{ since } 1 = 0 * 3 + 1, \\ 2 \bmod 3 &= 2, \text{ since } 2 = 0 * 3 + 2, \\ 3 \bmod 3 &= 0, \text{ since } 3 = 1 * 3 + 0, \\ 4 \bmod 3 &= 1, \text{ since } 4 = 1 * 3 + 1, \\ 5 \bmod 3 &= 2, \text{ since } 5 = 1 * 3 + 2, \\ 6 \bmod 3 &= 0, \text{ since } 6 = 2 * 3 + 0. \end{aligned}$$

We see that counting modulo three imposes an upper bound on the maximum number. Put it another way, the number system has in some sense become 'circular'. Mathematically you write $\mathbb{Z}/p\mathbb{Z}$ for the positive numbers modulo p (for example $\mathbb{Z}/3\mathbb{Z} = \{0, 1, 2\}$). Now since calculating $\bmod m$ gives us only m different numbers, it must be the case that after m steps the iteration $x_{i+1} = f(x_i)$ has repeated itself. We say the generator has a cycle of period n when $n = \min\{p | p = |i - j|, x_i = x_j, i \neq j, i, j > 0\}$.

For special choices of a, c, m in equation (3.10) we get efficient RNGs. Usually you want to work with $y = x/m$ which is in $[0, 1)$.

☞ Exercise 3.2: Periods of poorly chosen iterations

Take $a = 3, c = 4, m = 60$ and iterate (3.10). What is the period? You have to take care that, even for large m the period can become small. ■

Some values for the parameters that have been suggested in the literature (for example see [13]) are:

"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	$a = 1103515245$	$c = 12345$	$m = 2^{31}$

The "Park-Miller" generator was proposed in the classic paper [10], it is intended as a *minimal standard generator*. The generator has full period, the generated random numbers are highly statistically independent and an implementation is possible on any machine (independent of architecture). For a correct implementation Schrage's trick is needed, which will be discussed later.

Another popular class of generators are the *generalised feedback shift register* (GFSR) methods. Bits of members of the sequence are combined as $x_n = x_{n-p} \otimes x_{n-q}$ where $p > q$ and \otimes is a bitwise 'exclusive or' operation¹. We will not go into details, but the magic numbers $(p, q) = (250, 103)$ work very well.

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

The linear congruential generators can produce very large numbers. It might happen that the product ax becomes larger than the computer can handle. Let's make this more explicit. For efficiency we work with integers and the biggest integer we have is the `long signed int`. This data type has 32 bits of data, 31 for the value and one for the sign, thus we have the range $[-2^{31}, 2^{31})$. Now, how does the computer handle products of numbers that result in more than 32 bits? It throws away the leading *most* significant bits. This is nice for our RNGs since the least significant bits are harder to predict and can be considered to be somewhat 'more random'. It is now also the case that the sign might change, after truncation the leading (sign) bit has about equal chance of being 0 or 1. If our numbers can get negative we also need another procedure for normalising our random numbers to $[0, 1)$, division by m won't do the trick. Think about how you would normalise the random numbers. Also note that in 32-bit arithmetic the $\bmod 2^{32}$ is for free (see next section) if we use `unsigned long int`'s, I called this the 'quick' generator in the above table.

The practice of letting the numbers overflow is of course a bit fishy. We are no longer really iterating (3.10). Fortunately there is a trick to do $(ax) \bmod m$ exactly in 32-bits even if ax overflows. This is *Schrage's algorithm* and it is based on approximate factorisations:

$$m = aq + r, q = \lfloor m/a \rfloor, r = m \bmod a.$$

Schrage proved that, if $r < q$

$$ax \bmod m = a(x \bmod q) - r \lfloor x/q \rfloor$$

if this is ≥ 0 . If not, add m .

Note the restriction $r < q$, this makes Schrage's algorithm of limited use. However, this algorithm does allow us to implement the Park-Miller generator.

3.4 Implementation issues

Now, let's discuss the binary number system. Integer numbers in the computer are stored as bits, zeros and ones. A positive number x is represented in *binary* as $x = b_n b_{n-1} \dots b_0$ where the bit $b_i \in \{0, 1\}$. It is based on a representation by powers of two:

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

For example the number 23 is represented by 00010111 since $23 = 2^0 + 2^1 + 2^2 + 2^4$. The number of bits limits the largest number that can be stored, an n -bit positive number is evidently in the interval $[0, 2^n)$.

Now, how is addition done by the computer when the result is larger than the number of bits available? The *most significant* bits are thrown away. Thus $128 + 130 = 2$ in eight bit arithmetic, because $10000000 + 10000010 = 100000010$ but the leftmost (most significant bit) does not fit and is discarded and the result is 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 is of the type `unsigned long int` which is 32 bits, addition is thus modulo 2^{32} . For multiplication we have the same result, you can check this for yourself. The relevance for random number generation is of course the following: when calculating $f(x_{i+x}) = (ax_i + c) \bmod 2^{32}$, using `unsigned long int`, the modulo 32 operation is *for free*.

Now we introduce the C++ shift operators ' \ll ' and ' \gg '. The command $a \ll b$ 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 position are 'pushed out' and disappear. Therefore $128 \ll 1$ equals zero in 8-bit arithmetic. The ' \gg ' operator is a completely analogous shift to the right, yielding a 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 $(1 \ll 31) - 1$ is a quick way to obtain the largest possible 'signed long int'-number. The compiler gives you a warning message, but that's ok, we created the overflow on purpose.

Note that when $c = 0$ and the modulus is not equal to 2^{32} you can 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 evaluate $(ax_i + c) \bmod m$ and your RNG is probably not efficient anymore.

★ **Exercise 3.3: Implementation of RNGs**

Implement the generators from the previous section in C++. Sample code is available from my home page to get you started. Implement the 'quick' generator using `unsigned long int`'s. Use Schrage's trick for the other generators. If you have time, try the quick and dirty approach without Schrage's trick. ■

3.4.1 Negative numbers (optional)

Working with signed integers is a bit more tricky. In this case the most significant bit is used as a *sign bit*. If it is zero, the remaining 31 bits give us a positive number 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. 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. However, the modulus are no longer for free now. The best way to handle negative numbers is probably to work with positive numbers initially and normalise to a negative number.

3.5 Non-Uniform distributions

There are two main methods for generating non-uniform distributions from uniform distributions. The *inversion method* is the most straightforward solution, for which we need the following definition

Definition 1 (Generalised Inverse)

The generalised inverse of a distribution function F is defined by $F^{-1}(q) = \inf\{u : F(u) \geq q\}$ for $0 < q < 1$. ◆

A distribution function is always monotone increasing, yet need not be continuous nor strictly monotone. Figure 3.1 clarifies how the generalised inverse for such functions is defined. Before stating the main theorem we prove the following lemma,

Lemma 1

For all $x \in \mathbb{R}$ and $q \in (0, 1)$ we have

$$q \leq F(x) \leftrightarrow F^{-1}(q) \leq x.$$

◆

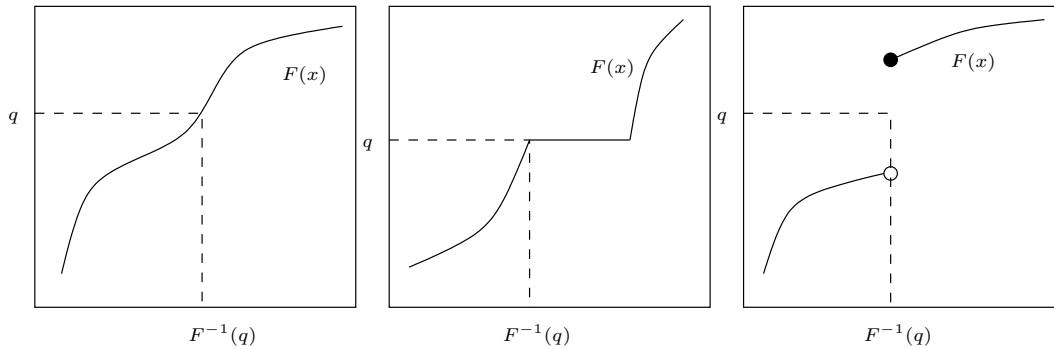


Figure 3.1: The generalised inverse for a continuous strictly monotone function (left), a continuous not strictly monotone function (middle) and a discontinuous function (right)

Proof. Firstly, suppose that $q \leq F(x)$. Now, $F^{-1}(q)$ has the smallest value among all x for which $F(x) \geq q$ (by definition). So certainly $q \leq F(x) \rightarrow x \geq F^{-1}(q)$. The other way around, suppose that $F^{-1}(q) \leq x$. We will prove $F(F^{-1}(q)) \geq q$, from which the result follows by $F^{-1}(q) \leq x \rightarrow F(F^{-1}(q)) \leq F(x) \rightarrow q \leq F(x)$. Note that in the first implication we used the fact that distribution functions are non decreasing. Define $S_q = \{u : F(u) \geq q\}$, then there is a sequence $\{u_\epsilon\}, u_\epsilon \in S_q, u_\epsilon > F^{-1}(q)$ and $u_\epsilon \rightarrow F^{-1}(q)$ as $\epsilon \rightarrow 0$. From the definition of S_q , $q \leq \lim_{\epsilon \rightarrow 0} F(u_\epsilon) = F(F^{-1}(q))$. \square

Now the inversion method is easily stated.

Theorem 3.4 (Inversion Method)

Suppose U is a uniformly distributed variable on $[0, 1]$. The stochastic variable $X = F^{-1}(U)$ has distribution function F . \blacklozenge

Exercise 3.4

Prove Theorem 3.4 yourself. \blacksquare

Exercise 3.5

Verify the following table analytically. The number U is from the uniform distribution on $[0, 1]$.

Distribution	Dist. Function	Inverse
Exponential	$F(x) = 1 - e^{-\lambda x}, (x, \lambda) > 0$	$F^{-1}(U) = -\frac{1}{\lambda} \log(U)$
Cauchy	$F(x) = \frac{1}{2} + \frac{1}{\pi} \arctan(x/\sigma)$	$F^{-1}(U) = \sigma \tan(\pi(U - 1/2))$

3.5.1 The Rejection Method

Sometimes we cannot calculate the inverse explicitly. In this case we can use another popular method called *the rejection method*. We give the algorithm below, and offer a proof in the next subsection. Suppose we want variables from a distribution density function p and suppose we can find a density function q for which

$$p(x) \leq cq(x), \text{ for some } c, \text{ for all } x. \tag{3.11}$$

The idea is that we choose a $q(x)$ from which it is easy to generate samples. The rejection method now works as follows

- Generate two variables, X from q and U from the uniform distribution on $[0, 1]$.

- Check whether $cUq(X) \leq p(X)$.
- If this is true we accept X as our random variable.
- If this is not true, we try again from the first step.

If we can easily find a density $q(x)$, then this is a nice and straightforward method. Note that to generate X according to probability function $q(x)$, the inversion method now *can* be used, since we choose q such that its primitive can be inverted by hand.

Example 3.4

A dominating density function $q(x)$ can be easily found in the case that the density $p(x)$ is bounded by M on an interval $[a, b]$. The simplest dominating density one can think of is $q(x) = (b - a)^{-1}$ (note that we need $\int_a^b q(x) dx = 1$). If we choose $c = M(b - a)$ then (3.11) becomes $p(x) \leq M$ which is true. The algorithm is now easy

- Generate X and Y from the uniform distribution on $[0, 1]$.
- Set $X \leftarrow a + (b - a)X$. Now X is uniform on $[a, b]$
- Check if $YM \leq p(X)$.
- If this is true we accept X as our random variable.
- If this is not true, we try again from the first step.

◆

★ Exercise 3.6

Program the procedure of Example 3.4 for a few bounded density distributions $p(x)$ on $[a, b]$. Generate non-uniform random numbers and test if the result is properly distributed by plotting bar plots. ■

3.5.2 Proof of the rejection method

As discussed in the previous section, the rejection method obtains samples from a nonuniform distribution π on (a subset of) $(-\infty, \infty)$, with probability density function $f(x)$. This is done by taking a sample X from Q on (a subset of) $(-\infty, \infty)$ and a sample U from $U[0, 1]$ (the uniform distribution on $[0, 1]$). The variates U and X must be independent, which means that for intervals A and B

$$P(X \in A, U \in B) = P(X \in A)P(U \in B).$$

Let X have pdf $q(x)$ and let U have pdf $\chi_{[0,1]}$, where $\chi_{[a,b]}(x)$ is the indicator function which is one if $x \in [a, b]$ and zero otherwise. The function c times $q(x)$ should have a simple form and be above $f(x)$ for some fixed c , that is

$$cq(x) > f(x), x \in Q.$$

Now we generate X and U , if

$$cUq(X) \leq f(X) \tag{3.12}$$

then X is accepted as a sample from π . Why does this work? Start by noting that the joint density function $f(x, u)$ of (X, U) can be written as

$$f(x, u) = q(x)\chi_{[0,1]}(u),$$

by independence of X and U . We now write the conditional probability of $X \leq t$, given that the sample is accepted, as

$$P(X \leq t | cUq(X) \leq f(X)) = P(X \leq t | U \leq \frac{f(X)}{cq(X)}).$$

We use the definition of conditional probability to obtain

$$\begin{aligned}
 P(X \leq t | U \leq \frac{f(X)}{cq(X)}) &= \frac{P(X \leq t, U \leq \frac{f(X)}{cq(X)})}{P(U \leq \frac{f(X)}{cq(X)})} \\
 &= \frac{\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} q(x) \chi_{[0,1]}(u) \chi_{\{u \leq \frac{f(x)}{cq(x)}, x \leq t\}}(x, u) dx du}{\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \chi_{[0,1]}(u) \chi_{\{u \leq \frac{f(x)}{cq(x)}\}}(x, u) dx du} \\
 &= \frac{\int_{-\infty}^t \int_{-\infty}^{\frac{f(x)}{cq(x)}} q(x) \chi_{[0,1]}(u) du dx}{\int_{-\infty}^{\infty} \int_{-\infty}^{\frac{f(x)}{cq(x)}} q(x) \chi_{[0,1]}(u) du dx} \\
 &= \frac{\int_{-\infty}^t \frac{f(x)}{cq(x)} q(x) dx}{\int_{-\infty}^{\infty} \frac{f(x)}{cq(x)} q(x) dx} \\
 &= \frac{\int_{-\infty}^t f(x) dx}{\int_{-\infty}^{\infty} f(x) dx}.
 \end{aligned}$$

For any pdf we know that $\int_{-\infty}^{\infty} f(x) dx = 1$, and we conclude

$$P(X \leq t | cUq(X) \leq f(X)) = \int_{-\infty}^t f(x) dx.$$

We see that X has pdf $f(x)$ and therefore is from π , which concludes the proof.

To make this procedure efficient we need a simple distribution Q to take samples from. For practical purposes we can simply take Q to be the uniform distribution on $[0, 1]$ and take $c = \sup f(x)$. Think of $f(x)$ as a graph and $q(x) = c$ as the horizontal line lying above it. The pair (X, U) is then a coordinate which may lie under $f(x)$ (\rightarrow accept) or between $f(x)$ and c (\rightarrow reject). The only reason for choosing $q(x)$ different from the constant function $q(x) = c$ is to reduce the number of rejections, thereby making the procedure more efficient. See figure 3.2 for a schematic representation.

3.6 Statistical tests

We will 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 if $x_i \in [0, 1/4)$, we go right if $x_i \in [1/4, 1/2)$, etc. If the walk repeats its structure after a number of steps then we have detected a period in the sequence.
- **Uniformity:** Every value x_j should occur with equal chance. We test this by creating M empty bins (i.e. variables that act as counters) and increase the value of bin i if a sample $x_j \in [i/M, (i+1)/M)$. The bar-plot of the bins should be approximately flat, and get flatter with increasing sequences of random numbers.
- **Correlations:** Seemingly random numbers can have subtle correlations. For example only in the last few bits. A nice way of visually finding correlations is by plotting points $(x_i, x_{i+1}), (x_{i+2}, x_{i+3}), \dots$ in a plane. Clustering or some structure hints at correlations in the numbers.

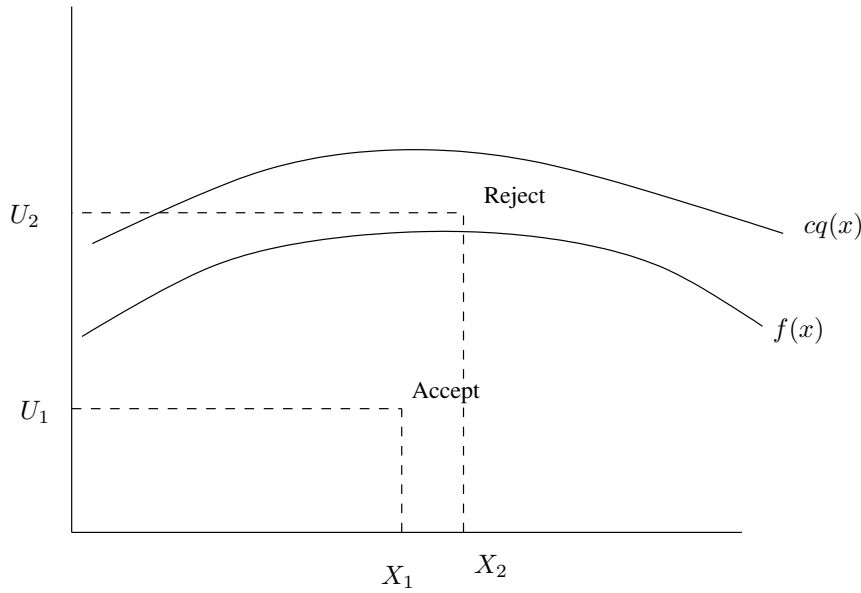


Figure 3.2: Two pairs of random deviates (X_1, U_1) and (X_2, U_2) have been generated. One is accepted, it passes (3.12), and the other is not. It is clear that $q(x) = 1$ would enlarge the area between $f(x)$ and $cq(x)$ and would lead to more rejections.

- **χ^2 -test:** This test measures the deviation of the observations from the statistical expectation. The number χ^2 is defined by

$$\chi^2 = \sum_{i=1}^M \frac{(y_i - E_i)^2}{E_i},$$

where y_i is the observed number in bin i (as defined above) and E_i is the expectation for bin i . The smaller χ^2 , the better. The next subsection will give more information on the χ^2 -test and its implementation.

3.6.1 A more quantitative χ^2 test

A small value of χ^2 indicates a good RNG. In this section we will elaborate on what 'good' means, and introduce the notion of a *confidence level*. Just as in the previous section we group our random numbers in M bins. Suppose we draw N numbers and the i -th random number goes in bin j if $(j-1)/M \leq x_i < j/M$. Let y_j be the number of x_i in bin j . Let E_j be the expected value of y_j , which is easily seen to be N/M . The χ^2 test is based on the fact that $(y_j - E_j)$ is distributed according to the χ^2 distribution function,

$$P(x, \nu) = \frac{1}{2^{\nu/2} \Gamma(\nu/2)} \int_0^x t^{(\nu-2)/2} e^{-t/2} dt.$$

The parameter ν is the number of degrees of freedom, which in our case is $M-1$. You might have thought we have M degrees of freedom in our system, but keep in mind that we draw N numbers and therefore $\sum_{j=1}^M E_j = N$, which is a constraint. The gamma function is defined as

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt. \quad (3.13)$$

For positive integers z the gamma function reduces to the factorial: $\Gamma(z+1) = z!$.

The more useful quantity is $Q(x, \nu) = 1 - P(x, \nu)$. The distribution function Q gives the probability

that the measured value of χ^2 exceeds x . In order to use this interpretation we need a number of measured values of χ^2 . Given a value of ν we can set a value for q and solve $Q(x, \nu) = q$ for x . Usually these values are taken from statistical tables. At the *confidence level* q we then expect x to be exceeded only $100(1 - q)$ percent of the time.

So suppose you want 95 percent confidence ($q = 0.05$), you look up the value of x in a table and find x_0 , you check if less than 5 percent of your χ^2 measurements exceeds x_0 . If this is true you have 95 percent confidence in the ability of your random number generator to generate uniform deviates. If this is not true, try some lower confidence level.

★ **Exercise 3.7: Testing RNG's**

Test the generators you wrote in (3.3) with the criteria of the previous section. Find confidence levels for several generators. Also try some really bad generators for comparison. Do you prefer some RNG over an other? Sample MATLAB code for visualisation can be found at the course web page [17].

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. ■

Chapter 4

Monte Carlo Integration

This section discusses Monte Carlo integration, which is a means of computing integrals using statistical averages. We can put the RNGs we developed in the previous chapter to good use here. Three subjects are covered: Monte Carlo, importance sampling and discrepancy integration. The discussion of discrepancy integration and the exercise on multi-dimensional integration is taken from [3]. The first section below considers a conventional integration technique: quadrature formulas. It also introduces the general integration problem that will be handled by Monte Carlo in the following section.

4.1 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 for higher-dimensional problems, quadrature (then called: *cubature*) schemes take exponential time, and the domains can be complicated and possibly non-convex.

4.1.1 Quadrature schemes in one dimension

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

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

$$\int_a^b f(x) dx. \quad (4.1)$$

The primitive function of $f(x)$ may be hard to find analytically, so we try to find a numerical approximation for the integral.

One of the simplest methods is to approximate $f(x)$ by a simple function $p(x)$ that is easily integrated. The *trapezoidal rule* uses a linear interpolation between a and b :

$$p(x) = \frac{x-b}{a-b}f(a) + \frac{x-a}{b-a}f(b),$$

such that the integral is approximated by:

$$\int_a^b f(x) dx \approx \int_a^b p(x) dx = \frac{b-a}{2}(f(a) + f(b)). \quad (4.2)$$

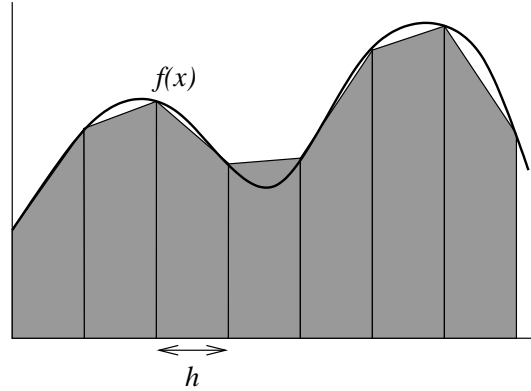


Figure 4.1: Schematic representation of the repeated trapezoidal for approximating the integral of a function. All intervals have equal size h . Notice the large errors that are made by the linear approximation in some intervals.

The error of this approximation is equal to the remainder

$$R = \int_a^b f(x) dx - \frac{b-a}{2}(f(a) + f(b)) = \frac{1}{2} \int_a^b (x-a)(x-b)f''(\xi(x)) dx. \quad (4.3)$$

If $f \in C^2$, i.e. is twice differentiable, this remainder is equal to

$$R = \int_a^b f(x) dx - \frac{b-a}{2}(f(a) + f(b)) = -\frac{(b-a)^3}{12}f''(\eta), \quad (4.4)$$

for some point $\eta \in [a, b]$.

☞ Exercise 4.1: Error of the trapezoidal rule

Derive the expression for the remainder value in (4.3). Next, give proof for the remainder value in (4.4).

HINT: Use the fact that the integrand in (4.3) 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 (4.3). ■

The error in (4.4) is usually too big. We can easily decrease it by dividing $[a, b]$ into n subintervals. We assume they all have equal size h , i.e. $nh = b - a$. Define the discrete points as $x_i = a + ih$ ($i = 0, \dots, n$). On each interval, we apply the trapezoidal rule (4.2). The approximation for the integral (4.1) by the *repeated trapezoidal rule* then becomes:

$$T(h) = h \cdot \left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n) \right] \quad (4.5)$$

Figure 4.1 shows the repeated trapezoidal rule schematically. The overall error is obtained by summing all remainders:

$$R(h) = -\frac{1}{12}h^3 \sum_{i=1}^n f''(\xi_i), \quad \text{with } x_{i-1} \leq \xi_i \leq x_i. \quad (4.6)$$

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

$$|R(h)| \leq \frac{b-a}{12}h^2 \max_{[a,b]} |f''(x)|. \quad (4.7)$$

☞ **Exercise 4.2: Error of the repeated trapezoidal rule**

Derive the expression for the remainder value of the repeated trapezoidal rule in (4.6). Next give proof for the upper bound on the remainder value, as given in (4.7).

HINT: For the proof, again use the *Intermediate Value Theorem* and the fact that f'' is continuous. ■

The error upper bound in (4.7) shows that the accuracy of the repeated trapezoidal is $O(h^2)$. Suppose we want to approximate an integral with a given accuracy. Automatic integration schemes use the estimate for the error to successively refine the discretisation (i.e. increase n , the number of points along an axis) until the requested accuracy is obtained. Let N denote the total number of discretisation points needed. In one dimension, $N = n \propto h^{-1}$. In two dimensions we need n points in the x -direction, and for each of those also n points in the y -direction, yielding a total of $N = n^2 \propto h^{-2}$ points. In general, in a d -dimensional domain, the total number of needed discretisation points is equal to $N \propto h^{-d}$. In other words: if we need to halve the size h for obtaining a certain accuracy, this requires 2^d as many discretisation points. Hence, the cost of the quadrature formula is exponential in d . 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*. Since the error for the repeated trapezoidal rule is $O(h^2)$, the error depends on the total number of points as follows :

$$\text{error} \propto \frac{1}{\sqrt[d/2]{N}} \quad (4.8)$$

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

When using a Monte Carlo simulation for approximating integrals instead, the error is of a statistical nature. It decreases with the square root of the sample size, *independently of the dimensionality of the integral*. This is expressed as:

$$\text{error} \propto \frac{1}{\sqrt{N}}, \quad (4.9)$$

which decreases (much) faster than (4.8) for $d > 4$. This is the main justification for studying Monte Carlo integration.

4.2 Alternative integration: Monte Carlo

4.2.1 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 $f : [0, 1] \rightarrow [0, 1]$,

$$E(f) = \int_0^1 f(x) dx.$$

If we define $S = [0, 1] \times [0, 1]$ and $A = \{(x, y) \in S | y \leq f(x)\}$, then we have that $E(f) = \int_0^1 f(x) dx$ equals the area of A . Therefore, if the random variables (X, Y) are uniformly distributed on S we have

$$P((X, Y) \in A) = \text{Area}(A) = E(f).$$

The hit-or-miss algorithm firstly generates a number of stochastic pairs $(X_1, Y_1), \dots, (X_n, Y_n)$, and sets $n_A = \#\{i | Y_i \leq f(X_i), 1 \leq i \leq n\} = \#\{i | (X_i, Y_i) \in A, 1 \leq i \leq n\}$. The law of large numbers states that

$$n_A/n \rightarrow P((X, Y) \in A) = E(f),$$

with probability one, when $n \rightarrow \infty$. In order to estimate the rate of convergence we observe that n_A is distributed according to the binomial distribution $\text{Bin}(n, E(f))$, see Example 3.3. Therefore we have for the mean squared error

$$E([n_A/n - E(f)]^2) = E([n_A/n - E(n_A/n)]^2) = \text{Var}(n_A/n) = E(f)(1 - E(f))/n.$$

The convergence is now established as

$$E(|n_A/n - E(f)|) \leq \sqrt{E([n_A/n - E(f)]^2)} = \sqrt{E(f)(1 - E(f))/n}. \quad (4.10)$$

4.2.2 The MC method using simple sampling

In this section we devise a method to compute the following integral

$$\int_a^b f(x) dx. \quad (4.11)$$

If we assume that x is a *random variable* which is uniformly distributed over $[a, b]$, then the *density function* of x is

$$p(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ 0, & \text{otherwise.} \end{cases} \quad (4.12)$$

and the *expectation* of x is

$$E(x) = \int_a^b xp(x) dx = \int_a^b \frac{x}{b-a} dx = \frac{b^2 - a^2}{2(b-a)} = \frac{b+a}{2}. \quad (4.13)$$

The expectation of a function f that depends on x is

$$E(f(x)) = \int_a^b f(x)p(x) dx = \int_a^b \frac{f(x)}{b-a} dx = \frac{1}{b-a} \int_a^b f(x) dx. \quad (4.14)$$

Which gives

$$\int_a^b f(x) dx = (b-a)E(f(x)). \quad (4.15)$$

So if we have an *estimate* for $E(f(x))$ we also have an estimate for the integral of f . The standard way to do it is to average f over a sample $\{X_i\}_{i=1}^N$ of our i.i.d. (independent and identically distributed) variable x

$$E(f(x)) \approx \overline{f_N(X)} = \frac{1}{N} \sum_{i=0}^N f(X_i). \quad (4.16)$$

We added a subscript N to indicate the number of samples taken to calculate the average.

4.2.3 Importance sampling

In the previous method we chose X_i uniformly 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 sampling from another pdf $g(x)$ chosen to be a reasonable approximation of the function we want to integrate:

$$\int_a^b f(x) dx = \int_a^b \frac{f(x)}{g(x)} g(x) dx. \quad (4.17)$$

See also equation (3.3) for the definition of the expectation. We draw points distributed according with $g(x)$ and compute the average of $f(x)/g(x)$ at these points. You may use techniques from section 3.5 to obtain the non-uniformly distributed variables. This technique is useful if g^{-1} is readily available, enabling us to use the inversion method. When we need to resort to the rejection method the computational cost becomes prohibitive and there is no advantage to using importance sampling.

The error of the MC method

In order to estimate the convergence of the method we may look at the mean squared error $E([\overline{f_N(X)} - E(f(x))]^2)$. From statistics we know that the sample mean is an *unbiased* estimator and the mean squared error is the same as the variance,

$$E([\overline{f_N(X)} - E(f(x))]^2) = \text{Var}(\overline{f_N(X)}) = \frac{\text{Var} \sum_{i=0}^N f(X_i)}{N^2} = \frac{\text{Var} f(X_0)}{N}.$$

see for example [15]. In order to obtain the mean absolute error we use the estimate

$$E(|X|^r) \leq E(|X|^r)$$

and obtain

$$E(|\overline{f_N(X)} - E(f(x))|) \leq \sqrt{E(\overline{f_N(X)} - E(f(x)))^2} = \sqrt{\text{Var}(\overline{f_N(X)})} = \sqrt{\frac{1}{N} \text{Var}(f(X_0))}. \quad (4.18)$$

The variance of $f(X_0)$ is unknown, but nonetheless we can conclude that the error decreases as \sqrt{N}^{-1} .

Note the following features of Monte Carlo approximation

- The error is probabilistic and does not depend on the dimensionality of the integral.
- The error does not depend on the smoothness of the function $f(x)$.

★ Exercise 4.3: Integrating a function with Monte Carlo

Calculate the integral

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

first using 'hit-or-miss' Monte Carlo. Secondly calculate the integral using 'sample mean' Monte Carlo, i.e. by calculating

$$E(\sqrt{1-X^2}) \approx \frac{1}{N} \sum_{i=1}^N \sqrt{1-X_i^2},$$

with $X, X_i \sim \tilde{U}(0, 1)$. 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 (4.10) and (4.18)). ■

4.2.4 Multi-dimensional Monte Carlo integration

In most instances, it is possible to generalise the Monte Carlo procedure to multiple dimensions. Hit-or-miss Monte Carlo is particularly simple. Suppose you need to integrate $f : \mathbb{R}^d \rightarrow \mathbb{R}$ over a domain $\Omega \subset [0, 1]^d$. You generate $d + 1$ random variables X_1, \dots, X_{d+1} in each step, treat the first d variables as coordinates of a point and test whether $f(X_1, \dots, X_d) < X_{d+1}$.

☞ **Exercise 4.4: Multi-dimensional hit-or-miss**

Formulate the multidimensional hit-or-miss procedure. Why does this approximate $\int_{\Omega} f(x) dx$? Can you derive the convergence speed in terms of the number of random variables needed? Does this depend on the dimension d ? ■

Generalising the regular Monte-Carlo procedure is somewhat more complicated. It is possible if the integration can be carried out by means of repeated integration (i.e. the d dimensional integral can be written as d nested integrals). The integration may then be written

$$\int_{\Omega} f(x_1, \dots, x_d) dx = \int_{a_d}^{b_d} \int_{a_{d-1}(x_d)}^{b_{d-1}(x_d)} \dots \int_{a_1(x_2, \dots, x_d)}^{b_1(x_2, \dots, x_d)} f(x_1, \dots, x_d) dx_1 \dots dx_d.$$

In principle, the Monte Carlo method may now be applied to each integration, starting at the inner integral and continuing up to the outer integration. In the first step we would consider f as a function of x_1 only and apply the Monte Carlo approximation,

$$\int_{\Omega} f(x_1, \dots, x_d) dx \approx \frac{1}{N} \sum_{i_1=1}^N \int_{a_d}^{b_d} \int_{a_{d-1}(x_d)}^{b_{d-1}(x_d)} \dots \int_{a_2(x_3, \dots, x_d)}^{b_2(x_3, \dots, x_d)} [(b_1(x_2, \dots, x_d) - a_1(x_2, \dots, x_d)) f(Y_1^{(i_1)}, x_2, \dots, x_d) dx_2 \dots dx_d,$$

where N is the number of random points taken and $Y_k^{(l)}$ is the l -th instance of the random number Y_k . Now, what's inside the integration is a function of x_2, \dots, x_d and we can again apply Monte Carlo considering this as a function of x_2 ,

$$\int_{\Omega} f(x_1, \dots, x_d) dx \approx \frac{1}{N^2} \sum_{i_1, i_2=1}^N \int_{a_d}^{b_d} \int_{a_{d-1}(x_d)}^{b_{d-1}(x_d)} \dots \int_{a_3(x_4, \dots, x_d)}^{b_3(x_4, \dots, x_d)} [(b_1(Y_2^{(i_2)}, x_2, \dots, x_d) - a_1(Y_2^{(i_2)}, x_3, \dots, x_d)) \cdot (b_2(x_3, \dots, x_d) - a_2(x_3, \dots, x_d)) \cdot f(Y_1^{(i_1)}, Y_2^{(i_2)}, x_3, \dots, x_d) dx_3 \dots dx_d.$$

Continuing in this fashion we end up with,

$$\int_{\Omega} f(x_1, \dots, x_d) dx \approx \frac{1}{N^d} \sum_{i_1, \dots, i_d=1}^N (a_d - b_d) \prod_{j=1}^{d-1} [b_j(Y_{j+1}^{(i_{j+1})}, \dots, Y_d^{(i_d)}) - a_j(Y_{j+1}^{(i_{j+1})}, \dots, Y_d^{(i_d)})] f(Y_1^{(i_1)}, \dots, Y_d^{(i_d)}). \quad (4.19)$$

There is now one final consideration, the distribution of the Y_i . They need to be distributed according to the boundaries in the integration, thus

$$\begin{aligned} Y_d &\propto U(a_d, b_d) \\ Y_{d-1} &\propto U(a_{d-1}(Y_d), b_{d-1}(Y_d)) \\ Y_{d-2} &\propto U(a_{d-2}(Y_d, Y_{d-1}), b_{d-2}(Y_d, Y_{d-1})) \\ &\vdots \\ Y_1 &\propto U(a_1(Y_2, \dots, Y_d), b_1(Y_2, \dots, Y_d)). \end{aligned}$$

It is easiest to generate X_1, \dots, X_d from $U(0, 1)$ and then set, in the given order

$$\begin{aligned} Y_d &= X_d(b_d - a_d) + a_d \\ Y_{d-1} &= X_{d-1}(b_{d-1}(Y_d) - a_{d-1}(Y_d)) + a_{d-1}(Y_d) \\ &\vdots \\ Y_1 &= X_1(b_1(Y_2, \dots, Y_d) - a_1(Y_2, \dots, Y_d)) + a_1(Y_2, \dots, Y_d). \end{aligned}$$

For some specific a_i and b_i you may be able to work out the above relations and insert them in formula (4.19).

The formula (4.19) looks a little troubling, the cost is N^d function evaluations and it seems that we have no gain over 'traditional' methods. However, looks deceive since we shall see in the next section that the error decreases as $N^{-d+1/2}$. It is therefore still in the order of one over the square root of the number of evaluations, independently of the dimension.

The error of multi-dimensional Monte Carlo

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 section 4.2.3 for the rationale behind this procedure. Let us work out the error expression,

$$e(N, d) = \sqrt{\text{Var}(f_N(Y_1, \dots, Y_d))} = \sqrt{\text{Var}\left(N^{-d} \sum_{i_1, \dots, i_d=1}^N f(Y_1^{(i_1)}, \dots, Y_d^{(i_d)})\right)},$$

where again $Y_k^{(l)}$ is the l -th realisation of the random variable Y_k . We now use independence of the random variables and the rules for working with variances to obtain

$$\begin{aligned} e(N, d) &= N^{-d} \sqrt{\text{Var} \sum_{i=1}^N f(Y_1^{(i)}, \dots, Y_d^{(i)})} \\ &= N^{-d} \sqrt{\sum_{i=1}^N \text{Var} f(Y_1^{(i)}, \dots, Y_d^{(i)})} \\ &= N^{-d} \sqrt{N \text{Var} f(Y_1^{(1)}, \dots, Y_d^{(1)})} \\ &= \mathcal{O}(N^{-d+1/2}). \end{aligned}$$

4.2.5 Discrepancy Sampling

Discrepancy sampling is a type of quasi-Monte Carlo integration. The points used are no longer random but predetermined point clouds. It turns out that coefficients of prime number expansions do very well. Let us firstly define how to write a number $i \in \mathbb{Z}$ in base p :

$$i = \sum_{i=0}^{\infty} a_i p^i.$$

The representation of i in the p -ary number system is then

$$i = a_0 a_1 a_2 \dots$$

You are already familiar with $p = 2$, which is the binary system. The treatment of real numbers is analogous to the way we write for example $0.321 = 3 * 10^{-1} + 2 * 10^{-2} + 3 * 10^{-3}$. For a real number between zero and one we define

$$x = \sum_{i=1}^{\infty} a_i p^{-i},$$

in the p -ary system. The representation of the number is preceded by a dot (and sometimes a zero):

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

If we combine the two systems we can write any real number,

$$x = \sum_{-\infty}^{\infty} a_i p^i,$$

with representation

$$x = \dots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \dots$$

Of course we do not write digits a_j if $a_k = 0$ for all $k \geq j > 0$ or all $k < j < 0$.

Say we want to compute an d -dimensional integral. We choose d prime numbers P_1, \dots, P_d . The first few prime numbers 2, 3, 5, 7, ... will work nicely. Now expand the number j in those prime bases,

$$\begin{aligned} j &\approx a_{01} + a_{11}P_1 + a_{21}P_1^2 + a_{31}P_1^3 \dots a_{m1}P_1^m, \\ j &\approx a_{02} + a_{12}P_2 + a_{22}P_2^2 + a_{32}P_2^3 \dots a_{m2}P_2^m, \\ j &\approx a_{03} + a_{13}P_3 + a_{23}P_3^2 + a_{33}P_3^3 \dots a_{m3}P_3^m, \\ j &\approx \vdots \\ j &\approx a_{0d} + a_{1d}P_d + a_{2d}P_d^2 + a_{3d}P_d^3 \dots a_{md}P_d^m, \end{aligned}$$

Now our j -th point $x_j = (x_{1j}, \dots, x_{dj})$ in d -space is defined using the expansions above. The i -th coordinate of the j -th point is created using the coefficients $a_{0i} \dots a_{mi}$ as follows

$$x_{ij} = 0.a_{0i}a_{1i}a_{2i} \dots a_{mi} = a_{0i}P_i^{-1} + a_{1i}P_i^{-2} + \dots + a_{mi}P_i^{-m-1}.$$

The number m is the precision in which you need your coordinates.

★ **Exercise 4.5: The volume of the d -dimensional unit sphere**

In this exercise you will compare Monte Carlo with discrepancy sampling. Our test problem is finding the volume of the d -dimensional unit sphere. The integral can be written

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

Explain why the applying Monte Carlo to this integral in fact amounts to the hit-or-miss algorithm. Alternatively, can you write down the integration using repeated integration, and apply regular Monte-Carlo. 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 `quasimc.cc` from the course web page [17] for the 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 (3.13). You can implement it using the code from [13], or using MATLAB's built-in gamma function.

Make graphs of the error in the Monte Carlo method and the discrepancy method as a function of the number of points N , for several values of the dimensionality d . It should hold that the error's behave as $O(N^{-1/2})$ respectively $O(N^{-1} \log^d(N))$ ($\approx O(N^{-1})$ for small d). ■

Chapter 5

Genetic Algorithms

5.1 Introduction

This chapter gives an introduction to genetic algorithms (GA's). Further introductions to the subject that the reader could consult are [2] and [5], but there is also a great number of resources available on the internet.

Genetic algorithms aim to solve *optimisation problems*, i.e. they try to either minimise or maximise some objective, subject to several parameters. We will assume that the objective may be cast in the form of a function ¹ that takes a vector $x \in \mathbb{R}$ (containing the parameters) and produces a number:

$$\min_{x \in D} f(x). \quad (5.1)$$

Here $D \subset \mathbb{R}^m$ is the domain of the function, i.e. the allowed range of the parameters. An important class of minimisation problems is the minimisation of the norm of a matrix vector product, $\min_{x \in \mathbb{R}} \|Ax\|_2^2$, many practical problems from e.g. physics are stated in this form. The graph of f is known as the *fitness landscape*, inspired by two dimensional graphs where you have hills and valleys. Finding the global minimum means finding the lowest valley, without getting stuck in another valley.

Genetic algorithms mimic evolution as it happens in nature, a population of candidates is evolved through several generations. In each generation members of the population are combined and mutated. One hopes that good solutions combine with other good solutions, creating better solutions and thereby evolving towards the optimal state. Let us discuss the algorithm in some more detail, firstly we define the i -th *generation*

$$\mathcal{G}^i = (x_1^i, \dots, x_n^i).$$

The column matrices $x_j \in D \subset \mathbb{R}^m$ are the *members*, *chromosomes* or *genes* of a generation, where m is the number of parameters. The \mathcal{G}^i may be interpreted as a matrix, the number of columns of \mathcal{G}^i (the size of the population) is denoted by $|\mathcal{G}^i|$. The first generation needs to be initialised, usually the (x_1^1, \dots, x_n^1) 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}^i . The two main operations are *selection* and *mutation*, but before describing them we need a way of evaluating how well a member minimises the problem. In genetic algorithm terminology: we want to determine the *fitness*.

¹This function need not be a continuous or differentiable. The function might be piecewise defined, or ever take function values from a table. This generality makes GA's suitable for a wide range of problems. Alternative methods (steepest descent, conjugate gradients, Newton iteration) require some smoothness.

5.1.1 Determining the fitness

Of course there are countless measures of fitness, for example if one knows the true solution t to (5.1) then one may evaluate $\|t - f(x_j^i)\|$ in some norm. However, we do not know the true solution. We do know that the x_j^i for which $f(x_j^i)$ is smallest (in generation i) is the best solution. We also have a second-best solution, etc. Now, we can assign values to x_j^i according to its place in the ordering. Let $p \in \mathbb{R}^m$ be such that

$$f(x_{p_k}^i) \leq f(x_{p_{k+1}}^i),$$

for $k = 1, \dots, m - 1$. Then p_j is a measure of the *fitness* for x_j^i :

$$\mathcal{F}(x_j^i) = m - p_j + 1.$$

In this approach it does not matter much how much better a solution is with respect to another solution. The first one will always have fitness one, the second one fitness two. The next section will show that it is beneficial to have a fitness that is proportional to the value of f , so that it is also measured how much better one solution is than another solution. The fitness values will be considered values of a discrete probability density function, therefore we want the fitness values to be positive and sum to one. Let $f_-^i = \min\{f(x_1^i), \dots, f(x_m^i)\}$ and $f_+^i = \max\{f(x_1^i), \dots, f(x_m^i)\}$ then we define the fitness evaluation as

$$\mathcal{F}(x_j^i) = \left[a + (b - a) \frac{f(x_j^i) - f_+^i}{f_-^i - f_+^i} \right] / \left[\sum_{j=1}^m \left(a + (b - a) \frac{f(x_j^i) - f_+^i}{f_-^i - f_+^i} \right) \right]. \quad (5.2)$$

In this way the objective scores are first mapped to $[a, b]$, after which it is made sure that all fitness scores sum up to one. Also note that a fitness score of zero will not occur. Think of the influence the values of a and b have on the final fitness of a chromosome.

5.1.2 Selection and reproduction

Selection is the process of determining which individuals are fit for reproduction and which individuals will be deleted from the current generation. The most common form of selection is *fitness proportional selection* where the probability of being selected for the next generation is given by (5.2). 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 one and then start adding the numbers $\mathcal{F}(x_0^i), \mathcal{F}(x_1^i), \dots$ until we exceed S . The analogy with the roulette wheel is obvious if one imagines the fitness of a gene to occupy a section of a roulette wheel. The probability that the ball falls in the section of roulette wheel occupied by a certain gene is of course proportional to the size of the section (the fitness). The variable R marks the spot where the ball rests on the wheel. Symbolically we represent the selection as

$$y^i = \mathcal{S}_S(\mathcal{G}^i),$$

the operator \mathcal{S}_S selects one member of \mathcal{G}^i .

How many individuals to select depends on our reproduction procedure. The most common practice is that selected individuals all have probability R of being selected for reproduction. Lets introduce the operator \mathcal{R}_R that determines if a member is selected for reproduction, this operator either returns a member or nothing.

Whenever we have two members selected for reproduction, we produce two offspring (see section 5.1.3 for various strategies). Symbolically this may be represented by

$$(y_1, y_2) = \mathcal{C}(x_1, x_2),$$

where the x are the parents and the y the offspring.

If we adopt *steady state* reproduction, where each generation has the same number of members, then we arrive at Algorithm 1.

Algorithm 1 The main GA loop with selection and steady state reproduction.

```

Initialise  $\mathcal{G}^0$ 
for  $i = 0$  to  $n - 1$  do
  while  $|\mathcal{G}^{i+1}| < m$  do
     $parent \leftarrow 0$ 
    while  $parent < 2$  and  $|\mathcal{G}^{i+1}| < m$  do
       $S \leftarrow \text{Random}, y \leftarrow \mathcal{S}_S(\mathcal{G}^i)$ 
       $R \leftarrow \text{Random}, z = \mathcal{R}_R(y_1)$ 
      if not empty  $z_1$  then
         $parent \leftarrow parent + 1$ 
         $x_{parent} \leftarrow z$ 
      end if
      Add  $y$  to  $\mathcal{G}^{i+1}$ 
    end while
    if  $|\mathcal{G}^{i+1}| < m - 1$  then
       $(x_1, x_2) \leftarrow \mathcal{C}(x_1, x_2)$ 
      Add  $x_1$  and  $x_2$  to  $\mathcal{G}^{i+1}$ 
    end if
  end while
  Perform analysis on  $\mathcal{G}^{i+1}$ 
  Write data to file
end for

```

5.1.3 Crossover

Crossover is the mechanism that does the actual optimisation. Selected members combine to make new members for the next generation. The hope is that the result of mixing two good chromosomes creates a new good chromosome. One often used type of crossover is *one point crossover*. We select two parent strings. At a certain randomly chosen position C in the strings we swap all elements of the chromosome that lie to the right of it. One can also try other crossover strategies, with for example multiple crossover points. You should adjust your crossover strategy to reflect the properties of your specific problem at hand.

As an example of one point crossover, take the following chromosomes (represented as bitstrings):

```

00110011
  x
11101111

```

We perform crossover at the mark. This gives the 'offspring'

```

00111111
11100011

```

5.1.4 Mutation

In nature, mutation is a spontaneous slight change in a chromosome. The reason for mutation is 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 fixed (small) value M . Next, each element within each chromosome is visited (for example the bits in the bitstring representing the chromosome) and a random number R_M is drawn. If $R_M < M$ the corresponding element of chromosome is slightly changed, for example by a bitflip if bitstrings are used. Take care when choosing the parameters S and M ! They should be chosen in such a way that solutions are not destroyed prematurely (by having M too high) and diversity in the population should be maintained.

5.1.5 Stopping criteria

One needs to think on the issue of when to stop the genetic algorithm. Since we do not know the true solution (in practical circumstances) we need a way to detect convergence. One good measure of convergence is to monitor 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 5.1: The genetic algorithm

Modify Algorithm 1 to include mutation and a stopping criterium based on convergence. ■

5.2 Implementation

This section several implementation issues are discussed. Up to this point it has not been made specific what a chromosome is, and what it represents. Here we will discuss how to represent chromosomes as bitstrings, and have the bitstrings represent real numbers. When using bitstrings to encode chromosomes, mutation becomes a simple bitflip.

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!

Use the code that is available from the course webpage, read the part about *classes* in appendix B if you haven't done so.

If the optimisation problem is expressed in the form of a function, then it is common practice to represent a chromosome by a bitstring², this bitstring then stands for a number. This is by no means mandatory, one could also let a chromosome be represented by a string of integers. However, by using the theory of *schemata* (see [6]) it is possible to show that a bitstring is most efficient in searching the fitness landscape.

However, a chromosome does not necessarily represent a number. Perhaps you have a weighted graph and aim to find the shortest path between two given points. In that case the chromosomes would represent different paths, not numbers. Selection and mutation operators should be redefined to be suitable for the particular problem at hand.

★ Exercise 5.2: 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 bitstring 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:

$$a = \sum_{i=0}^l 2^i b_i,$$

²Internally, this bitstring is implemented by a data structure such as a *linked list* or *array*.

where b_i is bit i in the chromosome and $l - 1$ is the total number of bits. ■

★ **Exercise 5.3: Representing real numbers**

In order to represent real numbers we firstly suppose we have an integer z represented by the chromosome. We can then transform to the real number by

$$r = mz + c,$$

where m and c are yet to be determined parameters. Note that $z_{min} = 0$ and $z_{max} = 2^l - 1$, where l is the number of bits in the chromosome. Now

$$\begin{aligned} r_{min} &= mz_{min} + c \\ r_{max} &= mz_{max} + c. \end{aligned}$$

These equations are easily solved to yield

$$r = \frac{r_{max} - r_{min}}{2^l - 1}z + r_{min}.$$

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

★ **Exercise 5.4: 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 bitflips, invert the bits according to the probability `double prob`. (Hint: look at `getBitstringText()` for an example of how to work with *iterators*) ■

★ **Exercise 5.5: 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 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 5.6: 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 criterium such that the genetic algorithm loop is terminated when there is convergence (within a specified tolerance). ■

5.3 Improvements

5.3.1 Elitism

The genetic algorithm mutates every element in the chromosome with a certain small probability. This implies that also the best 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 few best individuals are always selected and never mutated.

☞ **Exercise 5.7: Implementation of elitism**

Modify Algorithm 1 to include elitism. ■

★ **Exercise 5.8: 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. ■

5.3.2 Gray Codes

The way in which mutation is implemented has one important drawback. Numbers can be close together in 'representation space', while they are very different in 'number space'. For example, a mutation that turns 110000 into 010000 dramatically modifies the number that is encoded. Simple bit-flip mutations can yield big differences in the outcome.

It is favorable if small changes in the representation imply a small change in the represented number. One way to partially solve this are *Gray codes*. Such a code maps bitstrings to numbers and has the property that incrementing or decrementing a number will change only one bit of the representation. Gray codes for a given bitlength are not unique, there are many different valid Gray codes. Note that incrementing or decrementing the number implies one bitshift, however *the opposite is not true*. The probability that a bitshift leads to a smaller increase in the represented number has however become greater.

Let's introduce the concatenation operator \oplus that concatenates two sequences (e.g. $01 \oplus 10 = 0110$). Using this operator the recipe for generating all Gray Codes of a certain bit length is simple, See algorithm 2.

Algorithm 2 Generating Gray codes of bitlength n .

```

 $S_0 = \{ \}$ 
 $i \leftarrow 1$ 
for  $i = 1$  to  $n$  do
  Visit elements  $g_j$  for  $j = 0, \dots, i - 1$  in  $S_{i-1}$ , add  $0 \oplus g_j$  to  $S_i$ .
  Visit elements  $g_j$  for  $j = i - 1, \dots, 0$  in  $S_{i-1}$ , add  $1 \oplus g_j$  to  $S_i$ .
   $i \leftarrow i + 1$ 
end for

```

Fortunately we do not need to precalculate tables containing Gray codes, since a convenient algorithm exists (see [13]) for converting Gray codes $g_n \dots g_0$ to binary $b_n \dots b_0$. We present this efficient procedure in Algorithm 3, verify that this algorithm works using a few numbers generated with Algorithm 2.

Algorithm 3 Converting Gray codes g to binary codes b .

```

 $b_n \leftarrow g_n$ 
for  $i = n - 1$  to  $0$  do
  if  $b_{i+1} == g_i$  then
     $b_i \leftarrow 0$ 
  else
     $b_i \leftarrow 1$ 
  end if
   $i \leftarrow i - 1$ 
end for

```

☞ **Exercise 5.9: Gray Code generation**

Verify the following table. Use algorithm 2 to generate column three, then use 3 to check the Gray Code to binary conversion.

Decimal	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

■

★ **Exercise 5.10: 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, instead of converting the bitstring to decimal you should now convert the bitstring (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.

■

5.4 Projects

The following two subsections present two advanced optimisation problems. The travelling salesman is a classic problem: try to find the shortest route that visits all cities on a map. There are many local minima and no algorithm is known that does better than trying all possible routes. You will redefine chromosomes to represent a path between nodes. The second problem is the equidistribution of n repelling particles on a sphere. This problem is less well known than the travelling salesman, but certainly not less challenging. You will redefine chromosomes to represent sets of vectors of real numbers. Combining GA's with traditional methods may be needed to obtain satisfactory results.

★ **Exercise 5.11: Project**

Read the following sections and choose one of the problems as your main topic. These are hard optimisation problems, do your experiments in a systematic way and don't hesitate to consult additional literature.

■

5.4.1 The travelling Salesman

The so called *travelling salesman problem* (TSP) is a notoriously difficult optimisation problem. The salesman is visiting a number of n cities at locations (x_i, y_i) . He needs to visit every city exactly once, and of course he would like to take the shortest route possible.

Now enumerate the cities, and introduce a solution vector x containing a permutation (route) of

the city numbers. Now, the problem can be formulated as follows. Minimise

$$f(z) = d(z_1, z_n) + \sum_{i=2}^n d(z_i, z_{i-1}),$$

where the squared distances are given by

$$d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

This looks deceptively simple, yet despite many efforts that have been made, we cannot do much better than trying all possible combinations of cities. This gives us an algorithm running in $\mathcal{O}(n!)$ time. This is very slow, if 23 cities would take one hour, then adding only one city would cost one day.

We can implement the travelling salesman problem simply using our genetic algorithm. Create a number of random permutations of $\{1, \dots, n\}$ and evolve using the objective function $f(z)$. Yet, there is one catch. Mutation and crossover do not yield feasible tours! Double and missing cities will show up in the tour. We solve the problem by introducing new mutation and crossover procedures.

Mutation of tours

Mutation is simple, select two cities from the tour and swap them. Optionally you could evaluate the fitness before and after mutation, and keep the mutation only if the result is better.

Crossover

Crossover is more complicated. One nice way of doing this is *greedy crossover*, a procedure which continuously selects from the two parents (if possible) the closest city to the current city. Study Algorithm 4, which details the procedure for the case of producing a child C_1 starting from parent P_1 . Creating C_2 from P_2 is analogous. You need to think of the implementation of nextCity and Distance yourself.

Algorithm 4 Crossover for the TSP, creates child one C_1 using parent one P_1 .

```

curCity=1;
C1 ← empty
while C1 not full do
  N1 ← nextCity(P1, curCity)
  N2 ← nextCity(P2, curCity)
  d1 = Distance(P1(curcity), N1)
  d2 = Distance(P2(curcity), N2)
  if C1 contains N1 and N2 then
    Randomly select a non-selected city.
  else if C1 contains N1 then
    Add N2 to C1
  else if C1 contains N2 or d1 < d2 then
    Add N1 to C1
  else if d1 > d2 then
    Add N2 to C1
  else
    Add N1 to C1
  end if
end while

```

★ **Exercise 5.12: The travelling salesman**

Implement the travelling salesman. First try only mutation, when this works, proceed with crossover. You can find a wealth of test problems, with best known solutions at [14]. ■

5.4.2 Charges on a Sphere

This exercise is about finding the optimal distribution of repelling point charges on a spherical shell. Suppose we have an ideal conducting spherical shell 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 pose the problem in \mathbb{R}^m , where we consider the spherical shell

$$S^m = \{x \mid \|x\|_2 = 1, x \in \mathbb{R}^m\}.$$

Let $x_1, \dots, x_n \in S^m$ be points on the sphere. The distance between two points x_i and x_j on S^m is given by

$$d_{ij} = \|x_i - x_j\|_2.$$

We want to minimise the total Riesz s -energy,

$$E_s(n) = \sum_{i=1}^n \sum_{j>i} d_{ij}^{-s}, \quad (5.3)$$

which is for $s = 1$ 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. By fixing one particle at $(1, 0, \dots, 0)$ you have eliminated a lot of symmetry, and no loss of generality. The symmetries in the problem imply that there is no unique solution. 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*, for $s \rightarrow \infty$ it is the *Tammes problem*. The latter 'ultrarepulsive' optimisation problem is easily reformulated in a more compact way,

$$\text{find the points } x_1, \dots, x_n \text{ that maximise } \min_{i,j} d_{ij}. \quad (5.4)$$

In other words, 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 $n = 1, 2, 3, \dots$?

Implementation

From a genetic algorithm point of view you need an encoding of points on S^m . You have a few possible options to do this. You could create a string that encodes n vectors of length m . 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 $m - 1$ coordinates arbitrary and calculate the m -th as $\pm\sqrt{1 - \|(x_{i,1}, \dots, x_{i,m-1})\|^2}$, where i just denotes the i -th point on the 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 $\|x_i\| = 1$ always holds. An example of a genetic algorithm approach to the Thomson problem may be found in [8].

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 $m - 1$ angular coordinates ϕ_j . The coordinates of a point may then be represented as

$$\begin{aligned}
 x_1 &= r \cos(\phi_1) \\
 x_2 &= r \sin(\phi_1) \cos(\phi_2) \\
 x_3 &= r \sin(\phi_1) \sin(\phi_2) \cos(\phi_3) \\
 &\vdots \\
 x_{m-1} &= r \sin(\phi_1) \sin(\phi_2) \dots \sin(\phi_{m-2}) \cos(\phi_{m-1}) \\
 x_m &= r \sin(\phi_1) \sin(\phi_2) \dots \sin(\phi_{m-2}) \sin(\phi_{m-1}),
 \end{aligned} \tag{5.5}$$

where ϕ_1 is the angle between vector $\vec{x} \equiv [x_1, \dots, x_m]$ 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, etc.

Exercises

Choose one, or a few, of the following exercises.

Exercise 5.13: Energies for the Thomson problem

In [4] you will find tables of the energy $E_1(n)$ corresponding to n -particle configurations for the Thomson problem. Verify some entries using the genetic algorithm. ■

Exercise 5.14: 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 valid local minimum. In [4] a *steepest descent* procedure is described. The resulting algorithm is then as shown in Algorithm 5. Of course you may also use CG, Newton iteration, or any other optimisation technique. ■

Exercise 5.15: 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 $n = 1, \dots, 6$ 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 5 GA combined with steepest descent.

```

while no convergence do
  Evaluate
  Select
  Crossover
  Mutate
  Perform steepest descent iteration
end while

```

Appendix A

Writing Reports

The report should be written in \LaTeX , which has become the standard for scientific typesetting of documents. For Dutch speaking students I recommend Jaap van Oostrum's introduction to \LaTeX (get it from [18] or buy a hardcopy at the computer science department). Some worthwhile resources on the web include,

<http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/>
<http://www.cs.cornell.edu/Info/Misc/LaTeX-Tutorial/LaTeX-Home.html>
<http://it.metu.edu/latex/>
<http://heather.cs.ucdavis.edu/~matloff/latex.html>

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.

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.

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 mathematics knowledge. Write in 'article style', use formal but clear sentences. Do not refer to the exercises in the handouts directly. Thus, 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 $\text{\textcircled{E}}$ are intended to point you to interesting aspects.

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 C++ source code.

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:

```
(file mainprogram.h)
#define USESPECIALTRICK
```

Any other file that is also compiled can now make use of this macro:

```
(file mainprogram.c)
#include "mainprogram.h"

double dosomething()
{
    double d = 0.0;
    #ifdef USESPECIALTRICK
        d = somespecialtrick();
    #endif
    return d;
}
```

If, at compile time, the `USESPECIALTRICK` macro is set¹, the `#ifdef` condition evaluates to true, and the call to `somespecialtrick` will be active in the resulting program. The advantage of this

¹`#define` 'sets' a variable, even though it may not be with an explicit value.

approach, 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, and are *always the same* during any future execution of the program.

The only exception is that defined macros can also be undefined (at compile time only, that is), using `#undef NAME`.

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 `long` uses 4 bytes (i.e. $4 \cdot 8 = 32$ bit), so the number of different values that it can contain is $256^4 = 2^{32} = 4294967296^2$. 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 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:

²An `unsigned long` ranges from 0 to 4294967295, a `signed long` from -2147483648 to 2147483647. Notice how the value 0 also needs a place, hence the maximum value is always 1 less.

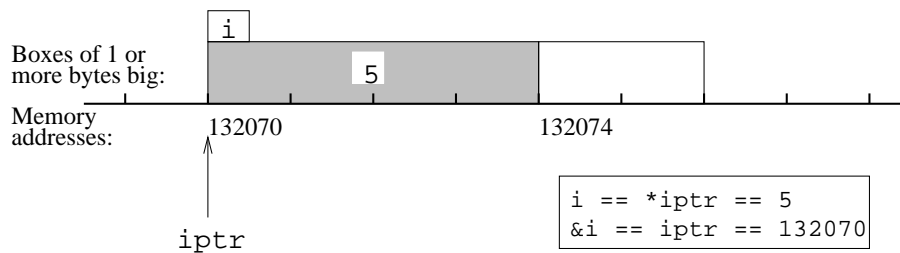


Figure B.1: Schematic representation of small piece of computer memory. A variable `int i` has an assigned value of 5, and pointer `int *iptr` points to `i`.

```
i = i + 1;
*iptr = *iptr + 1;
```

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 latest 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 equivalences³:

```
&a[i] ≡ a + i;
a[i] ≡ *(a + i)
```

Remember that `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 ’. The program knows however, that `a` contains values of type `int`, so any computations that are purely performed with pointer `a` or addresses `&a[i]` are automatically corrected to using the correct multiplication factor.

³Notice that an equivalence relation \equiv is shown, the first expression would not be a valid assignment in C++ code using `=`.

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 `ints` `x` and `y` as parameters and should swap their values⁴. 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...
}
```

⁴The example of swapping integers comes from [1].

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 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.

B.2.4 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];             // Not the preferred way
}
```

In plain C, the declaration of an array variable does not allow a variable to specify the array size. 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 practice 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;
```

B.2.5 What To Remember?

The preceding sections introduced pointer concepts including technical background. The main practical concepts that need to be remembered are summarised below:

<code>int *a</code>	declares a pointer <code>a</code> to an <code>int</code> .
Operator <code>*</code>	The contents of ...
Operator <code>&</code>	The address of ...
<code>a = new double[n]</code>	Allocates ('reserves') memory for an array of <code>n</code> doubles.
<code>delete n</code>	Cleans up all allocated memory for a scalar variable pointer <code>n</code> .
<code>delete[] a</code>	Cleans up all allocated memory for an array variable pointer <code>a</code> .
For a function <code>void foo(int *x)</code> , call it with either a pointer variable: <code>foo(a)</code> , or the address of a normal variable: <code>foo(&i)</code> .	

B.3 Object-oriented features in C++

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. Here is an example class definition:

```
#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;
    }

    void print()
    {
        cout << "Student " << name << " is " << age << " years old" << endl;
    }
};
```

```
    }
};
```

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 practice 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.

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 sample below serves as running example in this section. The paragraphs hereafter discuss each part separately.

```
#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;        // delete requires a pointer
}
```

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 can be used when accessing public member variables of an object.

⁵Class members can additionally be declared as `protected`, but we do not further discuss that here.

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.

Bibliography

- [1] Leen Ammeraal. *C++*. Academic Service, zesde druk edition, 2001.
- [2] David A. Coley. *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific, 1999.
- [3] Richard E. Crandall. *Projects in Scientific Computation*. Springer, 2000.
- [4] T Erber and G M Hockney. Equilibrium configurations of n equal charges on a sphere. *Journal of Physics A: Mathematical and General*, 24(23):L1369–L1377, 1991.
- [5] W. Banzhaf et al. *Genetic Programming, an Introduction*. Morgan Kaufmann Publishers, 1998.
- [6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Kluwer Academic Publishers, 1989.
- [7] The Mathworks. Matlab. <http://www.mathworks.com>.
- [8] 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(4):R1740–R1743, 1996.
- [9] C++ Resources Network. C++ tutorial. <http://www.cplusplus.com/doc/tutorial/>.
- [10] S. Park and K. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.
- [11] Alan C. Planz. *C Quick Reference*. Que Corporation, 1988.
- [12] Rob Pooley. C and c++ course. <http://www.macs.hw.ac.uk/~rjp/Coursewww/>.
- [13] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, second edition, 1992.
- [14] Gerhard Reinelt. Tsplib. <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.
- [15] John A. Rice. *Mathematical Statistics and Data analysis*. Duxbury Press, second edition, 1995.
- [16] Kermit Sigmon. Matlab primer. <http://math.ucsd.edu/~driver/21d-s99/matlab-primer.html>.
- [17] A.N. Swart. Course web page. <http://www.math.uu.nl/people/swart>.
- [18] Piet van Oostrum. Latex handleiding. <http://www.cs.uu.nl/people/piet/>.