Introduction to Scientific Computing
with
Maple Programming

Zhonggang Zeng
©2009

with contributions from David Rutschman

August 23, 2015
## Contents

### Table of Contents

1. **Fundamentals**
   - 1.1 Maple command lines ........................................... 2
   - 1.2 Simple programming ............................................. 10
   - 1.3 Conditional statements - Branching ............................ 18
   - 1.4 Common errors .................................................... 23
   - 1.5 Writing a project report on a Maple worksheet ............... 24
   - 1.6 Exercises .......................................................... 24
   - 1.7 Projects ............................................................ 30

2. **Loops. Part I**
   - 2.1 The “for-do” loop ............................................... 33
   - 2.2 Iteration ............................................................ 35
     - 2.2.1 Newton’s iteration (Numerical Analysis) ................. 37
     - 2.2.2 Chaotic map (Dynamic System) ............................. 38
     - 2.2.3 Continuous fractions (Number Theory) .................. 39
     - 2.2.4 Continuous fractions, revisited ........................... 40
   - 2.3 Recursive summation .............................................. 41
     - 2.3.1 Taylor series (Approximation Theory) ................... 42
     - 2.3.2 Quadrature (Numerical Analysis) .......................... 44
     - 2.3.3 Nested sums and products (Elementary Algebra) ....... 45
   - 2.4 Exploring scientific computing ................................. 47
     - 2.4.1 Fourier series (Fourier Analysis) ........................ 47
     - 2.4.2 Solving congruences (Number Theory) ................... 48
   - 2.5 Exercise ............................................................ 49
   - 2.6 Projects ............................................................ 58

3. **Loops. Part II**
   - 3.1 The “while-do” loop ............................................. 61
   - 3.2 The golden section method (Optimization) .................... 63
     - 3.2.1 The unimodal function ..................................... 63
### CONTENTS

3.2.2 The golden ratio .................................. 64  
3.2.3 Method of the golden section ...................... 65  
3.3 Vectors .............................................. 67  
3.3.1 Generating and accessing a vector .................. 67  
3.3.2 Vectors as input .................................. 68  
3.3.3 Vectors as output ................................ 69  
3.3.4 Sorting (Computer Science) ....................... 71  
3.4 Exploring scientific computing ....................... 72  
3.4.1 The bisection method (Numerical Analysis) .... 72  
3.4.2 The greatest common divisor (Number Theory) ... 73  
3.4.3 The greatest common divisor (Computer Algebra) 74  
3.4.4 Rational approximation (Number Theory) ........ 75  
3.4.5 Rational approximation revisited (Number Theory) 76  
3.4.6 The sieve of Eratosthenes (Number Theory) ....... 77  
3.5 Exercise ............................................. 79  

4 Carrying on ............................................ 87  
4.1 Data types ........................................... 87  
4.2 Subroutines .......................................... 88  
4.2.1 Sorting revisited (Computer Science) ............ 88  
4.2.2 Using vector dot product and norm subroutines .... 89  
4.2.3 The 3n + 1 iteration ................................ 90  
4.2.4 Quadrature revisited (Numerical Analysis) ....... 91  
4.2.5 Hypotrochoid in action ......................... 93  
4.2.6 Venn Diagram (Discrete Mathematics) .......... 93  
4.2.7 Mathematical painting ............................ 96  
4.2.8 Prisoner’s Dilemma (Game Theory) ................ 96  
4.3 Exploring scientific computing ...................... 99  
4.3.1 Julia sets (Dynamic Systems) ................... 99  
4.3.2 Factoring an integer (Number Theory) ............ 100  
4.3.3 Median and quartiles (Statistics) ................ 102  
4.3.4 Chinese Remainder Theorem (Number Theory) .... 103  
4.4 Exercise ............................................. 105  
4.5 Projects ............................................. 108  
4.5.1 The Game of Chicken (Game Theory) ............. 108  

5 Solving equations ................................ 111  
5.1 Solving equations for conventional solutions ....... 111  
5.1.1 Maple equation solvers ............................ 111  
5.1.2 Matrices and the Linear Algebra package ......... 113  
5.1.3 Percent mixture problem (Elementary Algebra, Chemistry) 115  
5.1.4 Creating quadrature rules (Numerical Analysis) ... 116  
5.1.5 Interpolations (Numerical Analysis) ............. 118  
5.1.6 Cubic splines (Numerical Analysis) ................ 119  

---

3.2.2 The golden ratio .................................. 64  
3.2.3 Method of the golden section ...................... 65  
3.3 Vectors .............................................. 67  
3.3.1 Generating and accessing a vector .................. 67  
3.3.2 Vectors as input .................................. 68  
3.3.3 Vectors as output ................................ 69  
3.3.4 Sorting (Computer Science) ....................... 71  
3.4 Exploring scientific computing ....................... 72  
3.4.1 The bisection method (Numerical Analysis) .... 72  
3.4.2 The greatest common divisor (Number Theory) ... 73  
3.4.3 The greatest common divisor (Computer Algebra) 74  
3.4.4 Rational approximation (Number Theory) ........ 75  
3.4.5 Rational approximation revisited (Number Theory) 76  
3.4.6 The sieve of Eratosthenes (Number Theory) ....... 77  
3.5 Exercise ............................................. 79  

4 Carrying on ............................................ 87  
4.1 Data types ........................................... 87  
4.2 Subroutines .......................................... 88  
4.2.1 Sorting revisited (Computer Science) ............ 88  
4.2.2 Using vector dot product and norm subroutines .... 89  
4.2.3 The 3n + 1 iteration ................................ 90  
4.2.4 Quadrature revisited (Numerical Analysis) ....... 91  
4.2.5 Hypotrochoid in action ......................... 93  
4.2.6 Venn Diagram (Discrete Mathematics) .......... 93  
4.2.7 Mathematical painting ............................ 96  
4.2.8 Prisoner’s Dilemma (Game Theory) ................ 96  
4.3 Exploring scientific computing ...................... 99  
4.3.1 Julia sets (Dynamic Systems) ................... 99  
4.3.2 Factoring an integer (Number Theory) ............ 100  
4.3.3 Median and quartiles (Statistics) ................ 102  
4.3.4 Chinese Remainder Theorem (Number Theory) .... 103  
4.4 Exercise ............................................. 105  
4.5 Projects ............................................. 108  
4.5.1 The Game of Chicken (Game Theory) ............. 108  

5 Solving equations ................................ 111  
5.1 Solving equations for conventional solutions ....... 111  
5.1.1 Maple equation solvers ............................ 111  
5.1.2 Matrices and the Linear Algebra package ......... 113  
5.1.3 Percent mixture problem (Elementary Algebra, Chemistry) 115  
5.1.4 Creating quadrature rules (Numerical Analysis) ... 116  
5.1.5 Interpolations (Numerical Analysis) ............. 118  
5.1.6 Cubic splines (Numerical Analysis) ................ 119
5.2 Exploring scientific computing

5.2.1 Sums of powers (Elementary Algebra, Discrete Mathematics) .......................... 123

5.2.2 Viète’s root-finding method (Numerical Analysis, Algebra) ............................ 125

5.2.3 From Viète to Durand-Kerner (Numerical Analysis) ......................................... 126

5.2.4 Cubic splines, revisited (Numerical Analysis) .................................................. 127

5.3 Exercises ................................................................................................................ 129
Scientific Computing Index

Numerical Analysis

- Zero-finding iterations
  - Babylonian method or Heron’s iteration 35
  - Newton’s iteration 37
  - Halley’s iteration 51
  - Laguerre’s iteration 51
  - Fixed point iteration 61
  - Bisection method 72
  - Viète’s root-finding method 125, 130
  - Durand-Kerner root-finding method 126, 130

- Quadrature, Numerical integration 44, 91
  - Trapezoidal rule 44
  - Simpson’s rule 57
  - Creating integration rules 116

- Interpolation
  - Lagrange interpolation polynomial 118
  - General interpolation 129
  - Cubic spline 119, 127, 130

Number Theory

  - Continuous fraction 39, 40, 52, 53, 54, 54, 54, 76
  - Ascending continuous fraction 59
  - Solving congruence 48, 54, 55
  - The Euclidean Algorithm for GCD 73
  - The Euclidean Algorithm for rational approximation 76
  - Rational approximation 75, 76, 81
  - Catalan numbers 49
  - Padovan sequence 49
  - Conway sequence 49
  - Fibonacci number 36
  - Fibonacci multiple 80
  - Bernoulli number 58
  - Pythagorean triple 79
  - Twin primes 79, 80
  - The $3n+1$ iteration 80, 90
  - The sieve of Eratosthenes 77, 85, 85
  - Ulam’s luck number 85
  - The Josephus problem 85
  - Integer factoring 100, 107
  - Chinese Remainder Theorem 103, 108
  - Euler Phi function 107
GCD and LCM by factoring 107

Approximation Theory

Taylor series 42, 55, 56, 56, 56
Fourier series 47, 56
continuous fraction 56, 56

Dynamic Systems

Henon map 38
Ikeda map 51
Tinkerbell map 52
Kaplan-Yorke map 52
Julia set 99, 107

Finance

Mortgage 26
Annuity 14
Tax schedule 29
Tuition calculation 30
cost calculation 27
longevity of a retirement fund 81

Elementary Algebra

Quadratic equation 16, 28, 28, 28, 28, 31
Cubic equation 24, 32
linear inequality 30
nested sum 45, 60, 57, 57
Percent mixture problem 115, 129
Sum of Powers 123, 129

Linear Algebra

Angle between vectors 82, 89
Vector projection 82, 105
Using dot product and norm 89, 105

Calculus

Harmonic series 80
Taylor series 42, 55, 56
infinite series 57, 58, 56, 56
Madhava-Leibniz series 55
Madhava series of $\pi$ 55
Wallis’ infinite product of $\pi$ 55
Bernoulli’s identity 58
CONTENTS

Arclength 112

Geometry
   Area of a polygon 83

Statistics
   Standard deviation 83, 106
   nearest component 83
   Pearson’s sample correlation coefficient 84, 84, 106, 106
   median and quartiles 102, 108

Discrete Mathematics
   Venn diagram 93
   Sum of Powers 123, 129

Game Theory
   Prisoner’s dilemma 96, 107
   The Game of Chicken 108

Computer Science
   sorting 71
   sorting by index 84
   sorting by insertion 84
   Sorting with a subroutine 88
   the Euclidean Algorithm for polynomial GCD 74, 81
   Sum of Powers 123, 129

Physics
   Physics formulas 24, 26, 26, 26, 26

Chemistry
   Percent mixture problem 115
Introduction for Math 340 Students

Welcome to Math 340!

This text is a work-in-progress, based on a decade of class materials prepared by Prof. Zhonggang Zeng with added materials by Prof. David Rutschman.

Our hope is that you will use this book to build on what is done in class, so that you can become a confident user of Maple. Maple is a very useful tool for mathematicians, and this is your opportunity to make it yours!

We have included many examples and exercises - more than what will be assigned or covered in class. We hope that you will use them as practice problems. Take the time to try some of the examples, make sure they run, and then play around with changes.

We also welcome your input on this text. Please feel free to contact us (in person or by email: D-Rutschman@neiu.edu or Z-Zeng@neiu.edu).

Enjoy!

Preface

This is not a book about the art of programming. It is about how to use programming as a tool when you do mathematics.
Chapter 1

Fundamentals

Maple and several other computer algebra systems (CAS) have been fundamentally changing the way we study, teach and apply mathematics. Each of those systems combines a comprehensive collection of mathematical functionalities, a graphical and computational environment, a programming language, a word processor, and many other features in one software package, making it possible to perform sophisticated mathematical computations with a single or a sequence of commands. Currently, Maple and Mathematica are arguably the two most successful computer algebra systems, along with Matlab that is mainly built for efficient numerical computations.

Maple is a product of MapleSoft, one of the leading commercial providers of software tools for mathematical, engineering and scientific computing. It includes feature-rich and user-friendly interfaces along with a huge collection of mathematical functionalities for manipulating numbers and algebraic expressions. Furthermore, Maple comes with a simple programming language that is capable of being used to implement sophisticated algorithms for scientific computing. It is an excellent platform for introducing scientific computing, basic programming, mathematical experimentation and simulation.
Maple starts in its “document mode” as shown in Figure 1.1, where word-processing and mathematical computation can be mixed in each line of the document. Throughout this book, we prefer to focus on scientific computing by separating command input from word-processing. For this purpose, a command prompt can be initiated by clicking the button “[>]” in the Maple toolbar (See Figure 1.1). Multiple prompts can be initiated by clicking the “[>]” button repeatedly. Alternatively, Maple can be started in “Worksheet mode”. This can be done by clicking “File → New → Worksheet mode”, or by changing the default starting interface to worksheet mode following “Tools → Options → Interface”.

Furthermore, we prefer to input commands in Maple notation style, which can be set up by clicking the following sequence of tabs: “Tools → Options → Display → Input display → Maple Notation → Apply Globally” (see Figure 1.1).

1.1 Maple command lines

After initiating command prompts in Maple, the user enters instructions at each prompt “[>]” from keyboard. Figure 1.2 shows some simple examples of command line computations.

![Figure 1.2](image)

End a command by a semicolon “;” (or a colon if for some reason output is to be hidden), and Maple executes the line typed when the Enter key is pressed following the semicolon.
Words typed following a “#” sign in each line form a *comment* that can be used to give a brief explanation or to leave a short note. A comment has no effect in the outcome of the computation.

Maple can perform many mathematical operations with one-line commands. For example, some basic computations like \( 3 + 5 \times 8 \), \( \frac{3}{5} \), \( 3^5 \), \( \sqrt{5} \), \( e^5 \), \( \ln 3 \) are shown in Figure 1.2.

Notice the difference in output between \( \sqrt{5} \) and \( \sqrt{5.0} \). When input numbers are integers, Maple performs the *symbolic computation* without calculating its numerical value. If a decimal number is involved, Maple carries out the *numerical computation* with a default 10-digit precision unless otherwise specified.

The `evalf` instruction forces a finite precision (floating point number) evaluation:

```
> evalf(sqrt(5));
2.236067978
> (1/2)^4; 0.5^4; evalf((1/2)^4);
\frac{1}{16}
.0625
.06250000000
```

Notice the differences among the last set of instructions.

One of the most important operations is to assign a value to a variable name using the assignment “:=” (colon + equal sign), say \( x := 2 \), which assigns a value 2 to the variable name \( x \). After that, the name \( x \) carries the number 2 and operations like \( x + 3 \) produces the same result as \( 2 + 3 \).

```
> x:=2; y:=5;
x := 2
y := 5
> (x+3*y)/(2*x-7*y);
\frac{-17}{31}
> evalf(%);
-.5483870968
```

The `%` symbol represents the result of the most recent execution. In this example, it represents \( \frac{-17}{31} \).

The assignment sign (“:=”) should not be confused with the equal sign, given statements such as:

```
> a := 2.5;
a := 2.5
> a := a*3.0 + a;
a := 10.00
```
Think of this as “updating” the variable. What is on the left is the “new” value, computed from the “old values on the right.

Maple accepts Greek letters:

\[ \alpha, \beta, \gamma, A, B, \Gamma \]

However, the name Pi carries the value of the special number

\[ \pi = 3.1415926535897932384626 \ldots \]

while pi does not.

\[ \text{evalf(Pi)}; \]

\[ 3.141592654 \]

\[ \text{alpha:=2*Pi/3;} \]

\[ \alpha := \frac{2\pi}{3} \]

\[ \text{sin(alpha), cos(alpha), tan(alpha), cot(alpha);} \]

\[ \frac{1}{2}\sqrt{3}, -\frac{1}{2}, -\sqrt{3}, -\frac{1}{3}\sqrt{3} \]

\[ \text{evalf(\%);} \]

\[ .8660254040, -.5000000000, -1.732050808, -.5773502693 \]

\[ \text{ln(alpha), exp(alpha);} \]

\[ \ln\left(\frac{2\pi}{3}\right), e^{(2/3 \pi)} \]

\[ \text{evalf(ln(alpha)), evalf(exp(alpha));} \]

\[ .7392647780, 8.120527402 \]

Table 1.1 is a list of frequently used mathematical operations and their corresponding Maple symbols and syntax.

If an arithmetic expression consists of several operations and function evaluations, Maple follows the usual precedence of operations:
### Table 1.1: Common operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Math notation</th>
<th>Example Maple Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>$3 + 5$</td>
<td>$3+5$</td>
<td>8</td>
</tr>
<tr>
<td>subtraction</td>
<td>$3 - 5$</td>
<td>$3-5$</td>
<td>$-2$</td>
</tr>
<tr>
<td>multiplication</td>
<td>$3 \times 5$</td>
<td>$3*5$</td>
<td>15</td>
</tr>
<tr>
<td>division</td>
<td>$3 \div 5$</td>
<td>$3/5$</td>
<td>$\frac{3}{5}$</td>
</tr>
<tr>
<td>exponentiation</td>
<td>$3^5$</td>
<td>$3^5$</td>
<td>243</td>
</tr>
<tr>
<td>exponential function</td>
<td>$e^{3.2}$</td>
<td>$\exp(3.2)$</td>
<td>24.53253020</td>
</tr>
<tr>
<td>absolute value</td>
<td>$</td>
<td>-3.2</td>
<td>$</td>
</tr>
<tr>
<td>square root</td>
<td>$\sqrt{5}$</td>
<td>$\text{sqrt}(5)$</td>
<td>$\sqrt{5}$</td>
</tr>
<tr>
<td>sine</td>
<td>$\sin \frac{3\pi}{4}$</td>
<td>$\sin(3\pi/4)$</td>
<td>$\sin \left( \frac{3\pi}{4} \right)$</td>
</tr>
<tr>
<td>cosine</td>
<td>$\cos$</td>
<td>$\cos(\pi/6)$</td>
<td>$\cos(\pi/6)$</td>
</tr>
<tr>
<td>tangent</td>
<td>$\tan$</td>
<td>$\tan(2\pi/3)$</td>
<td>$\tan(\frac{2\pi}{3})$</td>
</tr>
<tr>
<td>cotangent</td>
<td>$\cot$</td>
<td>$\cot(x)$</td>
<td>$\cot(x)$</td>
</tr>
<tr>
<td>natural logarithm</td>
<td>$\ln$</td>
<td>$\ln(2.3)$</td>
<td>.8129091229</td>
</tr>
<tr>
<td>factorial</td>
<td>$!$</td>
<td>$5!$</td>
<td>120</td>
</tr>
</tbody>
</table>
1. Exponentiation or function evaluation (highest)
2. Multiplication or division
3. Arithmetic negation (sign change)
4. Addition and/or subtraction (lowest)

For complicated mathematical expressions, one can use parentheses ( ) to override the standard precedences.

**Warning:** Although brackets [ ] and braces { } are legitimate substitutes for parentheses in mathematical writing, they are not allowed in Maple to override precedences. Brackets are used for the data type *list* and braces are used for the data type *set*. More on data types later!

**Example:** Try them!

The importance of using parentheses can be seen in the difference between $a+b/c+d$ and $(a+b)/(c+d)$:

```
> a+b/c+d;
a + \frac{b}{c} + d
> (a+b)/(c+d);
\frac{a + b}{c + d}
```

Pay attention to the use of parentheses in these more complicated expressions.

```
> (a/(b+c) - d^2)/(3*e);
\frac{1}{3} \frac{a}{b + c} - \frac{d^2}{e}
```

The expression $\left(1 + \frac{3P}{h^2}\right) \left[1 - \left(\frac{a}{T}\right)^6\right]$ should be entered as follows:

```
> (1+(3*P)/(h^2))*(1-(a/l)^0.6);
\left(1 + \frac{3P}{h^2}\right) \left(1 - \left(\frac{a}{T}\right)^6\right)
```

More examples:

```
> c*a[1]*a[2]*sqrt(g*(h[1]-h[2])/a)/(3*sqrt(a[1]^2-a[2]^2));
c a_1 a_2 \sqrt{\frac{g (h_1 - h_2)}{s}}
\frac{3 + \sqrt{a_1^2 - a_2^2}}{s}
```
1.1. **MAPLE COMMAND LINES**

> \[ 2\pi \varepsilon \arccos\left\{ a^2 + b^2 - \frac{d^2}{2ab}\right\} \]

**Functions:**

Functions can be defined in Maple in several ways. A common format is to define a function as an “operator” as follows.

> \[ f := x \rightarrow \frac{\sin(x)}{x}; \]

\[ f := x \rightarrow \frac{\sin(x)}{x} \]

This operator assignment (the arrow \( \rightarrow \) is obtained by combining the dash \( - \) with the “greater than” symbol \( > \)) allows the use of the customary mathematical notation \( f(x) \).

> \[ f(\beta), f(1.57); \]

\[ \frac{\sin(\beta)}{\beta}, 0.6369424732 \]

> \[ f(a+b); \]

\[ \frac{\sin(a + b)}{a + b} \]

Once defined, a function can be evaluated or manipulated. For instance, one can take the derivatives:

> \[ D(f); \] \# the first derivative

\[ x \rightarrow \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} \]

> \[ D(D(f)); \] \# the second derivative

\[ x \rightarrow -\frac{\sin(x)}{x} - \frac{2\cos(x)}{x^2} + \frac{2\sin(x)}{x^3} \]

And the antiderivative and the definite integral:

> \[ g := x \rightarrow \tan(x); \]

\[ g := \tan \]

> \[ \text{int}(g(a),a); \]

\[ -\ln(\cos(a)) \]

> \[ \text{int}(g(s),s=\Pi/4..\Pi/3); \]

\[ \frac{1}{2} \ln(2) \]

> \[ \text{evalf}(\%); \]

0.3465735903

The alternative to the mapping notation is the simple assignment (and the \texttt{subs} command is required for evaluation):
> h := sin(x)/x;

> subs(x=beta,h),evalf(subs(x=1.57,h));

> diff(h,x);

And reusing the same name for a different function:

> h := x^4+4^x;

> int(h,x);

> int(h,x=0..1);

> evalf(%);

Notice the difference, in both cases, between the antiderivative (a function) and the definite integral (a number).

Graphing functions:

Study the following examples (the first combines upper and lower semicircles to obtain a circle of radius \( r \)), and then try to graph functions of your choice. Try both the mapping and assignment notation for your functions.

> y := sqrt(r^2-x^2);

> r := 2;

> plot([y,-y],x=-r..r);
1.1. MAPLE COMMAND LINES

> f := x -> .5*x^3 + 2*x^2 - 3*x;
\[ f := x \rightarrow \frac{1}{2}x^3 + 2x^2 - 3x \]

> plot(f(x),x=-6..3);

And a bit of algebra:

> factor(x^2-b^2);
\[ (x - b)(x + b) \]

> expand(%);
\[ x^2 - b^2 \]

> solve(x^2-2*x+3=0);
\[ 1 \pm \frac{1}{2} \sqrt{2} \]

> solve(a*x^2 + 3*b*y=5,x);
\[ \frac{\sqrt{-4a^2b^2 + 5a}}{a}, \frac{-\sqrt{-4a^2b^2 + 5a}}{a} \]

> fsolve(x=cos(x));
\[ .7390851332 \]

Try experimenting with solve and fsolve. What is the difference?

Remark: In the Windows environment, a Maple instruction is executed when the cursor is placed anywhere on that line followed by the Enter key. One can re-execute an instruction by moving the cursor back to the line with that instruction.

Getting help:

Maple has a built-in manual which explains commands and gives examples (often the examples are the best explanation!). If one knows (or guesses) a command, ?command_name opens a help window, for example:

> ?factor

The Help drop-down menu at the top of the screen lists the topics (either by using the Introduction and selecting the topic, or by doing a search).
1.2 Simple programming

Try typing the following line in a Maple worksheet (replace the name with your choice). To change line within the same prompt (>), hold Shift key and hit Enter.

```maple
> myname := proc()
print("Hi, my name is John Doe");
end proc;
```

Maple will echo the same instructions. Now type:

```maple
> myname();
```

and what appears?

That is a computer program. A program can be simply a sequence of instructions that is prepared for a well-defined computing objective, and it is stored and ready to be used repeatedly on a computer whenever it is invoked. Whoever writes a Maple program becomes a software engineer creating a new command for Maple.

A program automates the process of solving a computational problem, especially for those processes requiring the repetition of similar steps or to be repeated for different data. A computational task, in general, involves three components:

1. data (input),
2. computations,
3. results (output).

As a simple example, consider the formulas for the volume and surface area of a cylinder:

\[
\text{volume} = \text{base area} \times \text{height},
\]

\[
\text{surface area} = 2 \times \text{base area} + \text{height} \times \text{base circumference}
\]

If the base of the cylinder is a circular disk with radius \( r \), then

\[
\text{base area} = \pi r^2, \quad \text{base circumference} = 2\pi r.
\]

To compute the volume of the cylinder from the radius \( r \) and height \( h \) the components are:

- **data:** \( r \) and \( h \)
- **method:** step 1: base area \( b = \pi r^2 \)
  step 2: base circumference \( c = 2\pi r \)
  step 3: volume \( v = s h \)
  step 3: surface area \( s = 2b + ch \)
- **result:** volume \( v \) and surface area \( s \)
A program to implement a computational task will accept data as *input*, perform the calculations, and transmit the results as *output*, as shown in the diagram:

```
Data      input               Program               output          results
          r, h                  v, s
```

A program works like a black box for users who enter data as input and activate the program and then see the results. A programmer’s job is to produce the black box by writing programs.

Using the cylinder example, a Maple program can be created with a sequence of Maple statements, called a *source code* as follows.

```maple
cylinder := proc(r,h) # program definition, with input r and h
local b, c, v, s; # local variables
b := Pi * r^2; # calculate base area
c := 2 * Pi * r; # calculate base circumference
v := evalf( b * h ); # calculate volume v
s := evalf( 2*b + c*h ); # calculate surface area s
return v, s; # output v and s
end proc;
```

This program is named *cylinder*, and requires the arguments *r* and *h* as input for radius and height respectively. Local variables (i.e. variables only used inside the program) *b*, *c*, *v* and *s* are defined and the computation is carried out. Finally, the results are stored in the output variables *v* and *s* to be send back by the *return* command.

Maple programs can be edited using any text editor, such as Notepad in Windows, and saved as a text file. The freely downloadable text editor Vim is highly recommended. Besides its many good features, Vim provides syntax highlight for “.mpl” files and current line number that are quite useful in coding and debugging. (See Figure 1.3 for a screen shot of Vim.)

For example, one can save the cylinder program as file *cylinder.mpl* in a folder, say *f:/test*. Notice that editing a program is a separate process from Maple. A program can not be executable in Maple until it is loaded in a worksheet with a “read” command:

```
> read "f:/test/cylinder.mpl";

  cylinder := proc(r, h)
    local b, c, v, s;
    b := Pi * r^2; c := 2 * Pi * r; v := evalf(b * h); s := evalf(2 * b + c * h); return v, s
  end proc
```

Maple responds to the successful *read* command with the program showing in the worksheet.

Suppose the radius is 5.5m and the height is 8.2m. Executing the program:

```
> radius := 5.5; height := 8.2;

radius := 5.5
```

height := 8.2

> vol, surf := cylinder(radius, height);

vol, surf := 779.2720578, 473.4380130

In summary, Maple programming can be carried out in four steps:

- **Step 1:** Edit and save the program in Vim as a .mpl file.
- **Step 2:** Read the program into a Maple worksheet.
- **Step 3:** Prepare input.
- **Step 4:** Executing the program.

Those four steps are illustrated in Figure 1.3.

The input data (namely, \( r \) and \( h \) in Figure 1.3) are accepted according to the order in which they are given, not by the names storing the data. The data can also be supplied directly when calling a program without prior assignment.

For instance, if the radius is 3 feet and the height 2 feet:
1.2. SIMPLE PROGRAMMING

> vol, surf := cylinder(3,2);

\[
\begin{align*}
\text{vol} & = 56.54866777, \\
\text{surf} & = 94.24777962
\end{align*}
\]

If the order is reversed, the answer is obviously different! The program treats the data to be radius 2 and height 3:

> vol, surf := cylinder(2,3);

\[
\begin{align*}
\text{vol} & = 37.69911185, \\
\text{surf} & = 62.83185308
\end{align*}
\]

Similarly, the output items are received according to the order specified in the program. In this example, the output results are assigned to "vol" and "surf" by the order they appear in the Maple command "[> vol, surf := cylinder(2,3)]."

In summary, a Maple source code consists of the following essential components:

**Procedure definition statement:**

\[
\begin{align*}
\text{program\_name} & := \ proc( \text{input\_argument\_list} ) \\
\text{end proc;}
\end{align*}
\]

The user specifies the \text{program\_name} and the \text{input\_argument\_list} in parenthesis to define the program as a Maple procedure. The argument list can be empty. *Do not end this line with a semicolon if there is a local variable declaration line.*

**Local variable declaration**

\[
\text{local variable\_list;}
\]

This declaration is technically part of the procedure definition statement. Variables used inside the program in addition to the input arguments should be declared as local. If a variable is neither declared as local nor included as an input argument, Maple will treat it as a local variable and respond with a warning message when the procedure is loaded. As a good programming practice, all warning messages should be cleared through revising the source code.

**Sequence of Maple commands** that would carry out the computations when the input items carry the data.

**Output** the results using the return command in the form of

\[
\text{return output\_list;}
\]

The return statement terminates the computation, exits the procedure and sends which sends the results back to Maple.

**Closing statement**, i.e. the line \text{end proc;}

Comments. Anything written after the \# character in a line is a comment for human to read and will be ignored by the computer. A comment has no effect on the program execution or the computing result. It is useful to include comments to make it easier for the programmer and others to understand the program. First time programmers might be surprised how quickly one forgets why a sequence of program statements were written in a certain way. Leaving comments in the source code can help reminding the programmer and collaborators the reasoning behind the statements and provide clues on how to improve the program or the way to correct mistakes.

There are several other optional components of a Maple procedure. Those options are omitted in this since they are useful but not essential for the scope of this text. Detailed documentations of program components are included Maple manuals and a short on-line help can be accessed by entering \?proc at the Maple prompt >.

Although a program can be directly typed into Maple at a prompt >, writing the source code as a separate text file is a standard programming practice in all platforms besides Maple. There are many advantages in doing so as well. A source code saved as a text file becomes a form of portable software that can be used in any Maple worksheet whenever needed, and can be used by other Maple programs as a “subroutine”. Moreover, it is common in scientific computing to use programs written with hundreds or thousands of lines, making it cumbersome or even impractical to type it in a worksheet. Editing source codes as a text file is also a valuable learning process for a taste of practical software engineering.

It is rare to finish writing the first draft of a source code without errors, particularly for long and complicated programs. Almost all computer programs require multiple rounds of testing and revision. When loading a program text file, Maple can identify some obvious syntax errors and respond with an error message. Sophisticated programming errors are called bugs that can be hidden deeply in the source code and emerge during testing or applications. When an output result is suspicious, a programmer must trace the flow of the command sequence to catch the error in the source code. Once a bug is identified and the source code is revised accordingly, the text file needs to saved and reloaded into Maple by executing the read statement again so that the new version can be tested.

Debugging can be painstaking, time consuming, and even frustrating for programmers. It is particularly so for beginners. So readers be warned.

Example 1.1 The future value of an annuity. Suppose a periodic deposit of $R is made to an account that pays an interest rate $i per period, for $n$ periods. The future value $S$ of the account after the $n$ periods will be

$$S = \frac{R((1 + i)^n - 1)}{i}.$$ 

Write a program to calculate the worth of a retirement plan at the end of the term if the plan pays an annual rate $A$ and requires monthly deposits of $R$ for $y$ years. Use the program to
1.2. SIMPLE PROGRAMMING

calculate the future value of a plan with a monthly deposit of $300 and an annual interest rate of 12% over a 30 year period.

Solution: The computational task involves the following input items:

\[ R \quad \text{— deposit} \]
\[ A \quad \text{— annual rate (instead of } i \text{)} \]
\[ y \quad \text{— the number of years} \]

Type the following program using a text editor, say VIM, and save the file as `a:future.mpl`. As mentioned earlier, the single quote (‘) in the print command is usually found on the upper-left corner of the keyboard below the ESC key. A common mistake is using the other quote (‘) found next to the Enter key.

```maple
# Program to compute the future value of an annuity
# Input: R: monthly deposit ($)
# A: annual interest rate (0.12 for 12\%)
# y: number of years
#
future := proc(R,A,y) # defines the program "future"
# with arguments R, A, y
local i, n, S; # define local variables
i := A/12; # monthly interest rate is
# 1/12 of the annual rate
n := y*12; # total number of months
S := R*((1+i)^n-1)/i; # calculation of future value
return S; # print the result
end proc; # the last line of the program.
```

Remarks:

(1) Do not put the semi-colon at the end of `proc(...)`, this statement ends with `local ...;`.

(2) At end of each definition or action inside the program, remember that Maple needs ‘;’.

(3) Especially, type the semicolon after `end proc`.

(4) Comments should be included generously in the program using ‘#’, as in the example.

(5) Notice how the readability of the program improves with indentation, blank lines, and well-placed comments. Maple can handle a program written all on one line, but no one would want to have to figure it out!
Now to read the file into Maple:

```maple
> restart;
> read "f:future.mpl";

future := proc(R, A, y)
    local i, n, S;
    i := 1/12 * A; n := 12 * y; S := R * ((1 + i)^n - 1)/i; return S
end proc
```

Input data can be defined by assignment, say:

```maple
> Deposit := 300; AnnualRate := 0.12; Years := 30;
Deposit := 300
AnnualRate := 0.12
Years := 30
```

The program is to be executed using the input items defined above. Notice that the variables \( R, A, y \) in the program source code are similar to local variables, which have meaning only within the program. When the program is called, the variables \( R, A, y \) are substituted by the input data according to the order it is provided, as below.

```maple
> F := future(Deposit, AnnualRate, Years);
F := 0.1048489240 \times 10^7
```

Or, one may directly input the arguments:

```maple
> F := future(300, .12, 30);
F := 0.1048489240 \times 10^7
```

**Example 2:** Write a program that computes the two solutions of the general quadratic equation

\[
ax^2 + bx + c = 0
\]

with the quadratic formula. Use the program to solve

\[
3x^2 - 5x + 2 = 0 \quad \text{and} \quad x^2 - 3x + 2 = 0
\]

**Solution:** Type the following program using a text editor (Notepad, perhaps) and save the file as `f:quad.mpl`.

```maple
quad := proc(a, b, c) # define the program
    local sol1, sol2; # local variables
    sol1 := (-b + sqrt(b^2 - 4*a*c))/(2*a); # calculate solution 1
    sol2 := (-b - sqrt(b^2 - 4*a*c))/(2*a); # calculate solution 2
    return sol1, sol2; # output results
end proc; # end of program
```

Now load the program.
1.2. SIMPLE PROGRAMMING

> read "f:quad.mpl";

quad := proc(a, b, c)
local sol1, sol2;
sol1 := 1/2 * (-b + sqrt(b^2 - 4*a*c))/a;
sol2 := 1/2 * (-b - sqrt(b^2 - 4*a*c))/a;
return sol1, sol2
end proc

To solve \(3x^2 - 5x + 2 = 0\) the coefficients are \(a = 3\), \(b = -5\), and \(c = 2\):

> sln1, sln2 := quad(3, -5, 2);

\[\text{sln1, sln2} := 1, \frac{2}{3}\]

To solve \(x^2 - 3x + 2 = 0\):

> a := 1; b := -3; c := 2;
> s1, s2 := quad(a, b, c);

\[s1, s2 := 2, 1\]

**Example 3:** The velocity \(v(t) = -9.81t + v_0\) (m/sec) and height \(h(t) = -4.905t^2 + v_0t + h_0\) (m) are familiar formulas for an object moving under the influence of gravity, where \(v_0\) and \(h_0\) are the initial velocity and height when \(t = 0\) respectively. Write a program that calculates the height above the ground and velocity of an object thrown vertically after a certain number of seconds. The input should be the initial height and velocity, as well as the elapsed time.

**Solution:** The source code

```maple
height := proc(h0, v0, t)
local v, h;
v := -9.81*t + v0;
h := -4.905*t^2 + v0*t + h0;
return v, h
end proc;
```

Execute the program with an initial height of 100 m and an initial velocity of 5 m/s (upwards) over 3 seconds:

> read "f:height.mpl";

\[\text{Velocity, Height} := \text{height}(100, 5, 3);\]

\[\text{Velocity, Height} := -24.43, 70.855\]
1.3 Conditional statements - Branching

Each program considered so far in examples has implemented a few simple formulas as a chain of instructions. It is often necessary to carry out different calculations depending on specific conditions (numerical values, logical truth of a statement, etc.). The programming mechanism such as *branching* is accomplished with the `if` ... `then` ... `else` ... `end` if statement block.

Consider the following Maple commands. Type them into a Maple worksheet (using Shift-Enter to change lines) and change the values of `a` and `b`.

```maple
> a := 2: b := 5:
if a < b then
  print("a is less than b");
else
  print("a is not less than b");
end if;
```

What happens?

"a is less than b"

**Example 1.** A piecewise function:

\[ f(x) = \begin{cases} 
- (x - 1)^2 + 2 & x < 0 \\
2x + 1 & 0 \leq x \leq 2 \\
5e^{-(x-2)^2} & x > 2 
\end{cases} \]

can be defined with a Maple program

```maple
func := proc(x) # program definition
  if x < 0 then
    return -(x-1)^2 + 2; # case for x < 0
  elif x <= 2 then
    return 2*x + 1; # case for 0 <= x <= 2
  else
    return 5*exp(-(x-2)^2); # case for x > 2
  end if;
end proc;
```

> read "f:func.mpl";

```maple
func := proc(x)
  if x < 0 then return -(x-1)^2 + 2
  elif x <= 2 then return 2*x + 1
  else return 5*exp(-(x-2)^2)
  end if
end proc
```
The program distinguishes three possible options that are separated by `else` and `elif` instructions. The Maple statement `elif` stands for “else if”, which can be multiple times as needed to add more branches.

```maple
> func(-3); func(1); func(3);

−14
3
5e(−1)
```

Such a piecewise function can also be plotted in a graph

```maple
> plot(func,-1..3);
```

---

**Structure of if blocks**

The piecewise function example illustrated the `if...elif...else...end if` block. In general, the if block has the following structures:

1. If there is a single condition, statements are executed only if the condition is `true`. Otherwise (if the condition is not true because it is either `false` or `FAIL`), the statement block is skipped:

   ```maple
   if condition then
   statement block
   end if;
   ```

2. If there are two branches given by one condition then the two statement blocks are separated by the `else` command.

   ```maple
   if condition then
   statement block 1
   else
   statement block 2
   end if;
   ```
Entering this if-block, the computer verifies the condition, if the answer is “true”, the statement block 1 will be executed and the statement block 2 will be skipped. On the other hand, if the answer to the condition is “false”, the computer will skip the statement block 1 and execute statement block 2.

3. If there are many branches for different conditions, the **elif** (else if) statement is used:

```plaintext
if condition 1 then
    statement block 1
elif condition 2 then
    statement block 2
    ...
    ...
elif condition n then
    statement block n
else
    final statement block
end if;
```

When this if-block runs, the Maple verifies condition 1 first. If it is met (true) only statement 1 is executed. If condition 1 is not true, the Maple verifies condition 2, and continues this way until a condition k returns true for the first time. Statement block k is executed and all other statements are skipped. If all conditions are not true, the Maple executes only the the final statement block. The **else** and final statement block are optional.

Remarks:

- Do not put a semicolon after **then** or **else**.
- It is good formatting style to align the **if**, **elif**, **else** and **end if** of the same if block, and have a 3-space indentation for each statement block. This makes the if-block much easier to read!
- Conditions must be verifiable by Maple or a ‘boolean error’ message is returned. More on this later.

Example 2. Suppose that in a certain course with five 100-point exams the course grade will be determined by the scale: A for 90% or above, B for 80% or above but less than 90%, C for 70% or above but less than 80%, D for 60% or above but less than 70% and F for any score below 60%. Write a program that with the input of the total points received on the five tests, prints (1) the course grade, (2) a borderline warning if the score is within 1% of the next higher grade.

Solution:
1.3. CONDITIONAL STATEMENTS - BRANCHING

```maple
grade := proc(points)
local g;

    if points >= 450 then
        g := "A";
    elif points >= 400 then
        g := "B"
        if points > 445 then g := "B, but borderline A"; end if;
    elif points >= 350 then
        g := "C"
        if points > 395 then g := "C, but borderline B"; end if;
    elif points >= 300 then
        g := "D"
        if points > 345 then g := "D, but borderline C"; end if;
    else
        g := "F"
        if points > 295 then g := "F, but borderline D"; end if;
    end if;

    return g;
end proc;

> read "f:grade.mpl":
> grade(367);
    "C"
> grade(397);
    "C, but borderline B"
> grade(310);
    "D"
```

**Example 3.** (Techniques in this example will be used later in probability simulations). Suppose we number a standard deck of cards from 1 to 52, such that

- hearts: 1-13
- spades: 14-26
- diamonds: 27-39
- clubs: 40-52

and in each suit, cards are numbered in the following order:

- ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king

Write a program that prints the number associated with the card given the card name (e.g. 3, spade, or jack, diamond). We will need to find a way to distinguish between cards with names and cards identified by number. For this we use Maple’s ability to distinguish data types.

**Data types**

Maple classifies data by type. (Try `?type` for the help entry). Data types are covered in more detail in section 5.2. We need to specify `numeric` and `string` types in this problem. Card values 2, 3, ..., 10 are numeric. A sequence of characters within a pair of double


quotes " " is called a string. A string carries no numeric value. For example, “ace”, “jack”, “queen”, and “king” are strings. By comparison, ace, jack, queen and king without quotes are variable names that can be assigned values. The function \texttt{type(expression, type name)} will return either \texttt{true} or \texttt{false} depending on whether the expression matches the type name. For instance:

\begin{verbatim}
> type(3.5,string);                  false
> type(3.5,numeric);               true
> type(queen, string);              false
> type("queen",string);            true
> Value:=6: type(Value,numeric);    true
> a:="king": type(a,string);        true
\end{verbatim}

This example program also deals with input error in an elegant way.

Now the program:

\begin{verbatim}
# program that numbers a card in a deck from 1-52
# input: rank --- "ace",2,3,...,10,"jack","queen", or "king"
#       suit --- "heart", "spade", "diamond", or "club"
# output: n --- the number of the card (1-52)
#
CardNumber := proc(rank, suit)
    local order, number;
    if type(rank,numeric) then
        order := rank;
    elif type(rank,string) then
        if rank = "ace" then order := 1;
            elif rank = "jack" then order := 11;
            elif rank = "queen" then order := 12;
            elif rank = "king" then order := 13;
            else return "Error: wrong input for rank";
            end if
    else
        return "Error: wrong input data type for rank";
    end if;
    if suit = "heart" then number := order;
    elif suit = "spade" then number := 13 + order;
    elif suit = "diamond" then number := 26 + order;
    elif suit = "club" then number := 39 + order;
    else
        return "Error: wrong input for suit";
    end if;
\end{verbatim}
1.4 Common errors

Beginning Maple programmers often make simple errors that are easy to avoid. Some of those errors are listed below.

- **Forgetting the colon or semicolon at the end of an instruction.** If a colon/semicolon is missed inside the program, Maple would treat the line as incomplete and continue reading the next line. Consequently, an error message may point to the line below. Missing a colon/semicolon at the "end proc" will get a message "unexpected end of input".

- **Forgetting to close an if-statement with the end if** (and coming later, closing the do-loop with the end do).

- **Boolean error in an if-statement.** An if-statement must be written so that the logical test can be carried out by Maple. For example, Maple cannot verify that $\sqrt{5} > 1$. When variable $s$ carries $\sqrt{5}$, the statement such as

  \[
  \text{if } s > 1 \text{ then}
  \]

  will return a message saying “Error, cannot determine if this expression is true or false: $1 < 5 \wedge (1/2)$.” To resolve the problem one could use

  \[
  \text{if evalf(s) > 1 then}
  \]

  instead.

- **Illegal use of a formal parameter.** For example,

  ```maple
  > formal := proc(a)
    a := a + 2 # attempt assignment to a that is an input argument
    end;
  > formal(3.5);
  ```
CHAPTER 1. FUNDAMENTALS

Error, (in formal) illegal use of a formal parameter

Maple does not allow input variables to be changed inside a program (as a precaution for when programs are nested). This error message appears during execution of the program. The fix is to pass the values of the input to local variables:

\[
\text{> formal:=proc(a)}
\begin{align*}
\text{local b;} \\
b &:= a; \quad \# \text{pass the value from a to b} \\
b &:= b + 2;
end;
\end{align*}
\]

1.5 Writing a project report on a Maple worksheet

A Maple worksheet can be used as a word processor to combine mathematical computing with scientific writing. When the cursor is at the prompt [>] in a classic worksheet, clicking the T button of the toolbar makes the prompt disappear so that text can be typed in that line. In such a text mode, clicking the Σ button enables the user to enter nonexecutable standard mathematical expressions in a text region.

The report editing features of a Maple worksheet are intuitive. A sample project report is shown in Figure 1.4.

1.6 Exercises

1. Solving a cubic equation. One of the three solutions to the cubic equation \( x^3 + ax^2 + b = 0 \) is given in the following formula

\[
x = \frac{\sqrt[3]{12\sqrt{12a^3b + 81b^2} - 8a^3 - 108b}}{6} + \frac{2a^2}{3\sqrt[3]{12\sqrt{12a^3b + 81b^2} - 8a^3 - 108b}} - a \frac{3}{3}
\]

Write a program that, for input \( a \) and \( b \), output this solution \( x \) in floating point number, and then verify that \( x \) satisfy the equation approximately. Suggestion: Decompose the formula in several steps and recycle the repeated parts of the formula.

2. Physics formula (Physics) A formula related to photochemistry in intermittent light is given as follows

\[
r = \frac{1}{p+1} \left\{ 1 + \frac{1}{t} \ln \left[ 1 + \frac{pt}{2p + 2 + \tanh t \cdot \frac{p}{p + 1}} \right] \right\}
\]

Write a program that, for input \( p \) and \( t \), outputs the values of \( r \).
1.6. EXERCISES

Problem 1. Write a program to compute and output the volume and surface area of a circular cylinder using radius and height as input.

Solution. Source code Vim image

![Cylinder Maple Code]

Test execution:
\[
\begin{align*}
> & \text{read("c:/temp/cylinder.mpl"):} \\
> & \text{voll, surf1 := cylinder(10, 25);} \\
& \quad \text{voll, surf1} := 7853.981635, 2199.114858 \\
> & \text{voll, surf2 := cylinder(6.6, 7.8);} \\
& \quad \text{voll, surf2} := 1067.412653, 597.1539317
\end{align*}
\]

Problem 2. Write a program to compute/output the two solutions of a quadratic equation \(a \cdot x^2 + b \cdot x + c = 0\) with input \(a, b, c\).

Solution. Source code Vim image

![Quadratic Maple Code]

Test execution:
\[
\begin{align*}
> & \text{read("c:/temp/quad.mpl"):} \\
> & \text{sl, s2 := quad(3,4,-5);} \\
& \quad s1, s2 := -\frac{2}{3} + \frac{1}{3} \sqrt{19}, -\frac{2}{3} - \frac{1}{3} \sqrt{19}
\end{align*}
\]

Figure 1.4: A sample project
3. **Physics formula (Physics)** A formula related to Lorentz-force on a particle is given as follows

\[
v = \frac{1}{m} \frac{p - \frac{eA}{m}}{\sqrt{1 + \left(\frac{q - \frac{eB}{m^2c^2}}{m^2c^2}\right)^2}}
\]

Write a program that, for input \( m, p, q, A, B, m, e, \epsilon \) and \( c \), outputs the values of \( v \).

4. **Physics formula (Physics)** A formula related to Lorentz-force on a particle is given as follows

\[
L = \frac{-mc^2 + \frac{eA}{m} - \frac{eA}{m} + \epsilon \phi}{\sqrt{1 + \left(\frac{q - \frac{eB}{m^2c^2}}{m^2c^2}\right)^2}}
\]

Write a program that, for input \( m, p, q, A, B, m, \epsilon, \phi \) and \( c \), outputs the values of \( L \).

5. **Physics formula (Physics)** A formula related to Sommerfeld’s theory is given as follows

\[
E = mc^2 \left[ 1 - \frac{1}{\sqrt{1 + \frac{\alpha^2}{n_r + \sqrt{n_r^2 - \alpha^2}}} - \frac{1}{\sqrt{1 + \frac{\alpha^2}{n_\theta + \sqrt{n_\theta^2 - \alpha^2}}}} \right]
\]

Write a program that, for input \( m, c, \alpha, n_r, \) and \( n_\theta \), outputs the values of \( E \).

6. **Physics formula (Physics)** A formula related to forced oscillations is given as follows

\[
u = a^3 \frac{\sin \omega t}{b \omega \sqrt{\omega}} \left[ \left( \cosh \frac{\sqrt{\omega} x}{a} - \cos \frac{\sqrt{\omega} x}{a} \right) \left( \sinh \frac{\sqrt{\omega} l}{a} + \sin \frac{\sqrt{\omega} l}{a} \right) \right]
\]

Write a program that, for input \( a, b, \omega, x, \) and \( l \), outputs the values of \( u \).

7. **A mortgage problem** Let \( A \) be the amount of a mortgage, \( n \) the total number of payments, \( i \) the interest rate per period of payment. Then the mortgage payment \( R \) per period is given by the formula:

\[
R = \frac{Ai}{1 - (1 + i)^{(-n)}}
\]

- Write a program that, for input: \( p, r, y, d \) as price of the purchase, annual percentage interest rate, number of years, and down payment percentage respectively, calculates and prints the monthly payment amount.
- Use your program to calculate the monthly payment of a $180,000 house, 20% down, 7.75% annual interest rate for 30 years.
- Use your program to calculate the monthly payment of a $15,000 car, 10% down, 9.25% annual interest, rate for 5 years.

8. **Geometry of a circle.** Write a program that displays the radius, circumference, and area of a circle with a given diameter.
9. **Parallel resistors, part I.** When three electrical resistors with resistance \( R_1 \), \( R_2 \), and \( R_3 \) respectively are arranged in parallel, their combined resistance \( R \) is given by the formula

\[
\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}
\]  

(1.1)

Write a program that calculates \( R \) given the input of \( R_1 \), \( R_2 \) and \( R_3 \).

10. **Parallel resistors, part II.** Again, consider three electrical resistors with resistance \( R_1 \), \( R_2 \), and \( R_3 \) respectively arranged in parallel. However, if one the resistor has zero resistance, the formula (1.1) would not work due to division by zero. The combined resistance is zero in this case. Furthermore, if one of the input \( R_1 \), \( R_2 \) or \( R_3 \) is negative, the application is invalid. Write a program to cover all those cases.

11. **Number Magic: Age test**\(^1\) (**Programming skill exercise**) Write a program that carries out the number magic game step-by-step: Input: the age of yours or someone else’s. Multiply the age by 12, then add the mysterious number 2856, then divide the result by 3, then divide the result by 4, then subtract your age from the result. Output the final number. Use the program to play the game several times with different input numbers. What is the result?

12. **Number Magic: Think-a-digit**\(^2\) (**Programming skill exercise**) Write a program that carries out the number magic game step-by-step: Think about a digit from 1 to 9 as input. Repeat the digit three times (for instance, if the input digit is 5, make 555), then divide the result by 3, then divide the number by the digit you thought of. Output the final number. Use the program to play the game several times with different input numbers. What is the result?

13. **Number Magic: A peculiar series**\(^3\) (**Programming skill exercise**) Write a program that carries out the number magic game step-by-step: Think about an integer of any size as input. Consider the input number as the first number. Create the second number by adding 7 to the first number. Add 7 again to get a third number. Add 7 once more to get a fourth number. Multiply the two numbers at both ends to get a product. Multiply the two middle numbers to get another product. Finally, subtract the smaller product from the larger and output the difference. Use the program to play the game several times with different input numbers. What is the result?

14. **Cost calculation.** (**Finance**) A small company offers a car rental plan: $20.00 a day plus $0.10/mile for up to an average of 200 miles per day. There is no additional cost per day for more than 200 miles (e.g., 3 days with 500 miles cost $110 while 3 days with 800 miles cost $120). Write a program that calculates the total rental fee given the number of days rented and the total distance traveled. Show results for 5 days with 800 miles, and 5 days with 1450 miles.

15. **Cost calculation II.** A travel agency offers a Las Vegas vacation plan. The price for a single traveler is $400. If a traveler brings companions (8 or fewer), each companion receives a 10% discount. A group of 10 or more receives a 15% discount per person (including the first traveler). Write a program that calculates the total price of the trip given the number of travelers. Show results for 1, 8, 9, 10 and 12 travelers.

16. **Salary calculation.** (**Finance**) A company pays its sales people according to years of experience and total sales generated. For a sales person of experience up to three years, the monthly salary is $1,000 plus 10% of sales above $10,000. When he/she has four to 9 years of experience, the monthly salary will be $2,000 plus 15% of the sales above $20,000. With ten or more years of experience, the monthly salary increases to $3,000 plus 20% of the sales above $30,000. Write a program that, for input years of experience and amount of total sales, output the monthly salary. Use nested if blocks.

---


17. **Salary calculation** (*Finance*) A company pays its employees by levels of seniority and hours worked. The regular hourly wage is $10, $12, $15 and $20 for levels 1, 2, 3 and 4 respectively for the first 40 hours of work in a week. Hours over 40 in a week will add 50% over time pay. Write a program that, for input level of seniority and hours worked in a week, calculate and output the salary of the week. Use nested if blocks.

18. A fitness camp sets kids of 10, 11 and 12 years old to distance running. Those 10-year-olds must start with one mile per day. For kids of age 11, the starting requirement is 1.5 miles per day. If a kid is 12 year old, he or she must start with 2 mile a day. For every seven days in the camp, the running requirement increases by 1 mile per day. Write a program that, for input age (of a kid) and the number of days in the camp, calculate and output the total accumulated miles the kid has run up to that day. Use nested if blocks.

19. **Temperature conversion.** Write a program that converts temperature values from Fahrenheit to Celsius or vice versa.

20. **Avoiding loss of significance in the quadratic formula** The loss of significance that can occur from the direct application of the standard quadratic formula to solve \( ax^2 + bx + c = 0 \) was discussed in this chapter. For example, we looked at the case \( b > 0 \) and claimed that \( x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \) can be replaced by \( x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \) which has no added roundoff error.

   Show that \( x_1 = \frac{c}{ax_2} \) and that \( x_2 = \frac{c}{ax_1} \). To show this rationalize the numerator (for example, for \( x_1 \) multiply and divide by \( -b - \sqrt{b^2 - 4ac} \) and simplify. A simpler way to demonstrate this is to start with \( ax^2 + bx + c = a(x - x_1)(x - x_2) \) (why is this true?) and show that \( c = ax_1x_2 \). The alternate formula for \( x_1 \) should be used if \( b > 0 \) in place of the standard form, and the one for \( x_2 \) when \( b < 0 \).

21. **Avoiding loss of significance in the quadratic formula, part 2** Modify the program `quad` (page 16) to incorporate the formulas from the previous problem (be careful, there are two cases: \( b \geq 0 \) or \( b < 0 \)).

22. **Real solutions to a quadratic equation.** For the general quadratic equation \( ax^2 + bx + c = 0 \) with \( a \neq 0 \), the discriminant is \( \Delta = b^2 - 4ac \), such that
   - if \( \Delta > 0 \), the equation has two real solutions;
   - if \( \Delta = 0 \), the equation has a single real solution (a double root) \( x = -\frac{b}{2a} \);
   - otherwise (\( \Delta < 0 \)) the equation has no real solutions.

   Write a program that determines the number of real solutions from input \( a, b, c \) using the discriminant. If there are one or two solutions, return the solution(s). Otherwise, return a message: “There are no real solutions.”

23. **More on quadratic equations.** The quadratic equation problem above can be extended to a more general case: if \( a = 0 \), the equation is reduced to a linear equation and there is only one real solution \( x = -\frac{c}{b} \) if \( b \) is nonzero. If both \( a = 0 \) and \( b = 0 \) the equation is invalid. Write a program that calculates the real solutions of a quadratic equations for all these cases.

24. **Card identification.** (See Example 3 in Section 1.3) Write a program that given a card number identifies the corresponding card name from a deck of 52 playing cards.

   Sample results
   ```
   > read('a:numcard.mpl');
   > numcard(1);
   
   The card is the ace of hearts.
   ```
1.6. EXERCISES

> numcard(25);

The card is the queen of spades.

> numcard(30);

The card is the 4 of diamonds.

> numcard(0);

Error: the input must be between 1 and 52.

> numcard(55);

Error: the input must be between 1 and 52.

25. **Leap year.** Before 1752, there is a leap year for every four years according to the Julian calendar. Namely, if the year is divisible by 4, there are 29 days in February. Since the Gregorian calendar is adopted in England in 1752, the leap year rule has been changed as follows: If a year is divisible by 4, it is a leap year except if it is divisible by 100. If the year is divisible by 400, however, it is again a leap year. Write a program that, for input year, outputs `true` or `false` depending on the year to be a leap year or not.

26. **Tax schedule.** The following is the 1996 federal income tax schedule. Write a program to calculate taxes, given as input `Status` and `TaxableIncome`.

<table>
<thead>
<tr>
<th>Status 1 (single)</th>
<th>TaxableIncome</th>
<th>Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 23,350</td>
<td>0.15*TaxableIncome</td>
<td></td>
</tr>
<tr>
<td>23,350 – 56,550</td>
<td>3,502.50 + 0.28*(TaxableIncome - 23,350)</td>
<td></td>
</tr>
<tr>
<td>56,550 – 117,950</td>
<td>12,798.50 + 0.31*(TaxableIncome - 56,550)</td>
<td></td>
</tr>
<tr>
<td>117,950 – 256,500</td>
<td>31,832.50 + 0.36*(TaxableIncome - 117,950)</td>
<td></td>
</tr>
<tr>
<td>256,500 – up</td>
<td>81,710.50 + 0.396*(TaxableIncome - 256,500)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status 2 (Married filing jointly or Qualifying Widow(er))</th>
<th>TaxableIncome</th>
<th>Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 39,000</td>
<td>0.15*TaxableIncome</td>
<td></td>
</tr>
<tr>
<td>39,000 – 94,250</td>
<td>5,850.00 + 0.28*(TaxableIncome - 39,000)</td>
<td></td>
</tr>
<tr>
<td>94,250 – 143,600</td>
<td>21,320.00 + 0.31*(TaxableIncome - 94,250)</td>
<td></td>
</tr>
<tr>
<td>143,600 – 256,500</td>
<td>36,618.50 + 0.36*(TaxableIncome - 143,600)</td>
<td></td>
</tr>
<tr>
<td>256,500 – up</td>
<td>77,262.50 + 0.396*(TaxableIncome - 256,500)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status 3 (Married filing separately)</th>
<th>TaxableIncome</th>
<th>Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 19,500</td>
<td>0.15*TaxableIncome</td>
<td></td>
</tr>
<tr>
<td>19,500 – 47,125</td>
<td>2,925.00 + 0.28*(TaxableIncome - 19,500)</td>
<td></td>
</tr>
<tr>
<td>47,125 – 71,800</td>
<td>10,660.00 + 0.31*(TaxableIncome - 47,125)</td>
<td></td>
</tr>
<tr>
<td>71,800 – 128,250</td>
<td>18,309.25 + 0.36*(TaxableIncome - 71,800)</td>
<td></td>
</tr>
<tr>
<td>128,250 – up</td>
<td>38,631.25 + 0.396*(TaxableIncome - 128,250)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status 4 (Head of household)</th>
<th>TaxableIncome</th>
<th>Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 31,250</td>
<td>0.15*TaxableIncome</td>
<td></td>
</tr>
<tr>
<td>31,250 – 80,750</td>
<td>4,687.50 + 0.28*(TaxableIncome - 31,250)</td>
<td></td>
</tr>
</tbody>
</table>
Sample results
> read('a:taxschdl.mpl')
> status:=2: TaxableIncome:=56890:
> taxschdl(status,TaxableIncome); 10859.20
> status:=4: TaxableIncome:=35280:
> taxschdl(status,TaxableIncome); 5975.90
> status:=1: TaxableIncome:=2590000:
> taxschdl(status,TaxableIncome); .1005776500e7

1.7 Projects

1. **Linear inequalities.** Write a program for solving linear inequalities

\[ ax > b, \ ax < b, \ ax \geq b, \ ax \leq b. \]

The program accepts input items \(a, x, s\) and \(b\), where \(a\), and \(b\) are numbers, \(x\) is the name of the unknown, and \(s\) represent a character string that is ">", ">=", "><", or "=<". The program must deal with cases where \(a\) can be positive, negative or zero. Sample input and output:

> ineq(-3,x,">", 12);

\[ x, "\leq", -4 \]

> ineq(0,y,">="; 12);

"no solutions"

> ineq(0,y,"<", 2);

\[ y, "can be any real number" \]

2. **Tuition calculation.** A state university charges semester tuitions differently from in-state and out-of-state students. For in-state students, the tuition is $250 per credit hour for the first 12 hours and $50/hour after 12. For out-of-state students, the tuition is $450/hr for the first 12 hours and $100 after 12. Write a program that that calculate the tuition according to the residence status and the credit hours. Your program accepts two input items. The first input item is character variable representing “in-state” or “out-of-state”, and the second input item is a positive integer representing the number of credit hour. The program should output an error message if the first input item is spelled incorrectly or the credit hour is given as a negative number.
3. **Quadratic Equations.** Build on what has been done in the exercises on the quadratic equation to write a program that solves all cases: invalid input, linear case, real roots (single or double), and complex roots. Use the formulas that avoid roundoff error for the real roots.

Hand in:

- program code (text file);
- a Maple worksheet with output similar to what is given below, including comparison with the quad program (using the standard formulas) (page 16);
- a paragraph explaining the cases considered, the derivation of the formulas used, and the roundoff error you observed. The last two output lines can help you discuss roundoff, since $c = ax_1x_2$. Why?

```maple
> quad2(0,0,1);
This is not a valid equation.
> quad2(0,2.,-1);
This is a linear equation with solution x=.5000000000.
> quad2(1,0,1);
The complex roots of this equation are: 
    1, -1
> quad2(1,1,1);
The complex roots of this equation are: 
    -1/2 + 1/2 I \sqrt{3}, -1/2 - 1/2 I \sqrt{3}
> quad2(1,2,1);
There is one real root of multiplicity 2, x=.1.
> quad2(3.1,1321,.25);quad(3.1,1321,.25);
The real roots of the equation are 
x=-.0001892506518 and x=-426.1288430.
The solutions are -.0001891935484 and -426.1288430.
> quad2(3.1,-1321,.25);quad(3.1,-1321,.25);
The real roots of the equation are 
x=426.1288430 and x=.0001892506518.
The solutions are 426.1288430 and .0001891935484.
> 3.1*426.1288430*.1892506518e-3;
.2499999999
```
4. **Cubic equations.** The Cardano formula for solving the general cubic equation \( x^3 + ax^2 + bx + c = 0 \) of real coefficients is given as follows: To start, compute

\[
Q = \frac{3b-a^2}{9}, \quad R = \frac{9ab-27c-2a^3}{54}, \quad P = Q^3 + R^2.
\]

Then there are two cases depending on the sign of \( P \):

**Step 1.** If \( P \geq 0 \), then there are one real solution and two complex solutions:

\[
\begin{align*}
    x_1 &= -\frac{a}{3} + S + T, \\
    x_2 &= -\frac{a}{3} - \frac{S + T}{2} - i\sqrt{3}\frac{S - T}{2}, \\
    x_3 &= -\frac{a}{3} - \frac{S + T}{2} + i\sqrt{3}\frac{S - T}{2},
\end{align*}
\]

where

\[
S = \text{sign}(R + \sqrt{P}) \sqrt[3]{|R + \sqrt{P}|},
\]

\[
T = \text{sign}(R - \sqrt{P}) \sqrt[3]{|R - \sqrt{P}|}.
\]

**Case 2** Otherwise,

\[
\begin{align*}
    x_1 &= 2\sqrt{-Q} \cos \frac{\theta}{3} - \frac{a}{3}, \\
    x_2 &= 2\sqrt{-Q} \cos \frac{\theta + 2\pi}{3} - \frac{a}{3}, \\
    x_3 &= 2\sqrt{-Q} \cos \frac{\theta + 4\pi}{3} - \frac{a}{3},
\end{align*}
\]

where

\[
\theta = \arccos \frac{R}{\sqrt{-Q^3}}.
\]

Write a program that, for input \( a, b, \) and \( c \), and outputs all three solutions. Verify that \( x_1, x_2 \) and \( x_3 \) are indeed the (approximate) solutions to the cubic equation.

---

\(^4\)The notation \( \text{sign}(x) \) stands for the sign function defined by \( \text{sign}(x) = 1 \) if \( x \geq 0 \) and \( \text{sign}(x) = -1 \) otherwise.
Chapter 2

Loops. Part I

2.1 The “for-do” loop

One the greatest strengths of machines over human is to perform the same task repeatedly and tirelessly with tremendous efficiency. Such repetitions are carried out as “loops” in all programming languages in various forms. The basic loop in Maple is in the form of “for-do” that can be illustrated in the example of checking integers one by one using the Maple function “isprime” and print out prime numbers:

```
> for k from 1 to 100 do
>   if isprime(k) then
>     print(k)
>   end if
> end do;
```

2
3
5
7
...

In this example, the inner block “if isprime(k) then print(k) end if” is repeated for \( k \) from 1 to 100. Since prime numbers are odd, the computation can be more efficient by skipping even numbers using the “by” option:

```
> for k from 1 by 2 to 100 do
>   if isprime(k) then print(k) end if
> end do;
```

The result is the same. The syntax of the “for-do” loop is
for loop_index from start_value by stepsize to end_value do
   [ block of statements to be repeated ]
end do;

The loop index (e.g. \( k \) in examples above) takes values starting from the initial value, increasing by the step size, to the ending value that need to be given by the programmer in those blanks \([\ldots]\). The common exceptions in this syntax is to omit “by 1” and/or “from 1” as they are defaults.

The loop can go “backward” by providing a negative stepsize, such as

```
for i from 20 by -2 to 2 do
   i^2;
end do;
```

produces \( 20^2, 18^2, 16^2, \ldots, 4^2, 2^2 \).

For-do loops work with the following conventions:

1. The loop_index is a variable name, often \( i, j \) or \( k \) (given the tradition of using these letters as variable names for integers).

2. The start_value and end_value can be numbers, variable names assigned to a numerical value, or executable expressions resulting in a number, such as 1, 10, \( k1, k2, 2 * n \), etc. The numbers they represent are usually integers, but decimal numbers or even fractions are allowed.

3. The stepsize is the value added to the loop_index at each step. If stepsize is not specified it is assumed to be 1.

4. The loop works in the following way:

   (a) The start_value is initially assigned to the loop_index.

   (b) The loop_index is compared with the end_value. If loop_index > end_value (assuming stepsize is positive - otherwise the opposite is true), the loop ends and execution continues on the line following end do; . Otherwise the block of statements to be repeated is executed with the new value carried by loop_index.

   (c) Never try to alter the loop_index inside the loop. The stepsize will be automatically and repeatedly added to loop_index until it exceeds the end_value.

   (d) The entire for-do loop will be skipped if the start_value exceeds the end_value in the direction of the stepsize. For instance, the statements inside the loop
Two typical applications of the for-do loop are iterations and recursive additions to be introduced in the next two sections.

### 2.2 Iteration

An ancient method for computing square root $\sqrt{s}$ is known as the Babylonian method or Heron’s method in the following example.

#### Example 2.1 Babylonian method for square root

Starting from any positive real number $x_0$, the iteration

$$x_i = \frac{1}{2} \left( x_{i-1} + \frac{s}{x_{i-1}} \right), \quad \text{for } i = 1, 2, \ldots$$

produces a sequence $x_1, x_2, \ldots$ that converges to $\sqrt{s}$. Write a program that, for input $s$ and $n$, output the iterate $x_n$ and the error $|x_n - \sqrt{s}|$

#### Solution:

```maple
Heron := proc( s, x0, n )
local x, k;
    x[0] := x0; # initialize x[0]
    for k from 1 to n do
        x[k] := evalf(0.5*(x[k-1] + s/x[k-1])); # iteration formula
        print(k, x[k], evalf(abs(x[k]-sqrt(s)))); # monitor the iteration
    end do;
    return x[n], evalf(abs(x[n]-sqrt(s))) # output
end proc;
```

A test run for approximating $\sqrt{2}$ by $x_6$ starting from $x_0 = 9.0$:

```maple
> s, x0, n := 2, 9, 6:
> Heron(s, x0, n);
```

```
1, 4.611111111, 3.196897549
2, 2.52243026, 1.108209464
3, 1.657655722, 0.243442160
4, 1.432089435, 0.017875873
5, 1.414325129, 0.000111567
6, 1.414213567, 5.10e-9
t, err := 6, 1.414213567, 5.10e-9
```
The program uses “print” to monitor the progress of the iteration and produces the required output items. Such a monitoring feature is useful in coding/debugging of the program, and usually should be removed when programming is finished.

Example 2.2 Fibonacci numbers. Leonardo of Pisa, known as Fibonacci, published the book *Liber abaci* in 1202 in which he posed the problem of counting the number of pairs of rabbits produced in a year by an initial pair of rabbits if every month each pair gives birth to a new pair which becomes productive from the second month\(^1\). This leads to the well-known sequence of Fibonacci numbers:

\[
F_1 = 0, \quad F_2 = 1, \quad F_k = F_{k-1} + F_{k-2}, \quad \text{for} \quad k = 3, 4, \ldots, n
\]

(2.1)

Write a program that generates the Fibonacci numbers \(F_0, F_1, \ldots, F_n\).

Solution:

```maple
Fibon := proc( n )
local F, k;
F[0], F[1] := 0, 1; # initialize the iteration
for k from 2 to n do
F[k] := F[k-1] + F[k-2]; # generating F[k]
end do;
return F; # output
end proc;

> n := 20:
> G := Fibon(n);
```

The above execution stores 21 Fibonacci numbers in the variable name \( G \), which can be accessed individually by \( G[6], G[9], G[15] \) etc, or the “seq” command

```maple
> G[6], G[7], G[8];
8, 13, 21

> seq(G[j], j=0..n);
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765
```

2.2. Newton’s iteration (Numerical Analysis)

As arguably the most widely used method for solving an equation \( f(x) = 0 \), the Newton’s iteration starts from an initial guess \( x_0 \) of a solution and defines a sequence

\[
x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad \text{for } k = 0, 1, 2, \ldots
\]

However, Newton’s iteration may or may not converge. Usually only a few steps are needed if it converges. Therefore, the iteration should be stopped if it exceeds a prescribed step count, or \( |x_{k+1} - x_k| \) is smaller than a given “error tolerance”. Such an early exit can be accomplished by using “return” statement.

Solution:

```maple
newton := proc( f, # the function
    x0, # the initial guess
    n, # step count limit
    tol # error tolerance
)
    local x, g, k;
    g := D(f); # the derivative f'(x)
    x[0] := evalf(x0); # initialize the iteration
    for k from 1 to n do # loop for newton's iteration
        # Newton's iteration formula
        x[k] := evalf( x[k-1] - f(x[k-1])/g(x[k-1]) );
        # exit early if possible
        if abs(x[k]-x[k-1]) < tol then
            return x[k];
        end if;
    end do;
    return "Newton's iteration fails"
end proc;
```

A simple polynomial function can be constructed to test the program:

\[
f(x) = (x + 1)(x + 2)(x - \pi) = x^3 + (3 - \pi)x^2 + (2 - 3\pi)x - 2\pi.
\]

with its graph
CHAPTER 2. LOOPS. PART I

With an initial guess \( x_0 = 3 \), step count limit 10 and error tolerance \( 10^{-6} \), an approximate solution is obtained:

\[
\begin{align*}
> f := x \rightarrow x^3 + (3-Pi)x^2 + (2-3*Pi)x - 2*Pi: \\
x0 := 3: \ n := 10: \ \text{tol} := 10.0^(-6): \\
s := \text{newton}(f,x0,n,tol);
\end{align*}
\]

\[
s := 3.141592653
\]

2.2.2 Chaotic map (Dynamic System)

The Henon map is a discrete-time dynamic system defined as an iteration

\[
x_{k+1} = 1 + y_k - ax_k^2, \quad y_{k+1} = bx_k
\]

where \( a \) and \( b \) are parameters that determine the behavior of the dynamic system. One of the most cited pair of values is \( a = 1.4 \) and \( b = 0.3 \). Write a program to carry out the iteration and plot the point set \( \{(x_k,y_k) \mid k = 1, \ldots, n\} \) for a large \( n \).

Solution.

\[
\begin{align*}
\text{HenonMap} & := \text{proc}( x0, y0, a, b, n ) \\
& \text{local x, y, k; } \\
& \quad x[1], y[1] := x0, y0; \quad \text{# initialize the iteration} \\
& \quad \text{for k from 1 to n-1 do} \\
& \quad \quad x[k+1] := 1+y[k]-a*x[k]^2; \quad \text{# the iteration formula} \\
& \quad \quad y[k+1] := b*x[k]; \\
& \quad \quad \text{end do;} \\
& \quad \text{return x, y; } \quad \text{# output} \\
& \text{end proc;} \\
\end{align*}
\]

\[
\begin{align*}
> \ n := 1000: \ x0, y0 := 0, 0: \quad \text{# prepare input} \\
> \ a, b := 1.4, 0.3: \\
> x, y := \text{HenonMap}(x0,y0,a,b,n): \quad \text{# execute the program} \\
> \text{ptlist := [seq([x[k], y[k]], k=1..n)]: } \quad \text{# point list} \\
> \text{plot(ptlist,style=point); } \quad \text{# plot the points}
\end{align*}
\]
2.2. Iteration

2.2.3 Continuous fractions (Number Theory)

A continuous fraction is a mathematical expression with infinite fractions nested in the denominator, such as the identity

\[ \pi = 3 + \frac{1}{9 + \frac{25}{6 + \frac{49}{6 + \frac{81}{6 + \ldots}}}}. \]  

The general form of a continuous fraction can be written as

\[ a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}}. \]  

where \( \{a_0, a_1, \ldots\} \) and \( \{b_1, b_2, \ldots\} \) are infinite sequences of integers. Equivalently, one can also consider the continuous fraction (2.3) an infinite sequence

\[ x_0 = a_0, \quad x_1 = a_0 + \frac{b_1}{a_1}, \quad x_2 = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2}}, \quad x_3 = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3}}}, \ldots. \]

Here \( x_n \) is called the \( n \)-th truncated continuous fraction of (2.3). One of the methods to compute \( x_n \) for a specified \( n > 0 \) is by an iteration where the index goes backward from \( n \) to 0:

\[ s_n = a_n, \quad s_{n-1} = a_{n-1} + \frac{b_n}{s_n}, \ldots, \quad s_1 = a_1 + \frac{b_2}{s_2}, \quad s_0 = a_0 + \frac{b_1}{s_1}, \]

and the final iterate \( s_0 \) equals to \( x_n \).

Using the identity (2.2) as an example, rewrite the identity as

\[ \pi + 3 = 6 + \frac{1^2}{6 + \frac{3^2}{6 + \frac{5^2}{6 + \ldots}}}. \]  

Namely \( 6 = a_0 = a_1 = \cdots \) and \( b_k = (2k - 1)^2 \) for \( k = 1, 2, \ldots \). We can write a program to compute an approximate value of \( \pi \) by computing the \( n \)-th fraction.

ApproximatePi := proc(n)
local s, k;
s[1] := 6; # initialize s[1]
for k from n-1 by -1 to 0 do
    s[k] := a[k] + s[k+1]/s[k];
end do;
s[0]
end proc;
CHAPTER 2. LOOPS. PART I

\[ s[k] := 6 + (2k+1)^2/s[k+1]; \quad \# \text{iteration} \]
end do;

return evalf(s[0] - 3) \quad \# \text{output } \pi
end proc;

> ApproximatePi(100);

3.1415926535897932384626433832795028841971693993751058209749445923078164062861973239416293583144

2.2.4 Continuous fractions, revisited

The symbolic computation feature of Maple provides a different way for computing the continuous fraction in (2.2). To begin, rewrite the identity (2.2) as

\[ \pi = 3 + \frac{1^2}{t_0}, \quad \text{where } t_0 = 6 + \frac{3^2}{6 + \frac{5^2}{6 + \frac{7^2}{6 + \ldots}}}. \]

Then

\[ t_0 = 6 + \frac{3^2}{t_1}, \quad \text{where } t_1 = 6 + \frac{5^2}{6 + \frac{7^2}{6 + \ldots}}. \]

Consequently, the continuous fraction is a recursion: starting with \( s_0 = 3 + \frac{1}{t_0} \), substitute \( t_0 \) with \( 6 + \frac{9}{t_1} \), substitute \( t_1 \) with \( 6 + \frac{25}{t_2} \), \ldots, etc. Such substitutions can be carried out in Maple using the command “subs”:

\[
> s[0] := 3 + 1/t[0]; \\
> s[1] := subs(t[0]=6+9/t[1], s[0]); \\
> ... 
\]

If the truncated fraction is needed, the final substitution is to substitute \( t_n \) with 6 for this example (why?). We can write a program to implement this symbolic method. In the program below, two input items are used. The first input item \( n \) is the number of fraction steps, and the second item \( c \) is a string representing “truncate” or else.

\[
\text{ContFracPi := proc(} \quad n, \quad \# \text{number of steps} \\
\quad c \quad \# \text{"symbolic" or else} \quad \text{)} \\
\text{local } s, t, k; \\
\text{s[0] := 3 + 1/t[0]; \quad \# initialize the iteration} \\
\text{for } k \text{ from 1 to } n \text{ do} \\
\text{s[k] := subs(t[k-1]=6+(2*k+1)^2/t[k], s[k-1]); \quad \# iteration} \\
\text{end do;} \\
\text{end proc;} \\
\text{ContFracPi(100, "symbolic");} \\
\text{3.1415926535897932384626433832795028841971693993751058209749445923078164062861973239416293583144} \
\]

3.141592411
2.3. RECURSIVE SUMMATION

Summation is an operation to add a sequence of numbers or functions. A “sigma” notation is a standard symbol for summations. For example

\[ \sum_{k=1}^{n} (2k - 1)^2 = 1^2 + 3^2 + 5^2 + \cdots + (2n - 1)^2 \]

is a sum of \( n \) consecutive odd numbers. This summation can be translated to a straightforward for-do loop in a Maple program:

```maple
AddOddSquares := proc(n)
    local s, k;
    s := 0;  # initialize the sum
    for k from 1 to n do
        s := s + (2*k-1)^2;  # recursive addtion
    end do;
    return s;
end proc;
```

which is an approximation to \( \pi \).
Or, alternatively, a “by” option can be used for skipping even numbers.

```plaintext
s := 0; # initialize the sum
for k from 1 by 2 to n do # recursive addition
  s := s + k^2;
end do;
```

In general, a recursive summation is carried out in Maple programming by a for-do loop consists of the following block of statements in which the variable name \( s \) is used to accumulate the sum:

```plaintext
s := 0; # initialize the sum
for [...] from [...] to [...] do # recursive addition
  s := s + [...] # recursive addition
end do;
```

### 2.3.1 Taylor series (Approximation Theory)

Infinite summations are called series, such as the Taylor series for the sine function

\[
\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots. 
\] (2.5)

With a “sigma” notation for (2.5):

\[
\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \approx \sum_{k=0}^{n-1} (-1)^k \frac{x^{2k+1}}{(2k+1)!}
\]

the implementation is straightforward:

```plaintext
sine := proc( x, n ) # independent variable of sine
  local s, k;
  s := 0; # initialize the sum
  for k from 0 to n-1 do # recursive addition
    s := s + (-1)^(k+1)*x^(2*k-1)/(2*k-1)!
  end do;
  return s;
end proc;
```

If the first input item \( x \) is provided as symbolic variable name or a fraction, the program outputs a symbolic sum:

```plaintext
> sine(t,5); # symbolic sum of 5 terms
```
2.3. RECURSIVE SUMMATION

\[
t - \frac{1}{6} t^3 + \frac{1}{120} t^5 - \frac{1}{5040} t^7 + \frac{1}{362880} t^9
\]

> sine(Pi/6,5);  # A 5-term symbolic sum approximating sin(Pi/6)

\[
\frac{1}{6} \pi - \frac{1}{1296} \pi^3 + \frac{1}{933120} \pi^5 - \frac{1}{1410877440} \pi^7 + \frac{1}{3656994324480} \pi^9
\]

A floating point number for the first input item produces a numerical result:

> sine(evalf(Pi/6),5);  # A 5-term numerical sum approximating sin(Pi/6)

0.5000000003

The result shows that a 5-term sum of the sine series (2.53) approximate \( \sin \frac{\pi}{6} = \frac{1}{2} \) to a 9-digit accuracy. A combined plot of the function \( \sin x \) and the 3-term sum of the series (2.5) shows they match well for \( x \) near zero:

> f := sine(x,3):
> plot([f,sin(x)], x=-4..4);

In this graph, the red curve is the 3-term sum of the series (2.5) approximating the sine function that is shown as the orange curve.

Remark: Professional programmers may prefer the sum in the sine program be coded like

\[
t := x;  # initialize the first term
s := t;  # initialize the sum with the 1st term
for k from 1 to n-1 do
    t := -t * x^2/((2*k)*(2*k+1));  # prepare the new term
    s := s + t;  # add a new term to the sum
end do;
\]

for efficiency in terms of operation count. The difference in performance, however, is negligible on today’s computers particularly in Maple. In fact, the more intuitive and straightforward approach of the sine program is more efficient in terms of coding, less likely to have errors, and thereby preferred throughout this book.
2.3.2 Quadrature (Numerical Analysis)

Exact values of a definite integral \( \int_a^b f(x)dx \) is not always attainable, such as \( \int_1^2 \frac{\sin x}{x}dx \). In those cases, numerical integration, or quadrature, can always be applied to compute an approximate value. The simplest quadrature is the Riemann sum that is used to derive the definition of a definite integral in Calculus. More generally, a quadrature can be carried out in two steps:

**Step 1.** Divide the interval \([a, b]\) in \(n\) subintervals \([t_0, t_1], [t_1, t_2], \ldots, [t_{n-1}, t_n]\) where \(t_0, t_1, \ldots, t_n\) are called “nodes” satisfying

\[
a = t_0 < t_1 < t_2 < \cdots < t_n = b
\]

For simplicity, those subintervals can be chosen to be of equal length:

\[
t_j = a + j \cdot h \quad \text{where} \quad h = \frac{b - a}{n}.
\]

**Step 2.** Using the property of the definite integral

\[
\int_a^b f(x)dx = \int_{t_0}^{t_1} f(x)dx + \int_{t_1}^{t_2} f(x)dx + \cdots + \int_{t_{n-1}}^{t_n} f(x)dx, \quad (2.6)
\]

apply a quadrature rule such as the trapezoidal rule

\[
\int_{\alpha}^{\beta} f(x)dx \approx \frac{\beta - \alpha}{2} [f(\alpha) + f(\beta)] \quad (2.7)
\]

on each of the integral \(\int_{t_{k-1}}^{t_k} f(x)dx\) in \((2.6)\) for \(k = 1, 2, \ldots, n\) and calculate the sum recursively.

A straightforward program for the trapezoidal sum is a translation of the \(\Sigma\) notations for \((2.6)\) along with applying \((2.7)\):

\[
\int_a^b f(x)dx = \sum_{k=1}^{n} \int_{t_{k-1}}^{t_k} f(x)dx \approx \sum_{k=1}^{n} \frac{h}{2} [f(t_{k-1}) + f(t_k)]
\]

leading to a sample program:

```plaintext
TrapezoidalSum := proc( f, # the integrand function a, # left end of the interval b, # right end of the interval n # the number of the subintervals )
local t, h, s, k;

# Step 1.
h := evalf((b-a)/n); # length of the subinterval

# Step 2.
s := 0;
for k from 1 to n do
    s := s + h/2 * (f(t) + f(t+h));
    t := t+h;
end do;
s := s + h/2 * (f(a) + f(b));

s;
end proc;
```


2.3. RECURSIVE SUMMATION

for k from 0 to n do  # compute nodes
t[k] := a + k*h;
end do;

# Step 2.
s := 0;  # initialize the sum
for k from 1 to n do  # loop for the recursive sum
    # the trapezoidal rule on the interval [t[k-1],t[k]]
s := s + 0.5*h*evalf(f(t[k-1])+f(t[k]));
end do;
return s;  # output
end proc;

A larger $n$ means finer subintervals and results in a more accurate approximation to the definite integral, as shown in the following test runs on $\int_0^\pi \sin x\,dx = 2$.

> TrapezoidalSum(sin, 0, Pi, 20)  # n = 20

1.995885971

> TrapezoidalSum(sin, 0, Pi, 200)  # n = 200

1.999958905

This sample program is the most straightforward implementation of the trapezoidal sum without optimization for efficiency. The trapezoidal sum is equal to

$$h \frac{f(t_0)}{2} + h \sum_{k=1}^{n-1} f(t_k) + \frac{h}{2} f(t_n),$$

leading to More efficient implementation.

2.3.3 Nested sums and products (Elementary Algebra)

Loops can be nested. For example, the following expression is a nested sum

$$S = [1^2 + 2^2 + \cdots + n^2] + \frac{1}{2} [1^2 + 2^2 + \cdots + (n-1)^2] +$$

$$+ \frac{1}{3} [1^2 + 2^2 + \cdots + (n-2)^2] + \cdots + \frac{1}{n-1} [1^2 + 2^2] + \frac{1}{n} 1^2$$

that can be written in Sigma notations such as

$$S = \sum_{k=0}^{n-1} \left( \frac{1}{k+1} \sum_{j=1}^{n-k} j^2 \right)$$

(2.8)

$$= \sum_{j=1}^{n} \left[ \frac{1}{j} \sum_{k=j}^{n} (k-j+1)^2 \right]$$

(2.9)
Writing Maple programs for such sums can be considered straightforward translations of the Sigma notations.

The Sigma notation \((2.8)\) can be translated to

```maple
nsigma1 := proc( n )
local s, k, t, j;
    s := 0; # initialize the k-Sigma
    for k from 0 to n-1 do # loop for the k-Sigma
        t := 0; # initialize the j-Sigma
        for j from 1 to n-k do # loop for the j-Sigma
            t := t + j^2; # accumulate a term for the j-Sigma
        end do;
        s := s + t/(k+1); # accumulate a term for the k-Sigma
    end do;
    return s
end proc;
```

A direct translation of the Sigma notation \((2.9)\) into a Maple program is given as follows.

```maple
nsigma2 := proc( n )
local s, j, t, k;
    s := 0; # initialize the j-Sigma
    for j from 1 to n do # loop for the j-Sigma
        t := 0; # initialize the k-Sigma sum
        for k from j to n do # loop for the k-Sigma
            t := t + (k-j+1)^2; # accumulate a term for the k-Sigma
        end do;
        s := s + (1/j)*t; # accumulate a term for the j-Sigma
    end do;
    return s; # output the sum
end proc;
```

For another example, the sum/product expression

\[
T = \frac{1/2}{2 + \sqrt{1}} + \frac{1/2 \cdot 2/7}{4 + \sqrt{1 + 3}} + \frac{1/2 \cdot 2/7 \cdot 3/11}{6 + \sqrt{1 + 3 + 5}} + \cdots + \frac{1/2 \cdot 2/7 \cdot 3/11 \cdots 89/90}{178 + \sqrt{1 + 3 + 5 + \cdots + 177}}
\]

can be written in a Sigma-Pi notation such as

\[
T = \sum_{k=1}^{89} \prod_{j=1}^{k} \frac{j}{j+1}.
\]

A Maple program can be written as a straightforward translation of this Sigma-Pi notation:
sigma3 := proc( )
local s, t, u, k, j, n;
s := 0; # initialize the k-Sigma
for k from 1 to 89 do # loop for the k-Sigma
    t := 1; # initialize the Pi
    for j from 1 to k do # loop for the Pi
        t := t*j/(j+1); # accumulate a factor in the Pi
    end do;
    u := 0; # initialize the n-Sigma
    for n from 1 to k do # loop for the n-Sigma
        u := u + 2*n-1; # accumulate a term in the n-Sigma
    end do;
s := s + t/(2*k+sqrt(u)); # accumulate a term in the k-Sigma
end do;
return s;
end proc;

Notice that the product in the Π loop is initialized as 1 and the sums are initialized as 0. The loops for the Π and the n-Sigma can be combined into a single loop.

2.4 Exploring scientific computing

2.4.1 Fourier series (Fourier Analysis)

Joseph Fourier (1768-1830) introduced Fourier series for solving problems in engineering, particularly in heat conduction. As now a part of Fourier Analysis, the Fourier series has a broad spectrum of applications such as signal processing, image processing, vibration analysis and electrical engineering.

In a nutshell, the Fourier series for a function \( f(x) \) on the interval \([-d, d]\) is given as

\[
\frac{a_0}{2} + \sum_{k=1}^{\infty} \left( a_k \cos \frac{k\pi x}{d} + b_k \sin \frac{k\pi x}{d} \right) \tag{2.10}
\]

where the Fourier coefficients

\[
a_k = \frac{1}{d} \int_{-d}^{d} f(x) \cos \frac{k\pi x}{d} \, dx \quad \text{and} \quad b_k = \frac{1}{d} \int_{-d}^{d} f(x) \sin \frac{k\pi x}{d} \, dx.
\]

The Fourier series converges to \( f(x) \) on \([-d, d]\) in certain sense for usual functions. A truncated sum

\[
S_n(x) = \frac{a_0}{2} + \sum_{k=1}^{n} \left( a_k \cos \frac{k\pi x}{d} + b_k \sin \frac{k\pi x}{d} \right)
\]
can be used as an approximation to the function \( f(x) \) on \([-d, d]\). Visualization of such approximations is interesting. For example, consider \( f(x) = x \) on \([-2, 2]\). A plot of \( f(x) \) and \( S_n(x) \) is as follows: 

![Plot of f(x) and S_n(x) for f(x) = x on [-2, 2].](image)

Computing truncated Fourier series is left as an exercise (Problem 27).

### 2.4.2 Solving congruences (Number Theory)

One of the fundamental problems in number theory is solving congruences

\[
f(x) \equiv b \pmod{c}
\]

for a given polynomial \( f(x) \) of integer coefficients. In simple words, solving the congruence (2.11) is to find all integers \( x \in \{0, 1, \ldots, c-1\} \) such that \( c \) divides \( f(x) \). For example, integers 3, 5, and 7 are all the solutions of the congruence

\[
x^2 \equiv 1 \pmod{8}
\]

since 8 divides \( x^2 - 1 \) for \( x = 3, 5, 8 \), while it is easy to verify that the congruence

\[
x^2 \equiv 1 \pmod{9}
\]

has no solutions.

A simple method for solving the congruence (2.11) is to test integers 0, 1, \ldots, \( c-1 \) for \( x \) one by one and collect all solutions. Writing a program to implement this method is an exercise (Problem 18).

The standard method for solving the linear congruence

\[
a x \equiv b \pmod{c}
\]

(2.12)

can be described as follows:

(a) Compute the greatest common divisor \( n = \gcd(a, c) \).
(b) If \( n \) does not divide \( b \), then there is no solution.

(c) Otherwise there are \( n \) solutions

\[ s + 0 \cdot r, s + 1 \cdot r, \ldots, s + (n - 1) \cdot r \]

where \( r = c/n \) and \( x = s \) is the unique solution to the linear congruence

\[ px \equiv q \pmod{r} \]

with \( p = \frac{a}{n} \), \( q = \frac{a}{n} \). This solution can be obtained by testing whether \( px - q \) is divisible by \( n \) for \( x = 0, 1, \ldots, n - 1 \) one by one.

For example, the linear congruence

\[ 15x \equiv 8 \pmod{6} \]

has no solution since \( 3 = \gcd(15, 6) \) and \( 3 \) does not divide \( 8 \). However, the linear congruence

\[ 15x \equiv 3 \pmod{6} \]

has three solutions

\[ 1 + 0 \cdot 2 = 1, \ 1 + 1 \cdot 2 = 3 \quad \text{and} \quad 1 + 2 \cdot 2 = 5 \]

since \( 3 = \gcd(15, 6) \) divides \( 3 \) and \( r = 6/3 = 2 \). Implementing this algorithm is also an exercise (Problem 19).

## 2.5 Exercise

1. **Catalan numbers.** Write a program that computes the Catalan numbers defined by

\[ C_0 = 1, \quad C_n = \frac{2(2n - 1)}{n + 1}C_{n-1} \]

for \( n = 1, 2, \ldots \).

2. **Padovan sequence.** Use the internet to find the definition of the Padovan sequence and write a program to compute it.

3. **The Conway series.** Use the internet to find the definition of the Conway series and write a program to compute it.
4. **A square root algorithm.** For a positive integer \( n \), an iteration is known as the Bhaskara-Brouncker algorithm for approximating \( \sqrt{n} \) by rational numbers \( \frac{a_i}{b_i} \), where \( a_i \) and \( b_i \) for \( i = 1, 2, \ldots \) are integers defined by the recurrence formula

\[
a_{i+1} = a_i + b_i \cdot n, \quad b_{i+1} = a_i + b_i \quad \text{for} \quad i = 1, 2, \ldots
\]

where \( a_1 = b_1 = 1 \). Write a program that, for input positive integers \( n \) and \( m \), carries out the iteration for \( m \) steps and outputs \( \frac{a_m}{b_m} \) and the error \( \left| \frac{a_m}{b_m} - \sqrt{n} \right| \).

5. **The \( \pi \) iterations.** The following are iterative schemes for computing digits of \( \pi \), with two of them being used to compute the first 206 billion decimal digits of \( \pi \). Write programs to implement the iterations and experiment with the \( m \) for finding approximate values of \( \pi \) (Maple Digits needs to be set properly to obtained more than 10 digits of \( \pi \)).

   (a) **Brent-Salamin Algorithm.** With \( a_0 = 1, \ b_0 = 1/\sqrt{2}, \ t_0 = 1/4, \) and \( p_0 = 1 \), iterate

   \[
a_{k+1} = \frac{a_k + b_k}{2}, \quad b_{k+1} = \sqrt{a_kb_k}, \quad t_{k+1} = t_k - (a_k - a_{k+1})^2 p_k \quad \text{and} \quad p_{k+1} = 2p_k
\]

   for \( k = 0, 1, \ldots, n - 1 \). Then \( \pi \approx (a_n + b_n)^2/(4t_n) \).

   (b) **Borwein-Borwein iteration with cubic convergence.** With \( a_0 = 1/3, \ s_0 = (\sqrt{3} - 1)/2 \), iterate

   \[
   r_{k+1} = \frac{3}{1 + 2 \sqrt{1 - s_k^2}}, \quad s_{k+1} = \frac{r_{k+1} - 1}{2}, \quad a_{k+1} = r_{k+1}^2 a_k - 3k(r_{k+1}^2 - 1)
   \]

   for \( k = 0, 1, \ldots, n - 1 \). Then \( \pi \approx 1/a_n \).

   (c) **Borwein-Borwein iteration with quartical convergence.** With \( a_0 = 6 - 4\sqrt{2} \) and \( y_0 = \sqrt{2} - 1 \), iterate

   \[
y_{k+1} = \frac{1 - 4/1 - y_k^2}{1 + 4/1 - y_k^2}, \quad a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k-3}y_{k+1}(1 + y_{k+1} + y_{k+1}^2)
   \]

   for \( k = 0, 1, \ldots, n - 1 \). Then \( \pi \approx 1/a_n \).

6. **Numerical differentiation** Let \( h_k = \frac{1}{2^k} \). Then the sequence

   \[
s_k = \frac{f(a + h_k) - f(a - h_k)}{2h_k}, \quad k = 1, 2, \ldots
\]

   converges to \( f'(a) \). Write a program that, for input \( f, a, \) and \( n \), outputs the sequence and test the performance of the program for approximating \( f'(a) \) for various functions.

---

7. **Halley’s iteration.** Halley’s method for solving \( f(x) = 0 \) is to start with an initial guess \( x_0 \) of the solution and iterate according to the formula
\[
    x_{k+1} = x_k - \frac{2f(x_k)f'(x_k)}{2[f'(x_k)]^2 - f(x_k)f''(x_k)}, \quad \text{for} \ k = 0, 1, \ldots
\]
Implement this iteration with an early exit mechanism and test its performance by constructing polynomial equations with known solutions.

8. **Laguerre’s iteration.** Laguerre’s iteration is a classic method in numerical analysis for finding polynomial roots. Let \( f(x) \) be a degree \( n \) polynomial whose roots are all real: \( z_1 \leq z_2 \leq \cdots \leq z_n \). Let \( x_0 \) be an initial iterate between two roots: \( z_k < x_0 < z_{k+1} \). Then the Laguerre’s iteration
\[
    x_{j+1}^\pm = x_j^\pm - \frac{nf(x_j^\pm)}{f'(x_j^\pm) \pm \sqrt{(n-1)^2 f'(x_j^\pm)^2 - n(n-1)f(x_j^\pm)f''(x_j^\pm)}},
\]
for \( j = 0, 1, \ldots \)
generates two sequences \( \{x_j^+\} \) and \( \{x_j^-\} \) that converge the two roots adjacent to \( x_0 \). Implement this iteration with an early exit mechanism and test its performance by constructing polynomial with known real roots.

9. **The Ikeda map.** A discrete-time dynamic system known as the Ikeda map is defined as an iteration
\[
    x_{k+1} = 1 + u(x_k \cos t_k - y_k \sin t_k)
\]
\[
    y_{k+1} = u(x_k \sin t_k + y_k \cos t_k), \quad k = 0, 1, \ldots
\]
where \( u \) is a parameter and
\[
    t_k = 0.4 - \frac{6}{1 + x_k^2 + y_k^2}
\]
Write a program that, for input \( x_0, y_0, u \) and \( n \), carries out the iteration and outputs the sequences \( \{x_0, x_1, \ldots, x_n\} \) and \( \{y_0, y_1, \ldots, y_n\} \). Test your program for \( u = 0.9 \) and \( n = 1000 \), and plot the point set \( \{(x_k, y_k) \mid k = 0, \ldots, n\} \). A sample graph is shown below.
10. **The Tinkerbell map**. A discrete dynamic system is given as an iteration:

\[
\begin{align*}
  x_{k+1} &= x_k^2 - y_k^2 + a x_k + b y_k \\
  y_{k+1} &= 2 x_k y_k + c x_k + d y_k,
\end{align*}
\]

where \( a, b, c \) and \( d \) are parameters. Write a program that, for input \( a, b, c, d, x_0, y_0 \) and \( n \), carries out the iteration and outputs the sequences \( \{x_0, x_1, \ldots, x_n\} \) and \( \{y_0, y_1, \ldots, y_n\} \). Test your program for commonly quoted values \( a = 0.9, b = -0.6013, c = 2, d = 0.5 \) and \( a = 0.3, b = 0.6, c = 2, d = 0.27 \) and plot the point set \( \{(x_k, y_k) \mid k = 0, \ldots, n\} \) (Use initial value, say \( x_0 = y_0 = 0.1 \)). A sample graph is shown below.

![Graph of the Tinkerbell map](image)

11. **Kaplan-Yorke map** A discrete dynamic system is given as an iteration: Starting from a point \((x_0, y_0)\), set \( a_0 = x_0 \) and \( b \) be a large prime number, say 350377. Then for \( k = 0, 1, \ldots, \)

\[
\begin{align*}
  a_{k+1} &= 2a_k \mod b \\
  x_{k+1} &= \frac{a_{k+1}}{b} \\
  y_{k+1} &= \alpha y_k + \cos 4\pi x_k
\end{align*}
\]

where \( \alpha \) is an adjustable parameter. Write a program that, for input \( \alpha, x_0, y_0, \) and \( n \), output the sequence of points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) and plot those points (Notice that \((x_0, y_0)\) should not be included in the graph). A sample result using input \( \alpha = 0.5, x_0 = 10000, y_0 = 0.8 \) and \( n = 3000 \):

![Graph of the Kaplan-Yorke map](image)

12. **Continuous fraction I**. Using the identity

\[
\sqrt{13} = 3 + \frac{4}{6 + \frac{4}{6 + \frac{4}{6 + \ldots}}}
\]
write a program that outputs an approximate value of $\sqrt{13}$ by computing the $n$-th truncated continuous fraction from input $n$.

13. **Continuous fraction II.** Using the identity

$$\frac{x}{\ln(1+x)} = (1 - 0 \cdot x) + \frac{1^2 x}{(2 - 1 \cdot x) + \frac{2^2 x}{(3 - 2x) + \frac{3^2 x}{\ldots}}}$$

for $|x| < 1$, write a program that outputs an approximate value of $\ln(1 + x)$ by computing the $n$-th truncated continuous fraction from input items $x$ and $n$. Sample input/output:

```latex
> s := ContFracLn(x,2,"symbolic");

\textbf{s} := \frac{x}{1 + \frac{x}{2 - x + \frac{4x}{3 - 2x + \frac{9x}{t_3}}}}

> t := ContFracLn(0.5,8,"truncate");

\textbf{r} := 0.4054205081

> r := ContFracLn(1/2,8,"truncate");

\textbf{r} := \frac{4969463}{12257280}

A sample plot for visualization of the approximation

```latex
> g := ContFracLn(x,8,"truncate");
> plot([ln(1+x),g],x=0.95..1.5,thickness=3);
```
14. **Continuous fraction III.** Using the identity

\[ e^x = 1 + \frac{x}{1 - \frac{2x}{x + 2 - \frac{3x}{x + 3 - \frac{5x}{x + 5 - \ddots}}} \]

for any real number \( x \), write a program that outputs an approximate value of \( e^x \) by computing the \( n \)-th truncated continuous fraction from input items \( x \) and \( n \).

15. **Continuous fraction IV.** Using the identity

\[ e^{2m/n} = 1 + \frac{2m}{(n-m) + \frac{m^2}{3n + \frac{m^2}{5n + \frac{m^2}{7n + \ddots}}}} \]

for integers \( m \) and \( n \), write a program that outputs an approximate value of \( e^{2m/n} \) by computing the truncated continuous fraction from input items \( m, n \).

16. **Continuous fraction V.** Using the identity

\[ \frac{4}{\pi} = 1 + \frac{1}{3 + \frac{4}{9 + \frac{16}{25 + \ddots}}} \]

write a program that outputs an approximate value of \( \pi \) by computing the \( n \) truncated continuous fraction from input \( n \).

17. **Nearest component.** Write a program, for an input vector \( x \) and number \( s \), outputs the index \( k \) and the value of \( t = x_k \) that is nearest to \( s \):

\[ |s - t| = \min_{1 \leq j \leq n} |s - x_j|. \]

18. **Solving congruence.** Write a program for solving the congruence (2.11) by testing \( x = 0, 1, \ldots, c - 1 \) one by one and collect all solutions. The program should accept input \( f, b \) and \( c \) and output Sample input/output
> f := x -> x^2;
> SolveCongruence(f,1,8);

3, 5, 7

> LinearCongruence(f,1,9);

"nosolutions"

19. **Solving linear congruence.** Implement the standard method for solving the linear congruence (2.12) described in §2.4.2. Use ?igcd in Maple to learn how to compute integer gcd.

20. **Madhava-Leibniz series.** Write the following series

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots
\]

in Sigma notation. Then write a program that, for input \( n \), outputs the approximate value of \( \pi \) using the sum of the first \( n \) terms.

21. **Madhava series of \( \pi \).** Madhava of Sangamagramma (India) discovered in late 1300 or early 1400 that

\[
\pi = \sqrt{12} \left( 1 - \frac{1}{3(3^2)} + \frac{1}{5(3^2)} - \frac{1}{7(3^2)} + \cdots \right).
\]

Write this series in Sigma notation. Then write a program to compute \( \pi \) by adding the first \( n \) terms of the Madhava series.

22. **Wallis’ infinite product for \( \pi \).** Write the following infinite product in big-\( \Pi \) notation, and then translate it into a Maple program to approximate \( \pi \) by computing the truncated infinite product

\[
\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \frac{10}{9} \cdots.
\]

23. **The cosine function** The cosine function can be calculated by the series

\[
\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!} - \cdots
\]

Write a program to approximate the cosine function by calculating the sum of the first \( n \) non-zero terms and plot a graph to visualize the approximation. How many terms are necessary for the sum approximating \( \cos\left(\frac{\pi}{3}\right) \) to have 10 accurate digits (set Digits to 12 or more)?
24. \((1+x)^p\) Write the power series in Sigma-Pi notation, and then translate it into a Maple program:

\[
(1 + x)^p = 1 + px + \frac{p(p - 1)x^2}{2!} + \frac{p(p - 1)(p - 2)x^3}{3!} + \cdots.
\]

It converges to the binomial expression \((1+x)^p\) for \(-1 < x < 1\) and any \(p\). Write a program that uses this series approximation and compute approximations for (use 10 and 20 terms): \((1.2)^5\), \((1.7)^{-3}\) and \((0.6)^{3.1}\). Compare your approximations with Maple calculations, and discuss the accuracy of the approximations as you use more terms. Search the internet and find out whose name is associated with the binomial expansion.

25. **The natural logarithm** The natural logarithm \((\ln x)\) can be approximated with the following series:

\[
\ln x = \ln a + \frac{(x - a)}{a} - \frac{(x - a)^2}{2a^2} + \frac{(x - a)^3}{3a^3} - \cdots, \quad 0 < x \leq 2a
\]

Visualize this approximation of \(\ln x\) around \(x = a\). Write a program to calculate \(\ln x\) given a specific \(a\). Try to generate each term of the sum from the previous one. The input should be \(x\) and \(a\) as well as the number of terms for the sum. The output should be the approximate value of \(\ln x\). Compare the approximation with the value calculated by Maple.

26. **The natural logarithm II**

\[
\ln x = \begin{cases} 
(x - 1) - \frac{1}{2}(x - 1)^2 + \frac{1}{3}(x - 1)^3 - \cdots & \text{if } 0 < x < 2 \\
\frac{(x - 1)}{x} + \frac{(x - 1)^2}{2x^2} + \frac{(x - 1)^3}{3x^3} + \cdots & \text{if } x \geq 2
\end{cases}
\]

Write a program implementing the two series approximations given above for a given number of terms \(n\). The input to the program should be \(x\) and \(n\). The output should give the approximation. Compare the approximation with the value calculated by Maple. Run the program with \(x = 0, 0.5, 1.0, 1.5, 2.5\).

27. **Fourier series.** Write a program to compute the truncated Fourier series

\[
f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{n} \left( a_k \cos \frac{k\pi x}{d} + b_k \sin \frac{k\pi x}{d} \right)
\]  

(2.13)

for input \(f, x, d,\) and \(n\) (cf. \(\S2.4.1\)). Where the Fourier coefficients \(a_k\) and \(b_k\) are

\[
a_k = \frac{1}{d} \int_{-d}^{d} f(x) \cos \frac{k\pi x}{d} \, dx, \quad b_k = \frac{1}{d} \int_{-d}^{d} f(x) \sin \frac{k\pi x}{d} \, dx,
\]

Use function \(f(x) = x\) on \([-2, 2]\) to visualize the approximation.
28. **Quadrature: Simpson’s rule**  The most widely used quadrature formula is the Simpson’s rule:

\[
\int_{\alpha}^{\beta} f(x) \, dx \approx \frac{\beta - \alpha}{6} \left[ f(\alpha) + 4f\left(\frac{\alpha + \beta}{2}\right) + f(\beta) \right]
\]

Write a program that, for input \( f \) and \( n \), approximates \( \int_{a}^{b} f(x) \, dx \) as the sum \( 2.6 \) and applies Simpson’s rule on all \( n \) subintervals. Test the program for \( \int_{0}^{\pi} \sin x \, dx \), \( \int_{1}^{5} \ln x \, dx \), etc.

29. **Nested sum I.**  For each of the following sums, write a program to carry out the computation.

(a) \[
\sum_{k=1}^{n} \left( \frac{k-1}{k} \left( \sum_{j=1}^{k} j^2 \right) \right)
\]

(b) \[
\sum_{k=1}^{n} \sum_{j=k}^{n} \sum_{i=1}^{j} \sin \left( \frac{i+j}{k} x \right)
\]

(c) Rewrite the following sum in sigma notation\(^3\)

\[
\frac{1}{1^2} - \frac{1}{1^2 + 2^2} + \frac{1}{1^2 + 2^2 + 3^2} - \frac{1}{1^2 + 2^2 + 3^2 + 4^2} + \ldots.
\]

Then write a program that calculates the sum of the first \( n \) terms of the series, which converges to \( 6\pi - 18 \).

30. **Nested sum II.**  The identity

\[
n^k = \sum_{\ell=1}^{k} \left( \ell^k \prod_{j=0}^{k} \frac{n-j}{\ell-j} \right)
\]

is true for all integers \( n > 0 \) and \( k > 0 \). Write a program that, for input \( n \) and \( k \), output the sum and verify that the sum equals to \( n^k \).

---

\(^3\)Evaluating this series analytically is the Problem No. 5 in the Fall 2008 Series of Purdue University Problem of the week. Readers can use internet search to find a clever analytically solution.
31. **Nested sums/products III.** Write the following sum/product expressions in Sigma-Pi notations and translate them into Maple programs for calculations.

(a) \[ \left( \frac{2}{3} \sqrt{1 - \frac{1}{2}} \right) \cdot \left( \frac{3}{4} \sqrt{1 + \frac{1}{3}} \right) \cdot \left( \frac{4}{5} \sqrt{1 - \frac{1}{4}} \right) \cdot \left( \frac{5}{6} \sqrt{1 + \frac{1}{5}} \right) \cdots \]

(b) \[ \frac{1}{2} - \frac{1 + 3}{\sqrt{2^2 + 4^2}} + \frac{1 + 3 + 5}{\sqrt{2^2 + 4^2 + 6^2}} + \cdots + \frac{1 + 3 + \cdots + 99}{\sqrt{2^2 + 4^2 + \cdots + 100^2}} \]

(c) \[ 1 + \left( \frac{1^2 + 2^2}{2^4} \right) + \left( \frac{1^2 + 2^2 + 3^2}{3^4} + \frac{3^2}{3^4} \right) + \cdots \]

(d) \[ \frac{1}{3^4} + \frac{1 \cdot 2}{4^5} + \frac{1 \cdot 2 \cdot 3}{5^6} + \cdots + \frac{1 \cdot 2 \cdot 3 \cdots 88 \cdot 89}{91^{92}} \]

(e) \[ \frac{1}{3} \left[ \ln 2 \left( 1 - \frac{1}{2} \right) \right] \cdot \left[ \frac{1}{\ln 2} \left( 1 - \frac{1}{2} - \frac{1}{3} \right) \right] \cdot \left[ \frac{1}{\ln 2} \left( 1 - \frac{1}{2} + \frac{1}{3} \right) \right] \cdots \]

32. **Bernoulli’s Identity (Calculus).** Write a program to compute approximate values of \( e^x \) using truncated versions of the Bernoulli’s Identity

\[ e^x = 1 + \sum_{k=0}^{\infty} \frac{B_k x^k}{k!} \]

where \( B_k \)’s are the Bernoulli’s numbers (Number Theory) defined by

\[ B_0 = 1, \quad B_k = -\frac{1}{k+1} \sum_{j=0}^{k-1} \frac{(k+1)!}{j!(k+1-j)!} B_j \quad \text{for} \quad k = 1, 2, \ldots. \]

### 2.6 Projects

1. **Nested radicals** A general nested radical is a mathematical expression in the form of

\[ \sqrt[n]{a_0 + b_1 \sqrt[n]{a_1 + b_2 \sqrt[n]{a_2 + b_3 \cdots}}} \]

For example, the golden ratio \( \phi = \frac{-1 + \sqrt{5}}{2} \) satisfies the identity

\[ \phi + 1 = \sqrt{1 + \sqrt{1 + \sqrt{1 + \cdots}}} \]  \hspace{1cm} (2.15)

Similar to continuous fractions, a nested radical can be computed by a recursion. Develop a top-down and a bottom-up recursions to compute the symbolic and truncated nested radicals. Use the following identities for experiments.
2.6. PROJECTS

(a) \[ 3 = \sqrt{1 + 2 \sqrt{1 + 3 \sqrt{1 + 4 \sqrt{1 + \cdots}}}. \]

(b) Ramanjan identity
\[ x + n = \sqrt{n^2 + (x + 0n)} \sqrt{n^2 + (x + 1n)} \sqrt{n^2 + (x + 2n)} \sqrt{\cdots}. \]

(c) \[ \sqrt{2} = \sqrt{\frac{2}{2^{2^0}}} + \sqrt{\frac{2}{2^{2^1}}} + \sqrt{\frac{2}{2^{2^2}}} + \sqrt{\frac{2}{2^{2^3}}} + \cdots. \]

(d) Vieta’s expression for \( \pi \)
\[ \frac{2}{\pi} = \sqrt{\frac{1}{2}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \cdots}}}}}. \]

(e) An identity by Euler
\[ \sqrt{\frac{1}{2} + \frac{x}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \frac{1}{2} \sqrt{\frac{1}{2} + \cdots}}}}}. = \begin{cases} \sqrt{\frac{1-x^2}{\arccos x}}, & |x| < 1 \\ 1, & x = 1 \\ \sqrt{\frac{x^2 - 1}{\arccos x}}, & x > 1 \end{cases} \]

2. Ascending continuous fraction There is an identity that can be used to convert an ascending continuous fraction to a descending continuous fractions:
\[ a_0 + \frac{a_1}{a_2 + \frac{a_3 + \cdots}{b_3 + \frac{b_4}{b_5 + \cdots}}} = a_0 + \frac{a_1}{b_1 - \frac{a_2 b_1}{a_1 b_2 + a_2 - \frac{a_2 a_3 b_3}{a_2 b_3 + a_3 - \frac{a_3 a_4 b_4}{a_3 b_4 + a_4 - \cdots}}}}. \]

Derive an inverse of this identity and write programs to convert the (descending) continuous fractions to their ascending counterparts and verify the identity.
3. **An Often-Summed Sum**\(^4\). For positive integers \( r, \ m, \ n \), write a program to verify the following equalities.

\[
\sum_{k \equiv r \ (\text{mod} \ m)} \binom{n}{k} = \frac{1}{m} \sum_{j=0}^{m-1} (1 + \zeta^j)^n \zeta^{-rj}
\]

\[
= \frac{1}{m} \sum_{k=0}^{n} \binom{n}{k} \sum_{j=0}^{m-1} \zeta^{(k-r)j}
\]

\[
= \frac{1}{m} \sum_{k=0}^{n} \binom{n}{k} \sum_{j=1}^{m} \cos \frac{2j(n-r-k)\pi}{m}
\]

where

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]

Chapter 3

Loops. Part II

3.1 The “while-do” loop

The “for-do” loop discussed in Chapter 2 is suitable for repeated computation when the number (or the upper bound) of the repetition is known. There are many cases where computation needs to be repeated for an unknown number of times until a certain condition eventually becomes false. The “while-do” loop is a programming feature for those situations, as shown in the following example.

Example 3.1 A fixed point iteration. It is known that the iteration

\[ x_{k+1} = \cos x_k, \quad \text{for } k = 0, 1, \ldots \]

converges to the unique “fixed point” \( x^* \) of \( \cos x \), namely \( x_* = \cos x_* \), from any starting point \( x_0 \). However, it is not known how many iteration steps is needed to achieve a desired accuracy. It may be desirable to keep the iteration running until the error estimate \( |x_k - x_{k-1}| < \delta \) where \( \delta \) is a prescribed error tolerance. In other words, the iteration continues while \( |x_k - x_{k-1}| > \delta \). The iteration can be carried out accordingly using a “while-do” loop directly in a Maple worksheet:

```maple
> x[0], x[1] := 1.0, evalf(cos(1.0)): delta := 10.0^(-8):
k := 1:
while abs(x[k]-x[k-1]) >= delta do
  k := k + 1:
  x[k] := evalf( cos(x[k-1]) ): # fixed point iteration
end do:
seq(x[j],j=1..k); # show all the iterates
```
It takes 46 iteration steps from $x_0 = 1$ to reach the approximate fixed point of prescribed accuracy.

The syntax of the while-do loop is

```
while loop_condition do
  [ block of statements to be repeated ]
end do;
```

The block of statements inside the while-do loop will be repeated as long as the `loop_condition` remain true. Before writing a while-do loop, it is important to be sure that the `loop_condition` will become false and the loop will not run forever. If an upper bound is known on the number of repetitions, or if the programmer wants to impose such an upper bound (say 100), the loop in Example 3.1 can be replaced by a “for-while-do” loop

```
for k from 1 to 100 while abs(x[k]-x[k-1]) >= delta do
  x[k+1] := evalf( cos(x[k]) );
end do;
```

or a for-do loop combined with a conditional “break” statement

```
for k from 1 to 100 do
  if abs(x[k]-x[k-1]) < delta then
    break;
  else
    x[k+1] := evalf( cos(x[k]) );
  end if
end do;
```

The `break` statement is quite useful to terminate a loop early when certain condition is met.
3.2 The golden section method (Optimization)

3.2.1 The unimodal function

The method of the golden section can be applied to find the maximum value of an unimodal function. A function is called *unimodal* if it is strictly increasing, reaches a maximum, and then strictly decreases. For example:

\[ f := x \rightarrow -x^2 + 3x - 2 \]

The graph is shown in Fig. 3.1. The maximum value of this unimodal function occurs at \( x = 1.5 \).

![Figure 3.1: A unimodal function](image)

The problem of maximizing a unimodal function arise in many practical applications with or without using computers. A scenario can easily be constructed involving such an application:

**A chicken roast problem:** A barbecue chain store plans to develop a new brand of roast chicken using their standardized oven. Those 4-pound chicken are slightly undercooked if roasted for 30 minutes but overcooked at 1 hour mark. To maximize the potential profit, the store wants to establish the optimal oven time that maximize the taste appeal, which can be quantified by assembling a focus group of typical customers to score the taste of the roast chicken from 0 to 10. It is obviously reasonable to expect this “taste function” to be a unimodal function of oven time with interval domain \([30, 60]\) in minutes.

The method of golden section is perhaps the best method to solve such problems.
3.2.2 The golden ratio

Let \([a, b]\) be an interval with a point \(c\) inside. The location of \(c\) in \([a, b]\) can be described in terms of a fraction of the length of the interval \([a, b]\). For example, one may say \(c\) is one third to the right of \(a\), meaning the length of \([a, c]\) is \(\frac{1}{3}\) of that of \([a, b]\). This “one third” is the section ratio of \(c\) in \([a, b]\).

\[
\begin{array}{c}
\cdot \quad a \quad \cdot \\
\cdot \quad c \quad \cdot \\
\cdot \quad b \quad \cdot \\
\end{array}
\]

In this case, the value of \(c\) equals that of \(a\) plus the length of \([a, c]\), which is \(\frac{1}{3}\) of the length of \([a, b]\). That is,

\[
c = a + (c - a) = a + \frac{1}{3}(b - a) = \left(1 - \frac{1}{3}\right)a + \frac{1}{3}b \tag{3.1}
\]

In general, a point \(c \in [a, b]\) with a section ratio of \(t\) (with \(0 \leq t \leq 1\)) is

\[
c = (1 - t)a + tb
\]

so the section ratio

\[
t = \frac{c - a}{b - a} = \frac{\text{length of } [a, c]}{\text{length of } [a, b]}
\]

The point symmetric to \(c\) about the midpoint of \([a, b]\) is

\[
d = ta + (1 - t)b
\]

For any section ratio \(t\), point \(c\) and \(d\) above are said to be conjugate points associated with the section ratio.

\[
\begin{array}{c}
\cdot \quad a \quad \cdot \\
\cdot \quad c = (1-t)a + tb \quad \cdot \\
\cdot \quad d = ta + (1-t)b \quad \cdot \\
\cdot \quad b \quad \cdot \\
\end{array}
\]

There is a special section ratio \(\tau\), called the golden section ratio, for which \(c\) in the interval \([a, b]\) is

\[
c = (1 - \tau)a + \tau b
\]

and its conjugate point \(d\) cuts the interval \([a, c]\) with the same ratio

\[
d = \tau a + (1 - \tau)b = (1 - \tau)a + \tau c
\]
3.2. THE GOLDEN SECTION METHOD (OPTIMIZATION)

It can be verified that the golden section ratio is

\[ \tau = \frac{-1 + \sqrt{5}}{2} \approx 0.618 \]

Rich and varied topics can be found in the literature and internet related to the golden section or the “golden mean”. with fascinating mathematics and applications.

3.2.3 Method of the golden section

Let \( f(x) \) be a unimodal function on \([a, b]\). A maximum point \( x_* \) of \( f(x) \) exists in \((a, b)\). The objective is to shrink the interval and zero in on \( x_* \).

As shown in Figure 3.2 let \( m_l \) and \( m_r \) be a pair of conjugate section points (mid-left and mid-right points respectively) corresponding to the golden section ratio. Either the left subinterval \([a, m_r]\) or the right subinterval \([m_l, b]\) contains the unique maximum point \( x_* \), depending on which function value, \( f(m_l) \) or \( f(m_r) \), is larger. If \( f(m_r) > f(m_l) \) as illustrated in Figure 3.2, then the interval \([m_l, b]\) containing \( m_r \) is the interval of choice. Similarly, if \( f(m_l) > f(m_r) \), one would shrink \([a, b]\) to \([a, m_r]\).

![Figure 3.2: Method of the golden section: interval \([m_l, b]\) is chosen over \([a, m_r]\) because \( f(m_r) > f(m_l)\)](image)

A simple rule can be summarized for choosing subintervals:

- **If the ‘left value’** \( f(m_l) \) **is greater**, choose the ‘left subinterval’ \([a, m_r]\).
- **If the ‘right value’** \( f(m_r) \) **is greater**, choose the ‘right subinterval’ \([m_l, b]\).

The beauty of the method is that \( m_l \) is one of the conjugate points of golden section in \([a, m_r]\), so is \( m_r \) in \([m_l, b]\). There is only one function evaluation in each subinterval at the other conjugate point.
Using the chicken roast problem as an example, the store R&D team would initially roast two chicken at

\[
\begin{align*}
m_r &= (1 - \tau) \cdot 30 + \tau \cdot 60 \approx 48.5 \\
m_l &= \tau \cdot 30 + (1 - \tau) \cdot 60 \approx 41.5
\end{align*}
\]

minutes. The focus group scores two chicken with, say, 7.8 and 6.2 points in taste. So the new interval containing the optimal over time is \([m_l, b] = [41.5, 60]\). To shrink this new interval, the R&D team only needs to roast one chicken at 52.9 minutes since its conjugate is 48.5 and the score 7.8 is already known! Let the taste score at 52.9 minutes be, say, 7.2. Then the optimal time is in the interval \([41.5, 52.9]\) and the next chicken is to be roasted for 45.8 minutes, scored by the focus group, and compared with the score 7.8 at the conjugate time 48.5. An approximate optimal oven time will be obtained by continuing this process.

The method of the golden section can be described in the following pseudo-code:

Input: \(a, b, f\) and error tolerance \(\delta\)

start with a working interval \([a, b]\) (more on this later)

calculate golden section points \(m_l\) and \(m_r\), along with \(v_l = f(m_l)\) and \(v_r = f(m_r)\)

While \(|b - a| > \delta\), repeat the following as a loop

If \(v_l > v_r\) then

Replace \([a, b]\) with \([a, m_r]\) (that is, set \(b = m_r\))

Set \(m_r\) and \(v_r\) to the current \(m_l\) and \(v_l\) respectively

Replace \(m_l\) with \(\tau a + (1 - \tau) b\), calculate \(v_l = f(m_l)\)

else

Replace \([a, b]\) with \([m_l, b]\)

Set \(m_l\) and \(v_l\) to the current \(m_r\) and \(v_r\) respectively

Replace \(m_r\) with \((1 - \tau) a + \tau b\), calculate \(v_r = f(m_r)\)

end if

A program based on the pseudo-code:

```plaintext
# A program to calculate the maximum point of an unimodal function
#
goldsec := proc( f, # the unimodal function
  a, # the left end point of the interval
  b, # the right end point of the interval
  delta # error tolerance )

  local tau, alpha, beta, ml, mr, vl, vr;

  if delta <= 0.0 then # avoid a negative delta
    return "error tolerance must be positive";
  end if;
```
3.3. VECTORS

tau := evalf( 0.5*(-1.0+sqrt(5.0)) ); # golden section ratio
alpha, beta := a, b; # working interval
mr := (1-tau)*alpha + tau*beta; # the mid-right point
ml := tau*alpha + (1-tau)*beta; # the mid-left point
vl, vr := f(ml), f(mr); # function values

while abs(beta-alpha) >= delta do # the main loop
    if vl > vr then
        alpha, beta := alpha, mr; # update working interval
        mr, vr := ml, vl; # update mr and vr
        ml := tau*alpha + (1-tau)*beta; # the new mid-left
        vl := f(ml); # the new vl
    else
        alpha, beta := ml, beta; # update working interval
        ml, vl := mr, vr; # update ml and vl
        mr := (1-tau)*alpha + tau*beta; # the new mid-right
        vr := f(mr) # the new vr
    end if;
end do;

return 0.5*(alpha+beta); # output
end proc;

A test execution of the program using the unimodal function in Figure 3.1:

> f := x -> -x^2 + 3*x -2:
> a, b, tol := 0, 2, 0.001:
> goldsec(f, a, b, tol);

1.500193498

Remark: Variables alpha and beta are initially set to be a and b respectively to form the necessary working interval \([\alpha, \beta]\), since a and b are input arguments that are not allowed to alter inside the program.

3.3 VECTORS

3.3.1 Generating and accessing a vector

For Maple a vector is an ordered set of Maple objects, so a vector can be used to store a sequence of data. For example, to divide the interval [0,1] into 100 subintervals of equal length 0.01, the subinterval endpoints (often called nodes) are:

\[ x_1 = 0, \ x_2 = .01, \ x_3 = .02, \ldots, \ x_{100} = .99, \ x_{101} = 1.00 \]
To generate a vector of nodes, open an (empty) vector first:

> node := Vector(101):

Then generate the nodes with a loop as follows. Notice that the index starts at 1.

> for i from 1 to 101 do
   node[i] := (i-1)*0.01
end do:

The value of \( x_i \) for each \( i \) is stored in \( \text{node}[i] \). The values can be shown by using the sequence command (seq):

0, .01, .02, .03, .04, .05, .06, .07, .08, .09, .10, .11, .12, .13, .14, .15, .16, .17, .18, .19, .20, .21, .22, .23, .24, .25, .26, .27, .28, .29, .30, .31, .32, .33, .34, .35, .36, .37, .38, .39, .40, .41, .42, .43, .44, .45, .46, .47, .48, .49, .50, .51, .52, .53, .54, .55, .56, .57, .58, .59, .60, .61, .62, .63, .64, .65, .66, .67, .68, .69, .70, .71, .72, .73, .74, .75, .76, .77, .78, .79, .80, .81, .82, .83, .84, .85, .86, .87, .88, .89, .90, .91, .92, .93, .94, .95, .96, .97, .98, .99, 1.00

Each entry of the vector is referenced by \( \text{vector\_name}[\text{index}] \) (notice the square brackets). In this example,

> node[19];

.18

> node[87];

.86

Vectors may contain numbers, strings, and other data types.

### 3.3.2 Vectors as input

When a vector (say \( x \)) is used as an input of a program, its dimension can be retrieved using the statement

\[
\text{n := LinearAlgebra[Dimension]}(\text{x});
\]

as shown in the example below. The function \text{Dimension} is in the package \text{LinearAlgebra}, which will be discussed extensively in later chapters.
Example 3.2  The following program calculates the mean value of the values stored in a vector:

```maple
mean := proc(x)
    local mn, k;
    n := LinearAlgebra[Dimension](x); # get dimension
    mn := 0; # calculate the sum
    for k from 1 to n do
        mn := mn + x[k];
    end do;
    return mn/n; # output the mean
end proc;
```

Test run:

```maple
> a := Vector([3.7,5.4,1.2,7.9,8.3]);
a :=
    [ 3.7 ]
    [ 5.4 ]
    [ 1.2 ]
    [ 7.9 ]
    [ 8.3 ]
> mean(a):
5.3
```

3.3.3  Vectors as output

For a program to generate and output a vector, there must be a statement to define a vector and subsequent statements to calculate/assign its entries, as shown in the following example.

Example 3.3 Fibonacci again. Suppose the values of the Fibonacci sequence are needed for later use. An easy way to do this is to output a vector. The following is a modification of the previous Fibonacci program.

```maple
# program to generate the first n terms of the Fibonacci sequence.
FibonacciVector := proc(n)
    local F, k;
    F := Vector[row](n); # define an empty vector
    F[1] := 0;  F[2] := 1; # the first two terms
    for k from 3 to n do # calculate/assign entries iteratively
        F[k] := F[k-1]+F[k-2];
    end do;
    return F; # output
end proc;
```
CHAPTER 3. LOOPS. PART II

Running the program:

> G := FibonacciVector(20);

By default, Maple does not show vectors of dimensions larger than 10. Functions `evalm` and `seq` can be used to view the entries of a vector:

> evalm(G);

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]

> seq(G[j],j=1..20);

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

> H := FibonacciVector(50): H[50]; # access a specific entry, say H[50]

7778742049

Besides the `Dimension` command in Example 3.2, another handy function in the package `LinearAlgebra` is `RandomVector`. It is useful for generating vectors for computing experiment. For example, to generate an integer vector of dimension 10 with random entries spread from -9 to 9:

> a := LinearAlgebra[RandomVector][row](10,generator=-9..9);

[-1 -3 -4 -6 5 -6 5 -8 -5 -1]

In contrast, to generate a real vector of the same dimension with random entries spread from -9.0 to 9.0:

> a := LinearAlgebra[RandomVector][row](10,generator=-9..9);

[-7.25162793775474412, -8.16891496863922838, -4.01538627070398046, -8.42700876520642694, 3.70829584352955862, -5.91863961939188776, 2.79860202319602002, -1.93991364838497304, 4.376384426849056, 4.639323504100141]

Obviously, the option `[row]` can be deleted if column vectors are needed.
### Example 3.4 Maximum entry of a vector.

The program for finding the value and location of the largest entry of an input vector:

```maple
findmax := proc(s) # input vector
    local i, n, value_max, index_max;
    n := LinearAlgebra[Dimension](s); # get dimension
    value_max := evalf(s[1]); # assume the first one is max
    index_max := 1;
    for i from 2 to n do # check the remaining entries
        if evalf(s[i]) > value_max then # if a bigger value is found ...
            value_max := s[i]; # update the maximum value
            index_max := i; # update the index of the max
        end if;
    end do;
    return value_max, index_max;
end proc;

> # generate a test vector
a := LinearAlgebra[RandomVector][row](10,generator=0..9);

a := [7, 3, 5, 7, 2, 8, 1, 3, 1, 4]

> vmax, imax := findmax(a);
vmax, imax := 8, 6
```

### 3.3.4 Sorting (Computer Science)

Sorting is the problem of rearranging a sequence of numbers \(x_1, x_2, \ldots, x_n\) in ascending order. The following steps constitute the most basic sorting method:

- **step 1**: find the smallest entry from \(x_1, x_2, \ldots, x_n\) and swap it to \(x_1\).
- **step 2**: find the smallest entry from \(x_2, x_3, \ldots, x_n\) and swap it to \(x_2\).
- **step 3**: find the smallest entry from \(x_3, x_4, \ldots, x_n\) and swap it to \(x_3\).

\[ \ldots \]
- **step n-1**: find the smaller entry between \(x_{n-1}\) and \(x_n\) and swap it to \(x_{n-1}\)

(Question to ponder: is there a step n?)

In summary, each loop for \(k = 1, 2, 3, \ldots, n - 1\), carries out the following step:

- **step k**: find the smallest entry from \(x_k, x_{k+1}, \ldots, x_n\) and swap it to \(x_k\).
There must be another loop nested inside to find the smallest entry in a subsequence.

```maple
sort_ascend := proc(x # vector to be sorted)
    local n, k, j, y, value_min, index_min;
    n := LinearAlgebra[Dimension](x); # get dimension
    y := x[1..n]; # the working vector
    for k from 1 to n-1 do
        # find the value_min and index_min of y[k..y[n]
        value_min := y[k]; index_min := k;
        for j from k+1 to n do
            if evalf(value_min) > evalf(y[j]) then
                value_min := evalf(y[j]); index_min := j;
            end if;
        end do;
        # swap to k-th entry if necessary
        y[index_min] := y[k]; y[k] := value_min;
    end do;
    return y;
end proc;
```

```maple
> a := Vector[row]([3,-4,9,8,-Pi,exp(1),sqrt(12),1/3,0]):
a := [3, -4, 9, 8, -\pi, e, 2 \sqrt{3}, \frac{1}{3}, 0]
> y := sort_ascend(a);
y := [-4, -\pi, 0, \frac{1}{3}, e, 3, 2 \sqrt{3}, 8, 9]
```

### 3.4 Exploring scientific computing

#### 3.4.1 The bisection method (Numerical Analysis)

The Intermediate Value Theorem covered in algebra and calculus courses states that if \( f(x) \) is a continuous function on the interval \([a, b]\) such that \( f(a) \) and \( f(b) \) have different signs, then there exists a number \( x_\ast \in [a, b] \) such that \( f(x_\ast) = 0 \). A sketch makes this theorem obvious (although there may be more than one root!).

Assuming a continuous function and two points \( a \) and \( b \) that satisfy the conditions of the theorem, the theorem can be exploited to approximate a root by cutting the interval in half once it is determined which half contains a root. The mid-point is \( mp = (a + b)/2 \) and the signs of \( f(mp) \) and \( f(a) \) can be compared to determine if they are the same or opposite (an arbitrary choice - comparing \( f(mp) \) with \( f(b) \) is equivalent), since the signs of these two numbers will determine whether we choose the interval \([a, mp]\) or the interval \([mp, b]\). The sign of \( f(mp) \times f(a) \) is positive if they have the same sign and negative otherwise, so
3.4. EXPLORING SCIENTIFIC COMPUTING

the solution $x_*$ is in or equal to \[
\begin{cases}
[a, mp] & f(a)f(mp) < 0 \\
mp & f(mp) = 0 \\
[mp, b] & \text{otherwise}
\end{cases}
\]

At each step the interval $[a, b]$ is replaced with the new subinterval, either $[a, mp]$ or $[mp, b]$, of half length. This process, called the *bisection method*, can be repeated until the length of the final interval is less than the error tolerance. At the end, the approximate solution can be given as the midpoint of the final interval. The implementation of the bisection method has much in common with the program for the method of the golden section, and is left to the reader.

How can one stop the loop, given a tolerance requirement? An option would be to calculate the number of steps: $n \geq \frac{(b-a)}{\text{tol}}$, and make sure you take at least that many steps. Another is to let Maple do it using the *while* .. *do* loop.

**Two notes:**

The expression $mp = (a + b)/2$ creates a potential roundoff problem and should be written $mp = a + \frac{b-a}{2}$. (Maple will do with this what it wishes, since it tends to simplify expressions, despite the user’s best intentions!)

The *while* loop can include a complicated logical statement such as: *while* $(b-a) > \text{tol}$ and $f(mp) <> 0$ *do* ........ (The ‘$<>’ symbol means ‘not equal to’.)

3.4.2 The greatest common divisor (*Number Theory*)

The greatest common divisor (GCD) of integers $m$ and $n$, denoted by $d = \gcd(m, n)$, is defined as the largest positive integer that divides both $m$ and $n$. For computational purpose, only the case $m > n > 0$ needs to be considered (why?). A classical method for computing GCD is the Euclidean Algorithm, which is one of the oldest algorithms in the history. The algorithm appeared in *Euclid’s Elements* around 300 BC., but it was probably discovered 200 years earlier.

The Euclidean Algorithm can be understood from the Euclid Division Lemma, which asserts that there is a unique pair of quotient $q$ and remainder $r$ such that

\[m = n \cdot q + r \quad \text{with} \quad 0 \leq r < |n|,\]

implying the GCD $d$ divides $r$ since the $d$ divides both $m$ and $n$. Furthermore, there is a decreasing sequence of remainders $r_j$ for $j = 0, 1, 2, \ldots$ satisfying

\[r_{j+1} = r_j \cdot q_j + r_{j+1} \quad \text{for} \quad j = 1, 2, \ldots \tag{3.2}\]

with $r_0 = m$ and $r_1 = n$. The GCD $d$ must divide every member of the remainder sequence. Therefore, the last nonzero remainder in the sequence must be the GCD. For
example, the Euclidean Algorithm for computing $\gcd(13578, 9198)$ generates the remainder sequence
\[
\begin{align*}
r_0 &= 13578, & r_1 &= 9198, & r_3 &= 4380, & r_4 &= 438, & r_5 &= 0
\end{align*}
\]
leading to the GCD 438.

The implementation of the Euclidean Algorithm will be an exercise. The Maple function \texttt{mod} can be conveniently used to find each remainder (c.f. \texttt{?mod} in Maple).

### 3.4.3 The greatest common divisor (Computer Algebra)

The greatest common divisor (GCD) of polynomials $f(x)$ and $g(x)$, denoted by $d = \gcd(f, g)$, is similar to the GCD of integers, defined as the monic polynomial $d(x)$ of the highest degree that divides both $f(x)$ and $g(x)$. Here a polynomial is monic if its leading coefficient is one. Computing polynomial GCD is one of the fundamental problems in Computer Algebra.

For example, the GCD of
\[
\begin{align*}
f(x) &= (x - 1)^3(x - 2)^5(x - 3) \quad \text{and} \\
g(x) &= (x - 1)^4(x - 2)^2(x + 3)^2
\end{align*}
\]
is $d(x) = (x - 1)^3(x - 2)^2$.

The Euclidean Algorithm can also be applied to compute the GCD of two polynomials. Without loss of generality again, assume the degree of $f(x)$ is no less than that of $g(x)$. Let $r_0(x) = f(x)$ and $r_1(x) = g(x)$. There is a remainder sequence $r_j(x)$ for $j = 1, 2, \ldots$ with decreasing degrees such that
\[
r_{j-1} = r_j(x) \cdot q_j(x) + r_{j+1}(x).
\]
The last nonzero remainder in the sequence is the GCD after making it monic.

Maple commands \texttt{rem}, \texttt{1coeff}, and \texttt{degree} are needed to compute remainders and leading coefficients. Syntaxes and examples of these commands can be accessed by \texttt{?rem} and \texttt{?1coeff}. The Euclidean Algorithm for finding GCDs can be illustrated using the polynomial pairs in (3.3):

```
> f := expand( (x-1)^3*(x-2)^5*(x-3) );
> g := expand( (x-1)^4*(x-2)^2*(x+3)^2 );
> degree(f,x), degree(g,x); # verify the degrees
9, 8
> r[0], r[1] := f, g;        # initialize r[0] and r[1]
> r[2] := rem( r[0], r[1], x ); # finding the remainder r[2]
```
3.4. EXPLORING SCIENTIFIC COMPUTING

3.4. EXPLORING SCIENTIFIC COMPUTING

\[ r_2 := 408 + 97x^7 - 672x^6 + 1692x^5 - 1578x^4 - 561x^3 + 2274x^2 - 1660x \]

\[ r_3 := \text{rem}(r[1], r[2], x); \]  # finding the remainder \( r[3] \)
\[ r_4 := \text{rem}(r[2], r[3], x); \]  # finding the remainder \( r[4] \)
\[ r_5 := \text{rem}(r[3], r[4], x); \]  # finding the remainder \( r[5] \)
\[ r_6 := 0 \]
\[ r_4/lcoeff(r[4], x); \]  # scale \( r[4] \) to make it monic, obtaining the GCD
\[ -4 + x^5 - 7x^4 + 19x^3 - 25x^2 + 16x \]

Implementing the Euclidean Algorithm for computing polynomial GCD will be an exercise (Problem 13).

3.4.4 Rational approximation \((\text{Number Theory})\)

As legend goes, Pythagoras held an unshakable belief that all numbers were rational and thus every number could be written as \( \frac{p}{q} \) with integers \( p \) and \( q \). One of Pythagoras’s students, Hippasus, was trying to find a rational representation for \( \sqrt{2} \) while traveling at sea but made an astonishing discovery: It is impossible to write \( \sqrt{2} \) as a fraction \( \frac{p}{q} \) using any pair of integers \( p \) and \( q \). Instead of celebrating the discovery of irrational numbers, however, Pythagoras could not accept the demise of his belief and he could not disprove Hippasus argument either. As a result, fellow Pythagoreans angrily threw Hippasus overboard and drowned him.

It is now known that it is impossible to find a rational number to equal a irrational number such as \( \sqrt{2}, \pi, e, \) etc. However, rational numbers are “dense” on the number line in the sense that there are rational numbers within any neighborhood of any irrational number. Consequently, an irrational numbers can be approximated by infinitely many rational numbers. For example, two fractions

\[ \frac{22}{7} = 3.14285714\ldots, \quad \frac{355}{113} = 3.14159292035\ldots \]

are the most famous rational approximations of \( \pi \). The question becomes how to find the most accurate rational approximation \( \frac{p}{q} \) to an irrational number, using smallest possible denominator \( q \).

Perhaps the simplest algorithm for computing rational approximations is as follows, with input \( r \) to be a real number that is to be approximated and \( N \) to be the upper bound on the denominator. Starting with \( p = \lfloor r \rfloor, \, q = 1 \) and the initial error \( \varepsilon_0 = |\frac{p}{q} - r| \), repeat the following as long as \( q \leq N \):
• If \( \frac{p}{q} < r \), increase \( p \) by 1.

• Otherwise, increase \( q \) by 1.

• With the new pair of \( p \) and \( q \), calculate the current error \( \varepsilon_1 = \left| \frac{p}{q} - r \right| \).

• If the current error \( \varepsilon_1 \) is smaller than the previous error \( \varepsilon_0 \), record the current rational approximation \( p/q \) and reset \( \varepsilon_0 := \varepsilon_1 \).

This process generates a sequence \( \{\frac{p_1}{q_1}, \frac{p_2}{q_2}, \frac{p_3}{q_3}, \ldots \} \) with increasing accuracy approximating \( r \). Implementation of this method will be an exercise. Some sample results are shown below.

Approximating \( \pi \) with denominator bound 16900:

```plaintext
> a, k := RationalApproximate(Pi, 16900):
> seq(a[j], j=1..k);
```

\[
\begin{align*}
3, & 13, 16, 19, 22, 179, 201, 223, 245, 267, 289, 311, 333, 355, 52518, 52873
\end{align*}
\]

(One can see the beauty of \( \frac{355}{113} \) from this sequence. A better approximation requires a denominator larger than 15,000.)

Approximating \( \sqrt{2} \) with denominator bound 20000:

```plaintext
> b, n := RationalApproximate(sqrt(2), 20000):
> seq(b[j], j=1..n);
```

\[
\begin{align*}
1, & 2, 3, 4, 7, 17, 24, 41, 99, 140, 239, 577, 1393, 3363, 4756, 8119, 19601, 27720
\end{align*}
\]

3.4.5 Rational approximation revisited (Number Theory)

The Euclidean Algorithm can be extended to generating a continuous fraction for approximating an irrational number \( r \). Assuming \( r > 0 \) without loss of generality, initialize \( r_0 = r, \ r_1 = 1 \), and the integer “quotient” \( q_1 = \lfloor r \rfloor \) such that

\[
r_0 = r_1 \cdot q_1 + r_2
\]

where the “remainder” \( r_1 \) satisfies \( 0 < r_2 < r_1 \). Equivalently

\[
\frac{r_0}{r_1} = q_1 + \frac{1}{r_2}.
\]
3.4. EXPLORING SCIENTIFIC COMPUTING

Using the integer “quotient” \( q_2 = \left\lfloor \frac{r_1}{r_2} \right\rfloor \) yields

\[
\frac{r_1}{r_2} = \frac{r_2 \cdot q_2 + r_3}{r_2} \quad \text{or} \quad \frac{r_1}{r_2} = \frac{q_2 + \frac{1}{r_3}}{r_2},
\]

leading to

\[
\frac{r_0}{r_1} = q_1 + \frac{\frac{1}{r_2}}{q_2 + \frac{\frac{1}{r_3}}{q_3 + \frac{\frac{1}{r_1}}{q_4 + \ddots}}}
\]

This process can be continued infinitely and produces

\[
\frac{r}{r_1} = \frac{r_0}{r_1} = q_1 + \frac{1}{q_2 + \frac{1}{q_3 + \frac{1}{q_4 + \ddots}}}
\]

Each truncation of the continuous fraction is a rational approximation to the irrational number \( r \). For instance, the Euclidean Algorithm produces the continuous fraction

\[
\pi = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \ddots}}}
\]

and rational approximations

\[
3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \ldots
\]

Computing rational approximations by the Euclidean Algorithm will be an exercise. The method for evaluating truncated continuous fractions described in §2.2.4 is recommended.

3.4.6 The sieve of Eratosthenes (Number Theory)

A classical algorithm for finding prime numbers is the sieve method devised by Eratosthenes of Alexandria in the third century BC. The method recursively strikes out non-prime numbers and, at the end, leaves only primes in the “sieve”.

Consider integers 1, 2, \ldots, \( n \). The sieve method can be described as the following process for \( k = 1, 2, 3, \ldots, \left\lfloor \sqrt{n} \right\rfloor \)

\( k = 1 \): Strike out 1 since it is neither prime nor composite.
CHAPTER 3. LOOPS. PART II

$k = 2$: Knowing $k = 2$ is in the sieve, strike out 4, 6, 8, ... since they are multiples of 2.

$k = 3$: Knowing $k = 3$ is still in the sieve, strike out its multiples 6, 9, 12, ...

$k = 4$: Since $k = 4$ has been struck out, do nothing.

... Generally at step $k$, strike out its multiples $2k$, $3k$, ... up to $n$ if $k$ is still in the sieve, otherwise do nothing.

The sieve method can be programed using a shadow vector $s$ of dimension $n$. Initially, every entry of $s$ is set to be a string "in", meaning all numbers from 1 to $n$ are initially in the sieve. If any number $k$ is to be struck out, reassign $s_k$ as "out". After finishing the sieving, collect those integer $k$'s where $s_k$ still carries the string "in" in a vector for output. A sample program and test run are as follows:

```plaintext
PrimeSieve := proc(n)
    local p, s, k, i, count;
    s := Vector(n); # initialize the shadow vector
    s[1] := "out"; # 1 is not a prime
    for k from 2 to n do
        s[k] := "in" # numbers 2, 3, ..., n are in the sieve initially
    end do;

    for k from 2 to floor(sqrt(n)) do # the loop of sieving
        if s[k]="in" then
            for i from k to n/k do
                s[i*k] := "out"; # mark multiples of k as "out"
            end do;
        end if;
    end do;

    # count the number of primes found
    count := 0;
    for i from 2 to n do
        if s[i]="in" then
            count := count+1;
        end if;
    end do;

    p := Vector[row](count); # open a vector to store prime numbers
    k := 0; # initialize counter
    for i from 2 to n do
        if s[i]="in" then # when s[i] is "in", i is the k-th prime
            k := k+1; p[k] := i; # store this prime
        end if;
    end do;

    return p[1..count];
end proc;

> p := PrimeSieve(100);```

```plaintext
[ 25 Element Row Vector
  Data type: anything
  Storage: rectangular
  Order: Fortran order ]
```
3.5 Exercise

1. **A division algorithm** (*Number Theory*). The Euclid Division Lemma in Number Theory states that for any given integers \( m \) and \( n \), there is a unique pair of integers \( q \) and \( r \) called the quotient and the remainder respectively, such that

\[
m = nq + r, \quad 0 \leq r < |n|.
\]

There is a simple algorithm for computing \( q \) and \( r \) assuming both \( m \) and \( n \) are nonnegative: For \( k = 0, 1, \ldots \), compute the decreasing sequence

\[
r_k = m - n \cdot k
\]

until \( r_k \) falls in the range \( 0 \leq r_k < n \). Then output the quotient \( q = k \) and the remainder \( r = r_k \). Use a while-do for the program.

2. **A division algorithm, revisited** (*Number Theory*). The division algorithm in Problem 1 above assumes both \( m \) and \( n \) be nonnegative. Analyze the three other cases \( \{m > 0, n < 0\}, \{m < 0, n > 0\} \) and \( \{m < 0, n < 0\} \). Then improve the algorithm and the program so that all four cases can be handled.

3. **Pythagorean triple** (*Number Theory*) An array of three positive integers \((a, b, c)\) is called a Pythagorean triple if \( a^2 + b^2 = c^2 \). Write a program that, for any starting integer pair \((a_0, b_0)\), find the next integer pair \((a_k, b_k)\) such that \( a_k^2 + b_k^2 \) is an integer square. Here

\[
(a_{j+1}, b_{j+1}) = \begin{cases} 
(a_j - 1, b_j + 1) & \text{if } a_j > 1 \\
(b_j + 1, 1) & \text{if } a_j = 1
\end{cases}
\]

for \( j = 0, 1, 2, \ldots \)

The output the discovered Pythagorean triple.

4. **Pythagorean triples** (*Number Theory*) Using the description in Problem 3 to write a program that, for input \( a_0, b_0 \) and \( n \), output \( n \) Pythagorean triples by searching the sequence of integer pairs \((a_j, b_j), j = 0, 1, 2, \ldots\).

5. **Twin primes** (*Number Theory*) A pair consecutive odd numbers \((k, k + 2)\) is call a twin prime if both are primes. For example, \((5, 7), (11, 13), (17, 19)\) are twin primes. It is a conjecture in Number Theory which states that there are infinitely many twin primes. Write a program that, for input integer \( n \), find a pair of twin primes that are larger than \( n \). Maple function “isprime” is convenient in checking prime numbers.
6. **Twin primes** (*Number Theory*) Using the description in Problem 5 to write a program that, for input \( n \) and \( m \), output \( m \) pairs of twin primes that are larger than \( n \). The output must be an \( m \)-dimensional vector with its entries being twin primes saved as 2-dimensional vectors.

7. **The 3n+1 iteration** (*Number Theory*) For an input integer \( x_0 \), the “3n+1 iteration” is described as follows:

\[
x_{k+1} = \begin{cases} 
\frac{x_k}{2}, & \text{if } x_k \text{ is even} \\
3x_k + 1, & \text{if } x_k \text{ is odd}
\end{cases}
\]

The iteration stops at \( x_k \) if it repeats one of the earlier iterates, or \( x_k = 1 \). This is because when an iterate reappears, the iteration will go into a cycle, and if \( x_k = 1 \) then the iteration will repeat the pattern 1, 4, 2, 1, 4, 2, 1 after that. When the iteration terminates at \( x[k] \), the index \( k \) is called the “length” of the iteration while the value of \( x_k \) is called the terminating value. Write a program that, for input integer \( x_0 \), outputs a sequence of the 3n+1 iterates, the length and the terminating value. Use your program to investigate the lengths and terminating values for input \( x_0 \) from 1 to, say, 100.

8. **Harmonic series** (*Calculus*). The harmonic series \( \sum_{j=1}^{\infty} \frac{1}{j} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \) is a classical example of divergent series. Namely, for any given number \( n > 0 \), there is a \( k_n > 0 \) such that

\[
\sum_{j=1}^{k_n} \frac{1}{j} \geq n
\]

Write a program that, for input integer \( n > 0 \), output the row vector \([k_1, k_2, \ldots, k_n]\). For example, when input \( n = 8 \), the output is \([1, 4, 11, 31, 83, 227, 616, 1674]\).

9. **Fibonacci m multiple** (*Number Theory*). The Fibonacci numbers \( F_0, F_1, \ldots \) are defined in (2.1) on page 36. Write a program that, for input integer \( m \), use a while-do to find the first Fibonacci number that is divisible be \( m \). For example, if the input \( m = 9 \), then the output: 12, 144, indicating that the first Fibonacci number divisible by 9 is \( F_{12} = 144 \).

10. **Extracting digits** (*Number Theory*) Write a program that, for any input integer, say 43975, outputs its digits as a vector, say \([4, 3, 9, 7, 5]\).

11. **A prime vector** (*Programming skill practice*) For an input integer \( n \), write a program that outputs the first \( n \) prime numbers in a vector. For instance, input \( n = 10 \) must lead to output \([2, 3, 5, 7, 11, 13, 17, 19, 23, 29]\).

The simplest method for finding these primes is to use the Maple function \texttt{isprime} on integers 1, 2, \ldots until \( n \) primes are identified.
12. **Longevity of a retirement fund** (*Finance*). A retiree wants to withdraw a fixed amount \( \$ x \) per month from a fund of initial balance \( \$ B \) paying \( r\% \) annual interest rate compounded monthly. He decided to write a Maple program for figuring out how long will the fund last. First of all, if the withdraw \( x \) is no more than the interest of the first month, the fund will last forever. Otherwise, the monthly balance will decrease according to

\[
\text{this month's balance} = \text{last month's balance} + \text{this month's interest} - \text{withdraw amount}.
\]

The withdraw would continue as long as the monthly balance stay positive. Write a program that, for input \( B \), \( r \) and \( x \), outputs the number of months the fund would last.

13. **Integer GCD: The Euclidean Algorithm** (*Number Theory*). Implement the Euclidean Algorithm of \( \S 3.4.2 \) by writing a program that, for input integers \( m \) and \( n \), output \( \gcd(m, n) \).

14. **Polynomial GCD: The Euclidean Algorithm** (*Computer Algebra*). Implement the Euclidean Algorithm in \( \S 3.4.3 \) by writing a program to compute the GCD of polynomials \( f(x) \) and \( g(x) \). The program should accept input \( f, g \) and \( x \) and output the (monic) GCD.

15. **Rational approximation: Simple method** (*Number Theory*). Implement the simple rational approximation method described in \( \S 3.4.4 \) that, for input irrational number \( r \) and denominator bound \( N \), outputs a sequence of rational approximations with decreasing errors.

16. **Rational approximation: The Euclidean Algorithm** (*Number Theory*). Implement the Euclidean Algorithm described in \( \S 3.4.5 \) for computing a sequence of \( n \) rational approximations to an input irrational number \( r \).

17. **A cyclic sum I** (*Programming skill practice*). Write a program that, for input vectors \( x = [x_1, \ldots, x_n] \), output a vector \( y = [y_1, \ldots, y_n] \) defined by the following pattern: For any integer \( n > 0 \), say \( n = 4 \),

\[
\begin{align*}
y_1 &= x_1 + x_2 + x_3 + x_4 \\
y_2 &= x_2 + x_3 + x_4 - x_1 \\
y_3 &= x_3 + x_4 - x_1 - x_2 \\
y_4 &= x_4 - x_1 - x_2 - x_3
\end{align*}
\]

Rewrite the definition of \( y \) using the sigma notation first.

18. **A cyclic sum II** (*Programming skill practice*). Write a program that, for input vectors \( x = [x_1, \ldots, x_n] \) and an integer \( m > 0 \), output a vector \( y = [y_1, \ldots, y_n] \)
defined by the following pattern: For any input, say \( x = [x_1, x_2, x_3, x_4] \), and \( m = 3 \),

\[
\begin{align*}
  y_1 &= x_1 + x_2 + x_3 \\
  y_2 &= x_2 + x_3 + x_4 \\
  y_3 &= x_3 + x_4 + x_1 \\
  y_4 &= x_4 + x_1 + x_2.
\end{align*}
\]

Namely, \( y_k \) is the sum of \( m \) consecutive entries of \( x \) starting from \( x_k \) as if \( x \) is arranged in a circle.

19. **Horner’s rule** (*Programming skill practice*). For a given monic polynomial \( p(x) = x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n \), its evaluation can be carried out more efficiently by using the Horner’s rule

\[
p(x) = (\cdots((x + a_1)x + a_2)x + \cdots)x + a_n.
\]

Write a program that, for input coefficient vector \( a = [a_1, \ldots, a_n] \) and \( x \), compute \( p(x) \) according to Horner’s rule. Interested readers may want to compare the efficiency between the Horner’s rule and a straightforward computation of \( p(x) \) by running each program repeatedly, say 10000 times, and measure elapsed time. Use \texttt{time} in Maple to learn how to measure elapsed time.

20. **Entry set of a vector** (*Programming skill practice*). A vector is an ordered array of entries, while a set has no order or redundancy. For example, \( x = [3, 2, 3, 1, 3, 1, 5] \) is a vector and \( \{1, 2, 3, 5\} \) is its entry set, so is \( \{3, 1, 2, 5\} \). Write a program that, for input vector \( x \), output its entry set. Use \texttt{set} to learn how to construct a set in Maple.

21. **Angle between two vectors** (*Linear Algebra*). Given two vectors \( x = [x_1, \ldots, x_n] \) and \( y = [y_1, \ldots, y_n] \), the angle \( \theta \) between \( x \) and \( y \) is defined as

\[
\theta(x, y) = \arccos \left( \frac{\sum_{j=1}^{n} x_jy_j}{\sqrt{\sum_{j=1}^{n} x_j^2} \sqrt{\sum_{j=1}^{n} y_j^2}} \right)
\]

Write a program that, for input vectors \( x \) and \( y \), output the angle \( \theta(x, y) \).

22. **Vector projection** (*Linear Algebra*). Given two vectors \( x = [x_1, \ldots, x_n] \) and \( y = [y_1, \ldots, y_n] \), the projection of \( x \) along the direction of \( y \) is a vector \( z \) defined by

\[
z = [\alpha y_1, \ldots, \alpha y_n], \quad \text{where} \quad \alpha = \frac{\sum_{j=1}^{n} x_jy_j}{\sum_{j=1}^{n} y_j^2}.
\]

Write a program that, for input vectors \( x \) and \( y \), output the projection \( z \) of \( x \) along the direction of \( y \).
23. **Area of a polygon** (Geometry). The area of a polygon with vertices \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) in either clockwise or counterclockwise order is given by

\[
\frac{1}{2} \left| (x_1y_2 - y_1x_2) + (x_2y_3 - y_2x_3) + \cdots + (x_{n-1}y_n - x_ny_{n-1}) + (x_ny_1 - y_nx_1) \right|.
\]

For example, the area of the triangle with vertices \((1, 2), (3, 4), (5, 8)\) is

\[
\frac{1}{2} \left| (1 \times 4 - 2 \times 3) + (3 \times 8 - 4 \times 5) + (5 \times 2 - 8 \times 1) \right| = \frac{4}{2} = 2
\]

Write a program that, for input vectors \(x = [x_1, x_2, \ldots, x_n]\) and \(y = [y_1, y_2, \ldots, y_n]\), output the area. Also, use Maple to plot the polygon so that you can visually verify the result (Hint: try \(>\text{plottools[polygon]}\)).

24. **Largest gap** (Statistics). Write a program that, for an input vector \(x = [x_1, \ldots, x_n]\), output the index \(i\) and the gap

\[
d_i = |x_i - x_{i+1}| = \max_{1 \leq j \leq n-1} |x_j - x_{j+1}|
\]

25. **Nearest component** (Statistics). Write a program that, for an input vector \(x = [x_1, \ldots, x_n]\) and a number \(s\), output the index \(i\) and distance \(d\) such that

\[
d = |s - x_i| = \min_{1 \leq j \leq n} |s - x_j|
\]

Here \(x_i\) is the component of \(x\) that is nearest to \(s\).

26. **Range** (Statistics). Write a program that, for an input vector \(x = [x_1, \ldots, x_n]\), output the range defined by

\[
r = \max_{1 \leq j \leq n} x_j - \min_{1 \leq j \leq n} x_j
\]

27. **Standard deviation** (Statistics). Write a program that, for an input vector \(x = [x_1, \ldots, x_n]\), calculates the standard deviation

\[
\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}
\]

where \(\mu\) is the arithmetic mean of the vector \(x_1, x_2, \ldots, x_n\), i.e.:

\[
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i
\]
28. **Pearson’s sample correlation coefficient, I (Statistics).** Let \((x_1, y_1), \ldots, (x_n, y_n)\) be a sequence of data pairs. The Pearson’s sample correlation coefficient \(r\) is defined as
\[
r = \frac{1}{(n-1)s_x s_y} \sum_{j=1}^{n} (x_j - X)(y_j - Y)
\]
where \(X\) and \(Y\) are arithmetic means of \(x = [x_1, \ldots, x_n]\) and \(y = [y_1, \ldots, y_n]\) respectively, while \(s_x\) and \(s_y\) are the standard deviations of \(x\) and \(y\) respectively. Write a program that, for input vectors \(x\) and \(y\), outputs the the Pearson’s sample correlation coefficient \(r\).

29. **Pearson’s sample correlation coefficient, II (Statistics).** Let \((x_1, y_1), \ldots, (x_n, y_n)\) be a sequence of data pairs. The Pearson’s sample correlation coefficient \(r\) can also be equivalently defined as
\[
r = \frac{\sum_{j=1}^{n} x_j y_j - \frac{1}{n} \left( \sum_{j=1}^{n} x_j \right) \left( \sum_{j=1}^{n} y_j \right)}{\sqrt{\sum_{j=1}^{n} x_j^2 - \frac{1}{n} \left( \sum_{j=1}^{n} x_j \right)^2 \cdot \sqrt{\sum_{j=1}^{n} y_j^2 - \frac{1}{n} \left( \sum_{j=1}^{n} y_j \right)^2}}}
\]
Write a program that, for input vectors \(x = [x_1, \ldots, x_n]\) and \(y = [y_1, \ldots, y_n]\), outputs the the Pearson’s sample correlation coefficient \(r\) using this formula.

30. **Sorting by index (Computer Science).** Sorting can be described in a different way: For every vector \(x = [x_1, \ldots, x_n]\), there is an ordered index list \(k_1, k_2, \ldots, k_n\) such that \(x_{k_1}, x_{k_2}, \ldots, x_{k_n}\) are in ascending order. Write a program that, for such an input vector \(x\), outputs the index list \([k_1, \ldots, k_n]\). For example, if \(x = [5.1, 3.2, 7.3, 2.4, 8.6]\) is the input vector, the output index list is \([4, 2, 1, 3, 5]\). Use **list** to learn how to construct a “list” in Maple. One way to define a list is to define a vector \(k = [k_1, \ldots, k_n]\) and then use Maple command \(k := [\text{seq}(k[j], j=1..n)]\). Sample result:

```maple
> x := Vector[row]([5.1, 3.2, 7.3, 2.4, 8.6]):  # define a vector
> k := IndexSort(x);  # execute the program
[4, 2, 1, 3, 5]
> x[k];  # use the index list k to access the sorted vector
[2.4, 3.2, 5.1, 7.3, 8.6]
```

31. **Sorting by insertion (Computer Science)** For an input vector \(x = [x_1, \ldots, x_n]\), it can be sorted into ascending order by the following insertion method: First of all, pass the input data to a working vector \(y = [y_1, \ldots, y_n]\). Consider initially \(y_1\) is sorted. Generally, assume \(y_1, \ldots, y_k\) are sorted into ascending order, then \(y_{k+1}\) will be compared with \(y_1, y_2, \ldots, y_k\) one by one and inserted into the proper place so that the new entries \(y_1, \ldots, y_k\) are in ascending order. This process is continued for \(k = 1, 2, \ldots, n-1\) and end up with a sorted vector \(y\). Write a program to implement this sorting method.
32. **Fibonacci primes** (*Number Theory*). Many Fibonacci numbers are primes that can be identified by the Maple function `isprime`. Write a program that, for input \( n \), identifies all the primes among the Fibonacci numbers \( F_0, F_1, \ldots, F_n \) and outputs them in a row vector. For example, when \( n = 20 \), the output should be \([2, 3, 5, 13, 89, 233, 1597]\).

33. **Ulam's lucky numbers** (*Number Theory*). From the list of positive integers \( 1, 2, 3, 4, \ldots, n \), remove every second number, leaving \( 1, 3, 5, 7, 9, \ldots \). The first surviving number is 3 and now remove every 3\(^{rd} \) number from the remaining numbers, yielding \( 1, 3, 7, 9, 13, 15, 19, 21, \ldots \). The next surviving number is 7 so every 7\(^{th} \) surviving number is removed, leaving \( 1, 3, 7, 9, 13, 15, 21, \ldots \). Continue this sieving process until all the surviving numbers are used. Numbers that are not removed are considered “lucky”. Write a program which outputs a vector of the lucky numbers between 1 and \( n \) using the sieve method described in §3.4.6. Sample result:

```plaintext
> u := UlamLuckyNumbers(100):
> seq(u[j], j=1..LinearAlgebra[Dimension](u));

1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69, 73, 75, 79, 87, 93, 99
```

34. **The Josephus problem** (*Number Theory*). During the Jewish rebellion against Rome (A.D. 70) 40 Jews were holding out in a cave and facing certain defeat. They would rather die than be slaves. Therefore they agreed upon a process of mutual destruction by standing in a circle and kill every seventh person until only one was left. That last one would commit suicide. The only survivor, Flavius Josephus, who quickly figured out the spot to become the last man standing, did not carry out the last step. Instead he lived to tell the story.

The generalized Josephus problem: Let \( n \) persons be arranged in a circle and numbered from 1 to \( n \). Starting from 1, remove every \( k \)-th standing person around the circle continuously until only one person (Josephus) remains standing. What is the Josephus’ number?

Write a program that, for input integer \( n > 0 \) and \( k \), output the vector containing the numbers in the order they would be removed. Josephus’ number is the last entry of the vector. Sample results

```plaintext
> p := Josephus(40,7):
> seq(p[j], j=1..40);

7, 14, 21, 28, 35, 2, 10, 18, 26, 34, 3, 12, 22, 31, 40, 11, 23, 33, 5, 17, 30, 4, 19,
36, 9, 27, 6, 25, 8, 32, 16, 1, 38, 37, 39, 15, 29, 13, 20, 24
```

The result shows that Josephus was at the number 24 spot.

---


Chapter 4

Carrying on

This chapter introduces further programming features, such as data types and subroutines. In particular, writing subroutines is essential in scientific computing because it enables us to breakdown a lengthy and complicated program into many movable “building blocks”. We can then conveniently and repeatedly access these building blocks, and assemble them into a larger computing procedure. Like previous chapters, we build on the programming techniques through exploration of scientific computing exercises and projects.

4.1 Data types

Maple command type checks data types and return true or false. For example, to check if a number is prime:

> type(5,prime);

true

> type(25,prime);

false

There are more than a hundred types of data classified in Maple. To show the complete list of data types, use the inquiry ?type. Among them, the following types are useful in this book:

<table>
<thead>
<tr>
<th>integer</th>
<th>Matrix</th>
<th>numeric</th>
<th>string</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator</td>
<td>Vector</td>
<td>prime</td>
<td>procedure</td>
<td>negative</td>
</tr>
<tr>
<td>odd</td>
<td>even</td>
<td>positive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

87
When writing a program, input data types can be, and should be specified. For example, the program \texttt{goldsec} on page 66 in §3.2.3 expects input \( f \) of type \texttt{operator} and \( a, b, \delta \) of type \texttt{numeric}. The program should start with the following lines

\begin{verbatim}
goldsec := proc( f :: operator, # the unimodal function a :: numeric, # the left end of the interval b :: numeric, # the right end of the interval delta :: numeric, # the error tolerance )
 ... 
end proc;
\end{verbatim}

When the program is executed, the data types will be checked for all input items whose types are specified with double colon ::, and an error message will be returned if any one of the input items violates the type specification. Starting from this chapter, all programs will have input data type specified.

As a good practice of codemanship, we suggest typing each input item in a separate line and align the double colons "::", as shown in the \texttt{goldsec} example above, with a short comment on each input item. This format makes the program easier to read, to understand and to debug the program.

### 4.2 Subroutines

Subroutines are common in programming languages such as FORTRAN, C, C++, etc. as building blocks for coding sophisticated software. Proper use of subroutines can substantially simplify program writing and enable teamwork by distributing those "building blocks" among team members to be constructed separately. Moreover, those subroutines can be reused as parts of other computing projects. In fact, many well known mathematical software packages such as LAPACK were released as collection of subroutines for users to incorporate them into their own programs.

Examples of writing and using subroutines are discussed in this section in case studies of scientific computing.

#### 4.2.1 Sorting revisited (Computer Science)

The sorting program \texttt{sort_ascend} on page 72 can be rewritten as one main program along with a subroutine. The objective is to sort the entries \( x_1, \ldots, x_n \) of an input vector into ascending order. The algorithm is to find the minimum number \( x_j \) in the subvector \( [x_k, x_{k+1}, \ldots, x_n] \) and swap with \( x_k \), and repeat this process for \( k = 1, 2, \ldots, n-1 \). Since finding minimum is a useful computation in its own right, we can write a separate procedure (i.e. a subroutine) and call it repeatedly by the main procedure to complete the algorithm.
4.2. SUBROUTINES

# the main program
SortAscend := proc( x :: Vector # vector to be sorted )
local n, k, y, value_min, index_min;

n := LinearAlgebra[Dimension](x); y := x[1..n]; # dimension & working vector

def for k from 1 to n-1 do
  vmin, imin := FindMin(y, k, n); # call subroutine
  y[index_min] := y[k]; y[k] := value_min; # swap entries
end do;

return y;
end proc;

# the subroutine for finding the minimum value and index
# among y[k], y[k+1], ..., y[n]
FindMin := proc( y :: Vector, # vector whose min is to be found
  k :: integer, # starting index
  n :: integer # ending index
)
local value_min, index_min, j;

index_min, index_min := k, y[k];

for j from k+1 to n do
  if evalf(value_min) > evalf(y[j]) then
    index_min, value_min := j, y[j];
  end if;
end do;

return value_min, index_min;
end proc;

In this example, the main program is substantially simplified, and the subroutine \texttt{FindMin}
can be written separately by another member of the programming team. It is recommended
that both procedures be edited in the same text file and load into Maple worksheet by a
single \texttt{read} command.

4.2.2 Using vector dot product and norm subroutines

For any vector $x = [x_1, x_2, \ldots, x_n]$ of real or complex numbers, its 2-norm $\|x\|_2$ is defined as

$$\|x\|_2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$$

which measures the length of the vector $x$. Let $x = [x_1, x_2, \ldots, x_n]$ and $y = [y_1, y_2, \ldots, y_n]$ be vectors of the same dimension $n$, the dot product of $x$ and $y$ in linear algebra is a number

$$x \cdot y = \overline{x_1}y_1 + \overline{x_2}y_2 + \cdots + \overline{x_n}y_n$$

where $\overline{c}$ for any complex number $c$ is the conjugate of $c$. Since dot product and norm evaluations are basic operations in linear algebra, we can write subroutines (as exercises) for both calculations and many linear algebra computations will need to call these two subroutines. For example, the angle between vectors $x$ and $y$ is defined as

$$\theta(x, y) = \arccos\left( \frac{\text{Re}(x \cdot y)}{\|x\|_2 \|y\|_2} \right)$$
where $\text{Re}(c)$ is the real part of the complex number $c$. Then the angle computation program can be written as follows

```maple
# Program for calculating the angle between two vectors
Angle := proc( x :: Vector,
               y :: Vector )
    local d, nx, ny;
    d := dotprod(x, y); # get x.y be calling dotprod
    nx := norm2(x);    # get the 2-norm of x by calling norm2
    ny := norm2(y);    # get the 2-norm of y by calling norm2
    return arccos(Re(d)/(nx*ny)); # output
end proc;

# subroutine for calculating the dot product
dotprod := proc( x :: Vector,
               y :: Vector )
    ... (left as exercise) ...
    return d;
end proc;

# subroutine for calculating the 2-norm
norm2 := proc( x :: Vector )
    ... ...
    return s;
end proc;
```

All three programs can be written and saved as a single file that can be read into Maple in a single "read" command.

### 4.2.3 The $3n+1$ iteration

The "$3n+1$ iteration" starts with a given initial integer iterate $x_0$ and proceeds recursively by the formula

$$x_{k+1} = \begin{cases} 
\frac{x_k}{2}, & \text{if } x_k \text{ is even} \\
3x_k + 1, & \text{if } x_k \text{ is odd}
\end{cases} \quad \text{(4.1)}$$

The iteration stops at $x_k$ if it repeats one of the earlier iterates, or $x_k = 1$. When the iteration terminates at $x[k]$, the index $k$ is called the "length" of the iteration while the value of $x_k$ is called the terminating value. We can simplify the programming in terms of strategy by breaking up the problem into three subproblems:

(i) The iteration \text{(4.1)}.

(ii) For a given sequence $x_0, x_1, \ldots, x_k$, find out if $x_k$ repeats one of its predecessors $x_0, x_1, \ldots, x_{k-1}$.

(iii) Putting things together.
4.2. SUBROUTINES

```plaintext
# the main program
it3n1 := proc( x0 :: integer )
    local x, k, repeat;
    x[0] := x0; k := 0; repeat := false; # initialization
    while x[k] <> 1 and repeat = false do # the loop
        k := k + 1;
        x[k] := NextIt(x[k-1]); # get the next iterate by calling subrouting NextIt
        repeat := CheckRepeat(x,k); # check repetition by calling subroutine CheckRepeat
    end do;
    return k, [seq(x[j],j=0..k)]; # output
end proc;

# subroutine for getting the next iterate
NextIt := proc( x :: integer # input iterate
    );
    if x mod 2 = 0 then
        return x/2; # output the next iterate for even input
    else
        return 3*x+1; # output the next iterate for odd input
    end if;
end proc;

# subroutine for checking repetition
CheckRepeat := proc( x :: symbol, # the iterates
    k :: integer # current index
    )
    local j;
    for j from 0 to k-1 do
        if x[k] = x[j] then
            return true; # found repeat, output true
        end if;
    end do;
    return false; # no repeat found at the end of the loop, output false
end proc;
```

### 4.2.4 Quadrature revisited (Numerical Analysis)

Quadrature, or numerical integration, has been introduced in §2.3.2. The following example illustrates the use of subroutines as input to a program. The objective of the computation is to compare the mid-point rule

\[
\int_{\alpha}^{\beta} f(x)dx \approx (\beta - \alpha) f \left( \frac{\alpha + \beta}{2} \right),
\]

the trapezoidal rule (2.7) on page 44 and the Simpson's rule (2.14) on page 57.

We first write a generic program CompositeQuadrature that breakdown a definite integral into a sum of definite integral over subintervals:

\[
\int_{a}^{b} f(x)dx = \int_{a}^{a+h} f(x)dx + \int_{a+h}^{a+2h} f(x)dx + \cdots + \int_{a+(n-1)h}^{a+nh} f(x)dx
\]

where \( h = \frac{b-a}{n} \) for an input \( n > 0 \). On each subinterval \([a + (k-1)h, \ a + kh] \) for \( k = 1, 2, \ldots, n \), the main routine apply an input quadrature rule. It is assumed the
quadrature rule is coded as a procedure that, for input $f$, $\alpha$ and $\beta$, outputs the approximate value of the integral $\int_{\alpha}^{\beta} f(x)\,dx$.

```plaintext
CompositeQuadrature := proc( f :: operator, # the integrand QuadRule :: procedure, # Quadrature rule a :: numeric, # lower limit of the integral b :: numeric, # upper limit of the integral n :: integer # the number of subintervals )
  local h, alpha, beta, s, k;
  h := evalf((b-a)/n); # length of subinterval
  s := 0;
  for k from 1 to n do
    alpha, beta := a+(k-1)*h, a+k*h; # prepare the subinterval
    s := s + QuadRule(f, alpha, beta); # call the input quad. rule subroutine
  end do;
  return s;
end proc;
```

In the program, the expected input argument `QuadRule` is of type `procedure`. When executing this program, this input item is to be provided by subroutines that translate the three quadrature rules:

```plaintext
MidpointRule := proc( f :: operator, # the integrand alpha :: numeric, # the lower limit beta :: numeric # the upper limit )
  return evalf( (beta-alpha)*f((alpha+beta)*0.5) );
end proc;

TrapezoidalRule := proc( f :: operator, # the integrand alpha :: numeric, # the lower limit beta :: numeric # the upper limit )
  return evalf( 0.5*(beta-alpha)*(f(alpha)+f(beta)) );
end proc;

SimpsonRule := proc( f :: operator, # the integrand alpha :: numeric, # the lower limit beta :: numeric # the upper limit )
  return evalf( ((beta-alpha)/6)*(f(alpha)+ 4*f((alpha+beta)*0.5) + f(beta)));
end proc;
```

The comparison of the three quadrature rules can be carried out as follows.

```plaintext
> f := x -> 4*x^5 - x^2; # define the integrand as an operator
> a, b := 0,2; n := 20; # specify the interval and number of subintervals
> CompositeQuadrature(f, MidpointRule, a,b,n), # using the midpoint rule
  CompositeQuadrature(f, TrapezoidalRule, a,b,n), # using the trapezoidal rule
  CompositeQuadrature(f, SimpsonRule, a,b,n); # using Simpson's rule
```
The exact integral is
\[ \int_0^2 (4x^5 - x^2)\,dx = 40 \]
and the results show the accuracy improvement from the midpoint rule to the trapezoidal rule and Simpson’s rule.

### 4.2.5 Hypotrochoid in action

### 4.2.6 Venn Diagram *(Discrete Mathematics)*

Venn diagrams are widely used in set theory, probability, computer science and statistics for illustrating logical relations of sets. For example, the set \((A \cap B) \cup C\) can be visualized by the Venn diagram in Fig. 4.1. Drawing Venn diagrams for such a simple set operation by hand is a good learning process. A sequence of set operation like
\[
[(A \cap B) \cup (A \cap C)] \cap (B \cap C)^c
\]
is somewhat more complicated than the previous one. Here the notation \((\cdot)^c\) represents the complement of the set \((\cdot)\). Instead of old-fashioned pencil-and-paper drawing, we can generate Venn diagrams by writing some generic Maple procedures, with the help of Maple plottools package. Whenever a set is given as a sequence of set operations, we only need to translate the operations into a subroutine and let Maple produce the desired Venn diagram.

The Maple procedures in this section is a case study of a program that outputs a graph, and a program consists multiple subroutines.

![Venn diagram](image)

**Figure 4.1:** A Venn diagram for the set \((A \cap B) \cup C\).

The main program *VennDiagram* below starts with constructing a base diagram consists of a 4×3 rectangle in the \(x\)-\(y\) coordinate system, three carefully placed circles representing sets
CHAPTER 4. CARRYING ON

A, B and C. Those rectangle, circles and texts A, B, C are generated by statements using `plottools[rectangle]`, `plottools[circle]`, and `plots[textplot]`. Syntaxes of those commands are accessible by Maple help inquiry “?”.

The main procedure `VennDiagram` expects the input `InSet` to be a procedure that checks whether a point in the rectangle belongs to the a set.

We set the rectangle $[0, 4] \times [0, 3]$ to contain $n = 20$ pixels per unit. Each pixel is a point $(i/n, j/n)$, for $i = 0, 1, \ldots, 4n$, $j = 0, 1, \ldots, 3n$.

The method in `VennDiagram` is to test those pixels one by one and collect those pixels in the set, and plot those points so the set is shaded.

The only input required for `VennDiagram` is a procedure that identifies if a point is in the given set and returns `true` or `false`.

```
VennDiagram := proc( InSet :: procedure # the procedure for the set checking )
local i, j, k, pt, p, b, A, B, C, At, Bt, Ct, d, pts:

# making the base diagram
b := plottools[rectangle]([0,0],[4,3]):
A := plottools[circle]([2.0,1.9],1, thickness = 4, color=red):
B := plottools[circle]([1.4,1.2],1, thickness = 4, color=green):
C := plottools[circle]([2.6,1.2],1, thickness = 4, color=blue):
At := plots[textplot]([2.0,2.5,"A"], font=[TIMES,BOLD,16], color=red):
Bt := plots[textplot]([1.0,1.0,"B"], font=[TIMES,BOLD,16], color=green):
Ct := plots[textplot]([3.0,1.0,"C"], font=[TIMES,BOLD,16], color=blue):

# generate shading points
n := 20: # 20 points per unit
k := 0: # shading point counter
for i from 0 to 4*n do
  for j from 0 to 3*n do
    pt := [i/n, j/n]: # get a pixel
    if InSet(pt) then # check if the pixel is in the set
      k := k + 1: p[k] := t: # collect a shading point
    end if:
  end do:
end do:
pts := plot([seq(p[j],j=1..k)],style=point, transparency=1): # the shading
return plots[display]({b,A,B,C,At,Bt,Ct,pts},scaling=constrained)
end proc:
```

The set $A$ is illustrated as a circle with center $(2,1.9)$ and radius 1. The subroutine `InA` identifies whether a point $p = (p_1, p_2)$ is in the set $A$ by checking if

$$(p_1 - 2.0)^2 + (p_2 - 1.9)^2 < 1$$

and return `true` or `false` accordingly. This checking can be done by a straightforward block of `if-then-else-end if` statements:

```
if (p[1]-2.0)^2+(p[2]-1.9)^2 < 1 then
  return true;
else
  return false;
end if;
```
or conveniently by the boolean evaluation function \texttt{evalb} as follows

\begin{verbatim}
   evalb( (p[1]-2.0)^2+(p[2]-1.9)^2 < 1 )
\end{verbatim}

Subroutines \texttt{InB} and \texttt{InC} are similarly coded for sets \( A \) and \( C \) respectively.

\begin{verbatim}
# procedure for checking if point p is in set A
InA := proc(p :: list)
   return evalb( (p[1]-2.0)^2+(p[2]-1.9)^2 < 1 ):
end proc:

# procedure for checking if point p is in set B
InB := proc(p :: list)
   return evalb( (p[1]-1.4)^2+(p[2]-1.2)^2 < 1 ):
end proc:

# procedure for checking if point p is in set B
InC := proc(p :: list)
   return ((p[1]-2.6)^2+(p[2]-1.2)^2 < 1 ):
end proc:
\end{verbatim}

Finally, to generate a Venn diagram for a set, say \((A \cap B) \cup C\), the user needs to write one additional subroutine

\begin{verbatim}
set1 := proc( p :: list ) # procedure for checking if point p is in the set
   return evalb( (InA(p) and InB(p)) or InC(p) )
end proc;
\end{verbatim}

as the input for \texttt{VennDiagram}. Now the Venn diagram in Fig. 4.1 can be generated by calling \texttt{VennDiagram} with \texttt{set1} as input:

\begin{verbatim}
> VennDiagram(set1);
\end{verbatim}

To generate a Venn diagram for another set, say

\begin{verbatim}
[(A \cap B) \cup (A \cap C)] \cap (B \cap C)^c
\end{verbatim}

Write a subroutine and execute \texttt{VennDiagram} accordingly:

\begin{verbatim}
set2 := proc( p :: list )
   return evalb( ((InA(p) and InB(p)) or (InA(p) and InC(p))) and
      (not (InB(p) and InC(p)) ));
end proc;

> VennDiagram( set2 );
\end{verbatim}
4.2.7 Mathematical painting

A painting artist brushes paint on canvas. For mathematicians armed with computers and a little creativity, we can make artistic painting pixel-by-pixel based on certain mathematical rules. A frequently mentioned such “drawing” is the attraction basin of Newton’s iteration, which we use as an example.

4.2.8 Prisoner’s Dilemma (Game Theory)

Game Theory is a branch of mathematics that studies how groups of people interact. Its applications arises in economics, political science, international relations, biology and philosophy. One of the classic problems in Game Theory is the famous Prisoner’s Dilemma, which can be described in a fictional story as follows.

Police arrested two suspects, Andy and Bill, for an armed robbery. The prosecutor only had evidences to convict them for minor offenses. In an attempt to solve the bigger case, the District Attorney separated Andy and Bill in different rooms and offered them the same deal:

If Andy confesses on the robbery and testifies against Bill who remains silent, then Andy gets 1 year prison time and Bill gets 10. Of course, the jail times reverse if Bill confesses and Andy refuses. If both Andy and Bill confess, then they would both get 6 years. If both remain silent, however, they would receive the same 3 year sentence for a minor offence.

The prosecutor then left Andy and Bill to make decisions in separate cells and made sure they couldn’t contact each other. What would each suspect do?

The assumption here is that both suspects want the shortest possible jail term for themselves. Namely Andy has no interest in helping Bill or other way around, and neither gives a damn on their reputation among criminals.

The readers are encouraged to make more in-depth study on the Prisoner’s Dilemma from the literature. Here we study the problem by a computer simulation to compare different strategies in choosing confession or silence based on the other prisoner’s past choices. There are infinitely many possible strategies. For instance, suppose there are five prisoners and each one sticks to one of the following strategies:

- Andy is a hardcore criminal who chooses to remain silent no matter what happens.
- Bill is a coward who always confesses.
- Charlie can’t decide so he flips a coin.
Danny wants to remain silent. However, if he knows the other guy confessed the last time, he would confess too.

Ed chooses a strategy that is the opposite of Danny’s. He would like to confess at the beginning but makes a choice that is exactly the opposite of the other prisoner most recent move, even though his strategy doesn’t make much sense. Namely, Ed would remain silent if the other prisoner confessed the last time, and confess if his partner remain silent.

We set up a scoring system: The shortest prison term (one year) worths 5 points, three year term for 3 points, six year term for 1 points, and ten year term for 0 points. These scores are summarized in the following payoff table:

<table>
<thead>
<tr>
<th>Prisoner 2:</th>
<th>confess</th>
<th>silent</th>
</tr>
</thead>
<tbody>
<tr>
<td>confess</td>
<td>(1,1)</td>
<td>(5,0)</td>
</tr>
<tr>
<td>silent</td>
<td>(0,5)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>

Table 4.1: Payoff table for the game of Prisoner’s Dilemma.

An entry, say (5,0), in the payoff table means Prisoner 1 scores 5 points for confessing while Prisoner 2 gets 0 for remaining silent. We write a Maple program that simulates those five criminals who compete with each other, say, 1000 times and see whose strategy is the best.

To carry out the simulation, we can write five subroutines representing the five strategies. Each subrouting accepts an input vector $h$ containing the history of the “opponent”. At the $k$-th round, the vector component $h_k$ (Maple syntax $h[k]$) is either the string "silent" or "confess", representing the opponent’s strategy in that round. The last component of the vector $h$, accessed by Maple vector notation $h[-1]$, contains the most recent strategy of the opponent. The main program accepts two input items: The vector of the subroutines, and the number of rounds the strategies to compete against each other. The programs can be written as follows:

```maple
PrisonerDilemma := proc( P :: Vector, # strategy subroutines
  n :: integer # games to play between two phisoners )
  local s, m, i, j, k, hi, hj, si, sj;
  m := LinearAlgebra:-Dimension(P); # number of phisoners
  s := Vector(m); # score keeper
  hi, hj := Vector(n), Vector(n); # strategy history of the players i and j.

  for i from 2 to m do
    for j from 1 to i-1 do
      # phinsoner i vs. phinsoner j for n rounds
      for k from 1 to n do
        # the k-th round game
```
CHAPTER 4. CARRYING ON

si := P[i](h[i..k-1]); # strategy of prisoner i
sj := P[j](h[i..k-1]); # strategy of prisoner j
hi[k], hj[k] := si, sj; # record history of strategies

# assign scores
if si = "silent" then # when prisoner i keeps silent
    if sj = "silent" then # prisoner j also silent
        s[i] := s[i] + 3; s[j] := s[j] + 3;
    else # prisoner j confesses
        s[i] := s[i] + 0; s[j] := s[j] + 5;
    end if;
else # when prisoner i confesses
    if sj = "silent" then # prisoner j remains silent
        s[i] := s[i] + 5; s[j] := s[j] + 0;
    else # prisoner j also confesses
        s[i] := s[i] + 1; s[j] := s[j] + 1;
    end if;
end if

end do

return s[1..-1]
end proc;

P := Vector(5): # open a vector to store strategy subroutines

# strategy of Andy: Always remain silent
P[1] := proc( h :: Vector ) # input h contains track record of the opponent
    return "silent" # always silent
end proc;

# strategy of Bill: Always confess
P[2] := proc( h :: Vector )
    return "confess" # always confess
end proc;

# strategy of Charlie: Flip a coin
P[3] := proc( h :: Vector )
    local c;
    # get a random number between 0 and 1
    c := RandomTools[Generate](integer(range=0..1))
    if c = 0 then return "confess" else return "silent" end if
end proc;

# strategy of Danny: Follow opponent's most recent history
P[4] := proc( r :: Vector )
    if LinearAlgebra:-Dimension(r) = 0 then
        return "silent" # remain silent if there is no history
    else # mimic partner's most recent strategy
        if r[-1] = "silent" then return "silent" else return "confess" end if
    end if
end proc;

# strategy of Ed: Opposite of Danny's
P[5] := proc( r :: Vector )
    if LinearAlgebra:-Dimension(r) = 0 then
        return "confess"
    else
        if r[-1] = "silent" then return "confess" else return "silent" end if
    end if
end proc;
The simulation of 1000 rounds:

```latex
> score := PrisonerDilemma(P,1000);

\[
\begin{bmatrix}
4410 \\
13984 \\
9057 \\
8482 \\
9496 \\
\end{bmatrix}
\]
```

The result shows that confess is the most rewarding strategy as the game theory predicts. It may be surprising that the strategy of Ed is the second best even though it may not appear to make much sense. A coin-flip beats half of the other strategies.

A group of readers can submit their strategy subroutines and participate in a tournament using the main program above (Exercise 6).

4.3 Exploring scientific computing

4.3.1 Julia sets (Dynamic Systems)

Discrete dynamic systems can generate enlightening computer graphics as shown in §2.2.2. The mathematical ramification goes far beyond the attractiveness of the images. This section introduces the programming of the Julia sets, which is also from the study of dynamic systems.

Consider a complex valued function \( f(z) \), say \( f(z) = 0.4e^z \), with a complex variable \( z \). Starting from a complex number \( z_0 \in \mathbb{C} \), the sequence

\[
z_{k+1} = f(z_k) \quad \text{for} \quad k = 0, 1, \ldots
\]  

is called the orbit of \( z_0 \in \mathbb{C} \). Here \( \mathbb{C} \) is field of complex numbers in the form of

\[
\mathbb{C} = \{ a + bi \mid a, b \text{ are real numbers} \}.
\]

When \( c = a + bi \in \mathbb{C} \), the real numbers \( a \) and \( b \) are the real part and imaginary part of \( c \) respectively.

Some orbits behave stably and others chaotically. The collection of the points like \( z_0 \) whose orbit behaves chaotically is called the Julia set of the function \( f \).

A straightforward algorithm\footnote{For further study, see R.L. Devaney and M.B. Durkin, The exploding exponential and other chaotic bursts in complex dynamics, The American Mathematical Monthly, Vol. 98, No. 3, pp. 217–233, 1991} can be described as follows for experimental observation of a Julia set. For a given function \( f \) with specified integer \( n > 0 \) and a real number bound \( B > 0 \), generate a point set \( S \).
1. In the \(x-y\) coordinate system, select a rectangle and form a grid.

2. At each grid point \((a_0, b_0)\) as the initial iterate, compute the orbit from \(z_0 = a_0 + b_0i\) up to \(z_n = a_n + b_ni\) using the iteration (4.2).

3. If there is an iterate \(z_j = a_j + b_ji\) whose real part \(a_i > B\), stop the iteration early and collect the starting point \((a_0, b_0)\) in the set \(S\).

The set \(S\) and its image can be used to study the Julia set of the function. The programming in this case is suitable for using a subroutine for carrying out the iteration (4.2) and returns true if the starting point is to be collected in the set \(S\) or false otherwise. The main procedure accepts input function \(f\), the grid count \(k\) per unit the maximum iteration count \(n\), the bound \(B\) on the iterative real part \(B\), and the lower left corner point \(P = (p_1, p_2)\) and the upper right corner \(Q = (q_1, q_2)\) of the selected rectangle. The output of the main program should be the list of points in the set \(S\).

For instance, let \(f(z) = 0.4e^z\), \(k = 40\), \(n = 25\), and \(B = 50.0\). The pseudo Julia set in the rectangle \([-2.5, 3] \times [-3, 3]\) is generated as follows:

```
> f := z -> 0.4*exp(z):
> P := [-2.5,-3]: Q := [3,3]: k := 40: n := 25: B := 50.0:
> pts := JuliaSet(f, P, Q, k, n, B):
> plot(pts, style=point, symbol=box, symbolsize=4);
```

The implementation of the Julia set program and its subroutine is an exercise (Problem 7).

### 4.3.2 Factoring an integer (Number Theory).

One of the cornerstones for Number Theory is the Fundamental Theorem of Arithmetic proved by Euclid. This theorem establishes the importance of prime numbers by asserting
that every positive integer \( N > 1 \) can be uniquely written as a product primes in the form of
\[
N = p_1^{e_1} \cdot p_2^{e_2} \cdots p_n^{e_n}
\]  
where
\[
p_1 < p_2 < \cdots < p_n
\]
are primes, and the integer exponents
\[
e_j > 0 \quad \text{for} \quad j = 1, \ldots, n.
\]
For example,
\[
20590675875 = 3^4 \cdot 5^3 \cdot 7^5 \cdot 11^2.
\]
Identifying the integer vectors \( p = [p_1, \ldots, p_n] \) and \( e = [e_1, \ldots, e_n] \) for a given integer \( N > 1 \) is required in many applications. This problem of integer factorization can be solved by the trivial division algorithm.

At the core of the trivial division algorithm, we need to extract the unique integers \( k \geq 0 \) and \( q \geq 1 \) from any given positive integers \( m \) and \( p \) in the factoring
\[
m = p^k q.
\]  
A simple subroutine can be written to compute \( k \) and \( q \) by repeatedly divide \( m \) by \( p \).

The trivial division algorithm for factoring a given integer \( N > 1 \) can then be described as follows: Starting from initial integers \( p = 2 \) and \( q_1 = N \), call the subroutine to find \( k_2 \) and \( q_2 \) such that
\[
N = q_1 = 2^{k_2} q_2.
\]
if \( k_2 > 0 \), then 2 is a prime factor, so \( p_1 = 2, \ e_1 = k_2 \) and we have a smaller number \( q_2 \) than \( N = q_1 \). After that, test \( p = 3 \) by calling the subroutine to find another \( k_3 \) and \( q_3 \) such that \( q_2 = 3^{k_3} q_3 \) and collect the prime factor 3 and exponent \( k_3 \) only if \( k_3 > 0 \).

The general process is to call the subrouting repeatedly for \( p = 2, 3, 4, \ldots \) to obtain \( k_p \) and \( q_p \) in the form of
\[
q_{p-1} = p^{k_p} q_p.
\]  
the factor \( p \) and exponent \( k_p \) are collected only if \( k_p > 0 \), which occurs only if \( p \) is a prime factor of \( N \) (why?).

The computation can be stopped in two ways. First of all, the process can be stopped if \( p^2 > q_{p-1} \). In this case \( q_{p-1} \) is the last prime factor of \( N \) with exponent 1. Secondly, the process can also stop after obtaining \( q_p = 1 \) by the subroutine.

Using the example of \( N = 20590675875 \) again, the iteration (4.5) produces
\[
\begin{align*}
20590675875 & = 2^9 \cdot 20590675875 \\
20590675875 & = 3^4 \cdot 254205875 \\
254205875 & = 4^3 \cdot 254205875 \\
254205875 & = 5^3 \cdot 2033647 \\
\ldots & \ldots \\
121 & = 11^2 \cdot 1
\end{align*}
\]
and stops at $q_{11} = 1$. In another example for $N = 13536$:

\[
\begin{align*}
13536 &= 2^5 \cdot 423 \\
423 &= 3^2 \cdot 47 \\
47 &= 47 \\
47 &= 7^2 \cdot 47
\end{align*}
\]

and the process stops since $7^2 > 47$, and produces the factorization $13536 = 2^5 \cdot 3^2 \cdot 47$.

The implementation of the trivial division algorithm is an exercise (Problem 8 on page 107).

### 4.3.3 Median and Quartiles (Statistics)

According to Wikipedia, there were 1,061,928 households in Chicago in the year 2000 with a median income of $38,625, which means that exactly half of the households have income above $38,625. Generally, let

\[ x_1, x_2, \ldots, x_n \]

be a sequence of real numbers sorted in ascending order. The median of those numbers is defined as follows:

**Case 1:** If $n$ is odd, then the median is the middle term $x_{\frac{n+1}{2}}$.

**Case 2:** if $n$ is even, then the median is the average $0.5 \cdot (x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$ of middle-left term and the middle-right term.

The median is also called the second quartile. The first quartile and the third quartile are defined as follows:

**Case 1:** if $n$ is even, then the sequence can be divided into the first half sub-sequence

\[ x_1, \ldots, x_{\frac{n}{2}} \]

and the second half sub-sequence

\[ x_{\frac{n}{2}+1}, \ldots, x_{n}. \]

The first quartile is the median of the first half sub-sequence and the third quartile is the median of the second half sub-sequence

**Case 2:** if $n$ is odd, then there is a middle term $x_{\frac{n+1}{2}}$. The first quartile is defined as the median of the sub-sequence

\[ x_1, \ldots, x_{\frac{n+1}{2}-1} \]

and the third quartile is defined as the median of the sub-sequence

\[ x_{\frac{n+1}{2}+1}, \ldots, x_{n}. \]
4.3. EXPLORING SCIENTIFIC COMPUTING

The objective is to compute all three quartiles. This is a typical programming problem of using subroutines. First of all, the input data vector $x = [x_1, \ldots, x_n]$ must be sorted in ascending order. Thus a sorting program is needed as a subroutine. Moreover, since each quartiles is a median of a subsequence, median computation will be repeated and should also be implemented as a subroutine.

Implementation of a quartile program is an exercise (Problem 12). The sample results are given below:

```plaintext
> x := Vector([23, 12, 34, 87, 25, 10, 5, 19, 65, 29, 71]):
> q1, q2, q3 := quartile(x);

q1, q2, q3 := 12, 25, 65

> y := Vector([22, 11, 33, 44, 51, 62, 12, 81, 37, 19, 9, 20, 18, 5]):
> q1, q2, q3 := quartile(y);

q1, q2, q3 := 12, 21.00000000, 44
```

### 4.3.4 Chinese Remainder Theorem (Number Theory)

The Chinese Remainder Theorem (CRT) is highly regarded by many as a jewel of mathematics. It first appeared in Sun Tzu’s book *Suan Ching (Calculation Classic)* in the third century A.D. and appears in many applications today such as cryptography. It is said that one of the oldest applications of the Chinese Remainder Theorem was for a general to get the exact count of his soldiers quickly.

For instance, let us consider a fictionary scenerio: After a fierce battle, the general gets his surviving soldiers and needs to know the strength of his troops precisely. By a glance of the battlefield, he knows he has no more than 20,000 soldiers. However, counting soldiers on the field takes time and can be inaccurate. Knowing the Chinese Remainder Theorem, the general orders his men to do what soldiers do best: lining up in several formations. The general only needs to know how many soldiers are left out in each formation.

When all the soldiers line up 5 in a row in whatever number of rows, there are 3 soldiers left out. Then the soldiers line up with 7 in each row with 5 left out. 9 in a row will have 6 left out; 11 in a row gets 5 left out; and finally, 13 in a row ends up with 3 left out. By counting the left out soldiers in those formations, the smart general quickly figures out he has exactly 9,168 soldiers by the Chinese Remainder Theorem.

The soldier counting problem can be formulated in number theory terms: Let the number of soldiers be $x$. Soldiers line up 5 in a row with 3 left out, implying 5 divides $(x - 3)$, namely $x \equiv 3 \pmod{5}$ (read “$x$ is congruent to 3 modulo 5”). Using this notation, the
soldier counting problem becomes solving the simultaneous congruence

\[
\begin{align*}
  x &\equiv 3 \pmod{5} \quad \text{i.e. 3 left out if 5 soliders in each row} \\
  x &\equiv 5 \pmod{7} \quad \text{i.e. 5 left out if 7 soliders in each row} \\
  x &\equiv 6 \pmod{9} \quad \text{i.e. 6 left out if 9 soliders in each row} \\
  x &\equiv 5 \pmod{11} \quad \text{i.e. 5 left out if 11 soliders in each row} \\
  x &\equiv 3 \pmod{13} \quad \text{i.e. 3 left out if 13 soliders in each row}
\end{align*}
\]

(4.6)

for an integer \( x \). In general, the Chinese Remainder Theorem solves the simultaneous congruence in the form of

\[
\begin{align*}
  x &\equiv a_1 \pmod{n_1} \\
  x &\equiv a_2 \pmod{n_2} \\
  \cdots \\
  x &\equiv a_k \pmod{n_k}
\end{align*}
\]

(4.7)

where \( n_1, \ldots, n_k \) are positive integers are "co-prime". Namely,

\[
\gcd(n_i, n_j) = 1 \quad \text{for } i \neq j
\]

(4.8)

which is satisfied automatically when \( n_1, \ldots, n_k \) are prime numbers.

The Chinese Remainder Theorem asserts that the solution of (4.7) can be obtained by the following legendary algorithm:

- **Input**: \( a = [a_1, \ldots, a_k], \ n = [n_1, \ldots, n_k], \)

- **Preparation (the general’s homework before the battle)**
  
  \begin{itemize}
  \item **Step 1.** Compute \( M = n_1 \cdot n_2 \cdots n_k. \)
  \item **Step 2.** Compute \( M_i = n_1 \cdots n_{i-1}n_{i+1} \cdots n_k \equiv M/n_i \) for \( i = 1, \ldots, k. \)
  \item **Step 3.** For \( i = 1, \ldots, k, \) compute \( b_i \) by solving the congruence
    \[
    b_i \cdot M_i \equiv 1 \pmod{n_i}
    \]
    (4.9)
    
    See §2.4.2 on page 48 on how to solve a congruence.
  \end{itemize}

- **Step 4** Set \( c_1 = b_1 M_1, \ c_2 = b_2 M_2, \ldots, \ c_k = b_k M_k \) and save these magic numbers and \( M \) on a card and take it to the battlefield.

- **Finishing touch (after soldiers line up in formations)**
  
  \begin{itemize}
  \item **Step 5** Compute the dot-product
    \[
    s = a_1 \cdot c_1 + a_2 \cdot c_2 + \cdots + a_k \cdot c_k
    \]
    (4.10)
  \end{itemize}

- **Step 6** Compute

  \[
  r = s \mod M,
  \]

  namely, divide \( s \) by \( M \) to get the remainder \( r. \)
• Output: The numbers \( r \) and \( M \), implying the solution to (4.7) is 

\[ x = r + j \cdot M, \quad \text{for } j = 0, \pm 1, \pm 2, \ldots \]

Based on this algorithm, numbers \( M \), \( M_i \)'s \( b_i \)'s are calculated as

\[
\begin{align*}
M &= 45045 \\
M_1 &= 9009, \quad M_2 = 6435, \quad M_3 = 5005, \quad M_4 = 4095, \quad M_5 = 3465 \\
b_1 &= 4, \quad b_2 = 4, \quad b_3 = 1, \quad b_4 = 4, \quad b_5 = 2.
\end{align*}
\]

from the preplanned formations 5, 7, 9, 11 and 13 regardless of the troop size. Therefore the general has a card in his pocket containing \( M = 45045 \) and five magic numbers:

\[
c_1 = 36036, \quad c_2 = 25740, \quad c_3 = 5005, \quad c_4 = 16380, \quad c_5 = 6930.
\]

corresponding to the five preplanned formations. As soon as his soldiers completed the formations and left out 3, 5, 6, 5 and 3 men respectively, the general orders his secretary to finish the last two steps of the algorithm with a simple dot-product and a division using the magic numbers

\[ s = 3 \times 36036 + 5 \times 25740 + 6 \times 5005 + 5 \times 16380 + 3 \times 6930 = 369528 \]

\[ r = 369528 \mod 45045 = 9168 \]

and knows he has 9168 soldiers precisely.

The Chinese Remainder Theorem actually states that the possible number of soldiers in this case is \( 9168 + 45045 \cdot j \), or

\[ 9168, \quad 54213, \quad 99258, \quad 144303, \quad \ldots \]

The general knows the number must be 9168 since he can surely tell the difference of troops sizes between 10000 and 50000.

Based on the Chinese Remainder Theorem, one can write a Maple program for solving the simultaneous congruence (4.7). A subroutine should be separately implemented for solving congruences at Step 3 using a method described in §2.4.2 on page 48. If the reader has written programs for either Problem 18 on page 54 or Problem 19 on page 55 those programs can be used as subroutines directly.

Furthermore, another subroutine should be written for computing \( M, \ c_1, \ldots, \ c_k \) from the “formations” \( n_1, \ldots, n_k \). The main procedure carries out the last two steps. The implementation of the algorithm is an exercise (Problem 13).

4.4 Exercise

1. Using dot product and norm (Linear Algebra) Study §4.2.2 then write a subroutine for vector dot product and a subroutine for vector 2-norm.
(i) Using the subroutines for dot product and 2-norm to write a program for projecting a vector \( x \) in the direction of \( y \), namely a vector \( z \) defined by
\[
z = \alpha y \quad \text{where} \quad \alpha = \frac{y \cdot x}{\|y\|_2^2}.
\]

(i) Using the subroutines for dot product and 2-norm to write a program for calculating the sine of the angle between \( x \) and \( y \) defined as
\[
\sin(x, y) = \left\| \frac{x}{\|x\|_2} - \beta \frac{y}{\|y\|_2} \right\|_2 \quad \text{where} \quad \beta = \frac{y \cdot x}{\|x\|_2 \|y\|_2}
\]

2. **Standard deviation** (Statistics) Write a subroutine program that, for an input vector \( x = [x_1, \ldots, x_n] \), calculate its arithmetic mean
\[
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i
\]
Using this subroutine to write a main program for calculating the the standard deviation
\[
\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}
\]

3. **Pearson’s sample correlation coefficient, I** (Statistics). Let \((x_1, y_1), \ldots, (x_n, y_n)\) be a sequence of data pairs. The Pearson’s sample correlation coefficient \( r \) is defined as
\[
r = \frac{1}{(n - 1) s_x s_y} \sum_{j=1}^{n} (x_j - X)(y_j - Y)
\]
where \( X \) and \( Y \) are arithmetic means of \( x = [x_1, \ldots, x_n] \) and \( y = [y_1, \ldots, y_n] \) respectively, while \( s_x \) and \( s_y \) are the standard deviations of \( x \) and \( y \) respectively. Using the arithmetic mean subroutine and the standard deviation subroutines to write a program that, for input vectors \( x \) and \( y \), outputs the the Pearson’s sample correlation coefficient \( r \).

4. **Pearson’s sample correlation coefficient, II** (Statistics). Let \((x_1, y_1), \ldots, (x_n, y_n)\) be a sequence of data pairs. The Pearson’s sample correlation coefficient \( r \) can also be equivalently defined as
\[
r = \frac{\sum_{j=1}^{n} x_j y_j - \frac{1}{n} \left( \sum_{j=1}^{n} x_j \right) \left( \sum_{j=1}^{n} y_j \right)}{\sqrt{\sum_{j=1}^{n} x_j^2 - \frac{1}{n} \left( \sum_{j=1}^{n} x_j \right)^2} \sqrt{\sum_{j=1}^{n} y_j^2 - \frac{1}{n} \left( \sum_{j=1}^{n} y_j \right)^2}}
\]
Using the dot product subroutine and the 2-norm subroutine in Problem [1] as well as the arithmetic mean subroutine in Problem [2] to write a program that, for input vectors \( x = [x_1, \ldots, x_n] \) and \( y = [y_1, \ldots, y_n] \), outputs the the Pearson’s sample correlation coefficient \( r \) using this formula.
5. **Venn diagrams** Generate Venn diagrams for following sets

(a) \((B \cap C) \cap (\sim A)\)

(b) \((B \cup C) \cap (\sim (A \cup (B \cap C)))\)

6. **Prisoner’s Dilemma** (*Game Theory*) Arrange a group of readers to participate a prisoner’s dilemma tournament: Each member of the group submit a strategy subroutine that returns either "silent" or "confess" from the input of the opponent’s history. Using the main program in [§4.2.8](#) to decide the winner and the ranking of all participants.

7. **The Julia set** Implement the Julia set program described in [§4.3.1](#) and recreate the sample result.

8. **Integer factoring** (*Number Theory*). Implement the trivial division algorithm in [§4.3.2](#) by writing a main procedure `TrivialDivision` that recursively calls the subroutine `ExtractFactor`. The main procedure `TrivialDivision` accepts input \(N > 1\) and outputs vectors \(p = [p_1, \ldots, p_n]\) and \(e = [e_1, \ldots, e_n]\) satisfying (4.3). The subroutine `ExtractFactor` accepts input integers \(m\) and \(p\) and outputs \(k\) and \(q\) satisfying (4.4). A sample result:

\[
> p, e := \text{TrivialDivision}(20590675875);
\]

\[
p, e := \begin{bmatrix} 3 \\ 5 \\ 11 \end{bmatrix}, \begin{bmatrix} 4 \\ 3 \\ 2 \end{bmatrix}
\]

9. **Euler Phi function** (*Number Theory*). For any integer \(N > 0\), the Euler’s Phi function \(\varphi(N)\) is defined to be the number of positive integers in \(\{1, 2, \ldots, N\}\) that are co-prime to \(N\). Clearly \(\varphi(1) = 1\). When \(N > 1\) and the prime factorization (4.3) on page 101 is available, the value of \(\varphi(N)\) can be easily computed as

\[
\varphi(N) = \prod_{j=1}^{n} (p_j^{e_j} - p_j^{e_j-1}).
\]

Write a program to compute the Euler Phi function \(\varphi(N)\) for any input positive integer \(N\) using the program `TrivialDivision` in Problem 8 as a subroutine.

10. **Integer GCD and LCM** (*Number Theory*). One of the applications of The Fundamental Theorem of Arithmetic is the representation of the GCD and LCM of two integers. Let \(a = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}\) and \(b = q_1^{f_1} q_2^{f_2} \cdots q_l^{f_l}\) be prime factorizations of integers \(a\) and \(b\). We can first rewrite the factorizations using the same primes \(r_1, \ldots, r_m\) in the forms of

\[
a = r_1^{c_1} r_2^{c_2} \cdots r_n^{c_n} \quad \text{and} \quad b = r_1^{d_1} r_2^{d_2} \cdots r_n^{d_n}.
\]
For example, we can rewrite $2^65^4$ and $3^25^87^1$ as $2^63^05^47^0$ and $2^03^25^87^1$ respectively using the same prime bases. When the factorizations in (4.11) are available, it is easy to see that

\[
\begin{align*}
gcd(a, b) &= r_1^{\min\{c_1, d_1\}} r_2^{\min\{c_2, d_2\}} \cdots r_n^{\min\{c_n, d_n\}}, \\
\text{lcm}(a, b) &= r_1^{\max\{c_1, d_1\}} r_2^{\max\{c_2, d_2\}} \cdots r_n^{\max\{c_n, d_n\}}.
\end{align*}
\]

Using the factorization program in Problem 8 as a subroutine, write a program that outputs the GCD and LCM of any given pair of integers based on the above identities.

11. **Pairwise co-prime integers** (*Number Theory*). Integers $n_1, \ldots, n_k$ are pairwise co-prime if they satisfy (4.8). Write a program that, for an input vector $n = [n_1, \ldots, n_2]$ of integers, determines if they are pairwise co-prime by calling a subroutine for computing the GCD using the Euclidean Algorithm (c.f. § 3.4.2 on page 73). The GCD subroutine is an earlier exercise (Problem 13 on page 81).

12. **Median and quartiles** (*Statistics*) Write a program that, for a input vector $x = [x_1, \ldots, x_n]$, outputs three quartiles according to the definition given in § 4.3.3. The program must use a sorting program and a median program as subroutines.

13. **The Chinese Remainder Theorem** (*Number Theory*) Implement the CRT algorithm by incorporating four Maple procedures:

   (a) A subroutine for computing the GCD by the Euclidean algorithm (c.f. § 3.4.2 on page 73 or the exercise problem 13 on page 81).

   (b) A subroutine for solving the congruence (4.9).

   (c) A subroutine that, for input vector $n = [n_1, \ldots, n_k]$ of positive integers, carries out the preparation (Step 1, Step 2 and Step 3) of the CRT algorithm and outputs $M$ along with the vector $c = [c_1, \ldots, c_k]$.

   (d) A main procedure that, for input vector $a = [a_1, \ldots, a_l]$ and $n = [n_1, \ldots, n_k]$, verify (4.8) by the GCD subroutine and solves the simultaneous congruence (1.7) based on the algorithm in § 4.3.4.

Use the soldier counting problem to test your program.

4.5 **Projects**

4.5.1 **The Game of Chicken** (*Game Theory*)

Similar to Prisoner’s Dilemma, the Game of Chicken is also a classic problem in game theory. The term “game of chicken” appears frequently in discussions of international conflicts. A
4.5. PROJECTS

typical example is the Cuban missile crisis in 1961 when the United States and the Soviet Union were on a collision course and President Kennedy dared the Soviet leader Khrushchev on a potential nuclear war.

The game can be described as two teenagers driving toward each other on a two-way street with left wheels on the middle line. Whoever “chickened out” by swerving right is the loser. The price of bravery may be quite high and the hero(s) may not live to brag about it if both refuse to be chicken, or at least the cars would be totaled. For either player in the game, the best case scenario is to go straight with the opponent swerves, the second best would be both chickens out, followed by getting out of the way and letting the opponent to be hero, with the worst outcome being a collision which both players must avoid. Clearly, the key for a player to winning the game of chicken is to convince the opponent that he is reckless by showing a history of craziness.

We can experiment with the Game of Chicken by comparing different strategies, similar to the simulation of the game of Prisoner’s Dilemma discussed in 

- Create a scoring system according to the preference of the outcome.
- Create several strategies as subroutines that accept opponents history of actions as input and output a choice between “swerve” and “don’t swerve”.
- The main procedure set up a tournament so that each strategic subroutine plays against every other subroutine \( n \) times and output the score of each strategy.
- In a class, each student submits a strategic subroutine and participates in the tournament of the game of chicken and the main procedure decides the winner and ranking.

Readers can find many discussions on the Game of Chicken by searching the internet and by looking up the literature, such as the book *Mathematics and Politics* by Taylor and Pacelli.
Chapter 5

Solving equations

5.1 Solving equations for conventional solutions

5.1.1 Maple equation solvers

Using Maple, we can solve equations in a convenient and intuitive way that is exactly how we formulate the equations mathematically. For example, to solve the equation $5x + \pi = 9$ for the unknown $x$:

> solve( 5*x+Pi=9, x);

$$-\frac{1}{5}\pi + \frac{9}{5}$$

To solve the system of equations

$$\begin{cases} 3x + 6y = 5 \\ 4x - 5y = 3 \end{cases}$$

for two unknowns $x$ and $y$:

> solve( {3*x+6*y=5,4*x-5*y=3}, {x,y} );

$$\begin{cases} y = \frac{11}{39} \\ x = \frac{43}{39} \end{cases}$$

> assign(%);

The statement `assign(%)` immediately following the execution of `solve(...)` assigns the solution values to the variable names $x$ and $y$ so that we can use them for further computations. Without the `assign(%)` statement, the solutions shown in (5.2) are displayed but $x$ and $y$ do not carry the values as shown.
As one of the leading computer algebra systems for symbolic computation, Maple is built for solving the so-called “literal equations”:

\[
\begin{align*}
&x, y := 'x','y': \quad \text{# restore } x \text{ and } y \text{ as unassigned names.} \\
&\text{solve( } \{a*x+b*y=c, d*x+e*y=f\}, \{x,y\} ); \quad \text{# solve the literal system of equations} \\
&\begin{cases}
  x = \frac{-bf-ce}{ae-db}, \\
  y = \frac{af-dc}{ae-db}
\end{cases}
\end{align*}
\]

(5.3)

**Example:** (Calculus) A particle is moving in space with its location at

\[
\begin{align*}
x &= t \cos t, \\
y &= t \sin t, \\
z &= t
\end{align*}
\]

after \( t \) seconds. How long does it take for the particle to travel the first 5 meters?

This is an arclength problem in Calculus that can be expressed as solving the equation

\[
\int_0^\tau \sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2 + (\frac{dz}{dt})^2} \, dt = 5
\]

for the unknown \( \tau \). We can apply the `solve` command with a straightforward translation of the equation (5.4):

\[
\begin{align*}
&> x, y, z := t*\cos(t), t*\sin(t), t; \\
&> \text{solve(int(sqrt(diff(x,t)^2+diff(y,t)^2+diff(z,t)^2), t=0..tau)=5, tau);} \\
&\text{RootOf} \left( Z^4 + 2Z^2 - 4 \arcsinh \left( \frac{1}{2} \sqrt{2} Z \right) \right)^2 + 40 \arcsinh \left( \frac{1}{2} \sqrt{2} Z \right) - 100 \right) \\
&> \text{evalf(%); \# obtain solution in floating point number} \\
&2.525998459
\end{align*}
\]

The `solve` command produced a *symbolic* solution (5.5), which is the root of the transcendental function in the variable “\( Z \)”. Maple can not find a closed-form solution to the equation. Instead, Maple simplifies the equation (5.4) as much as it can and reports the solution as the root of the reduced function. If an approximate solution is desired, the symbolic solution (5.5) can be converted to a numerical solution using the `evalf` function as shown above. Alternatively, one can use the `fsolve` command that would be equivalent to the combination of the two statements in this case:

\[
\begin{align*}
&> s := \text{fsolve(int(sqrt(diff(x,t)^2+diff(y,t)^2+diff(z,t)^2), t=0..tau)=5, tau);} \\
&2.525998459 \\
&> \text{int(sqrt(diff(x,t)^2+diff(y,t)^2+diff(z,t)^2), t=0..s); \# verify the solution}
\end{align*}
\]
It should be understood that there is no universal method for solving nonlinear equations in general. As a result, Maple may or may not be able to find a solution. Even if one or more solutions are found, there could be some missed solutions. It is the user responsibility to decide if the Maple output is acceptable.

A fundamental equation in scientific computing is the system of linear equations in the form of

$$Ax = b$$  \hspace{1cm} (5.6)

where $A$ is a given matrix, $b$ is a given vector, and $x$ is an unknown vector. For example, it is often preferable to write the system of equations (5.1) in the matrix form

$$
\begin{bmatrix}
3 & 6 \\
4 & -5
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}
= 
\begin{bmatrix}
5 \\
3
\end{bmatrix}
$$

and to solve the matrix equation using the `LinearAlgebra:-LinearSolve` command:

```plaintext
> A := Matrix( [[3,6], [4,-5]] ): # define matrix A
> b := Vector( [5,3] ): # define vector b
> A, b; # display A and b

> x := LinearAlgebra:-LinearSolve(A, b);
```

Defining large matrices and vectors may require using loops or subroutines.

### 5.1.2 Matrices and the Linear Algebra package

Matrices can be generated in several ways. For example, the same matrix

$$T = \begin{bmatrix}
1 & 2 & 0 \\
0 & 3 & 4 \\
5 & 0 & 6
\end{bmatrix}$$

can be defined row-by-row with
column-by-column in

\[ T := \begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 4 \\ 5 & 0 & 6 \end{bmatrix} : \]

or entry-by-entry:

\[ T := \begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 6 \end{bmatrix} : \]

The command \( \text{Matrix}(m,n) \) outputs an \( m \times n \) zero matrix. Thus only nonzero entries require assignment. The 100 \( \times \) 100 tridiagonal matrix

\[
A = \begin{bmatrix}
2 & 1 & & & \\
1 & 2 & 1 & & \\
& \ddots & \ddots & \ddots & \\
& & \ddots & \ddots & 1 \\
& & & 1 & 2
\end{bmatrix}
\]

can be generated using a loop:

\[
A := \text{Matrix}(100,100) : \\
A[1,1] := 2 : \\
\text{for } k \text{ from 2 to 100 do} \\
\text{end do:}
\]

After generating some matrices and vectors, we can perform addition “+”, subtraction “-” and multiplication “.” the same way as number operations, except that the multiplication of matrices \( A \) and \( B \) is carried out by \( A.B \), not \( A*B \). Further algebraic manipulations require the \texttt{LinearAlgebra} package. Readers are encouraged to look up the manual by \texttt{?LinearAlgebra}. The command \texttt{LinearAlgebra:-LinearSolve} is the linear system solver in the package. The following examples demonstrate some common matrix operations using the \texttt{LinearAlgebra} package.

\[
A := \text{LinearAlgebra:-RandomMatrix}(3,2,\text{generator}=0..5) : \quad \# \text{generate random matrices } A \text{ and } B \\
A := \text{LinearAlgebra:-RandomMatrix}(2,4,\text{generator}=0..5) : \\
A, B ; \quad \# \text{display } A \text{ and } B
\]
5.1. SOLVING EQUATIONS FOR CONVENTIONAL SOLUTIONS

\[
\begin{bmatrix}
4 & 4 \\
5 & 3 \\
4 & 2
\end{bmatrix}
\begin{bmatrix}
4 & 4 & 2 & 1 \\
4 & 3 & 4 & 5
\end{bmatrix}
\]

\[
C := A.B \quad \# \text{multiply } A \text{ and } B
\]

\[
\begin{bmatrix}
32 & 28 & 24 & 24 \\
32 & 29 & 22 & 20 \\
24 & 22 & 16 & 14
\end{bmatrix}
\]

\[
\text{LinearAlgebra:-Transpose}(C); \quad \# \text{transpose of } C
\]

\[
\begin{bmatrix}
32 & 32 & 24 \\
28 & 29 & 22 \\
24 & 22 & 16 \\
24 & 20 & 14
\end{bmatrix}
\]

\[
E := A[2..3,1..2]; \quad \# \text{extract a 2x2 submatrix of } A \text{ from rows 2-3 and columns 1-2}
\]

\[
\begin{bmatrix}
5 & 3 \\
4 & 2
\end{bmatrix}
\]

\[
\text{LinearAlgebra:-MatrixInverse}(E); \quad \# \text{obtain the inverse matrix}
\]

\[
\begin{bmatrix}
-1 & 3 \\
2 & -5
\end{bmatrix}
\]

5.1.3 Percent mixture problem \textit{(Elementary Algebra, Chemistry)}

A chemist needs to mix chemical solutions, say mixing acid solutions in 11\% and 4\% to produce 700 ml of 6\% percent solution, in the lab. Since he is doing it often with different concentrations, he decides to write a Maple program to solve the general percent mixture problem: Using solutions of concentrations \(a\%\) and \(b\%\) to produce a new solution of concentration \(c\%\) in the amount of \(s\) ml, how many ml of each solution is needed?

From elementary algebra, it is easy to set up the system of literal equations

\[
\begin{aligned}
x + y &= s \\
a x + b y &= c s
\end{aligned}
\]

and a Maple program can be written accordingly:

\[
\text{PercentMixture} := \text{proc}(\text{a} :: \text{numeric}, \# \text{concentration 1} \\
\text{b} :: \text{numeric}, \# \text{concentration 2} \\
\text{c} :: \text{numeric}, \# \text{result concentration} \\
\text{s} :: \text{numeric} \quad \# \text{amount needed} \\
\text{)} \\
\text{local equations, variables, solutions, x, y;}
\]
5.1.4 Creating quadrature rules (Numerical Analysis)

We have seen some numerical integration formulas such as the Simpson's rule (c.f. page 57). One may wonder how it was created and how to create other rules for intellectual curiosity or, more importantly, for practical purposes. We can easily recreate these formulas by solving certain equations using simple approximation models.

First of all, all definite integrals can be transformed to the interval \([0, 1]\) by the following identity (why?)

\[
\int_a^b f(x) \, dx = (b - a) \int_0^1 f((b - a)t + a) \, dt
\]

Therefore, we simplify the general integration problem to \(\int_0^1 f(x) \, dx\) without loss of generality.

A common form of the approximation model for numerical integration is a linear combination of function values:

\[
\int_0^1 f(x) \, dx \approx a_1 f(x_1) + a_2 f(x_2) + \cdots + a_n f(x_n)
\]

In this model, parameters \(a_1, \cdots, a_n\) are coefficients to be determined, and \(x_1, \cdots, x_n \in [0, 1]\) are points in the interval. The points \(x_1, \cdots, x_n \in [0, 1]\) can be predetermined or unknowns. So we have a quite some freedom to create a lot of quadrature rules.

As an example, we can recreate the Simpson's rule by setting up the approximation model

\[
\int_0^1 f(x) \, dx \approx uf(0) + vf(s) + wf(1)
\]

with 4 parameters \(u, v, w\) and \(s\) to be determined. Among them, the parameter \(s\) is used to identify the best possible point inside \((0, 1)\) for optimizing the accuracy. So we need 4 equations for deciding the values of the parameters. In other words, we can have 4 wishes to be met since we have 4 helpers (parameters).
Equations come from what we wish for the rule \((5.9)\). There are again many possible “wishes”. A typical “wish list” in numerical analysis is for the formula \((5.9)\) to give exact value of the integration for polynomials of highest possible degree. The justification for this requirement is the Weierstrass Theorem, which asserts every continuous function can be approximated by polynomials. Therefore, we would like the model to be exact for 4 functions

\[
\begin{align*}
  f_1(x) &= 1, \\
  f_2(x) &= x, \\
  f_3(x) &= x^2, \\
  f_4(x) &= x^3
\end{align*}
\]

making the resulting formula exact for all polynomials with degrees up to three. Substituting \(f(x) = f_k(x)\) for \(k = 1, 2, 3, 4\) into the model \((5.9)\) yields a sequence of 4 equations

\[
\int_0^1 f_k(x) \, dx = uf_k(0) + vf_k(s) + wf_k(1), \quad \text{for } k = 1, 2, 3, 4. \tag{5.10}
\]

Namely

\[
\begin{align*}
  u + v + w &= 1, \\
  vs + w &= 1/2, \\
  vs^2 + w &= 1/3, \\
  vs^3 + w &= 1/4
\end{align*}
\]

and Maple solves the system of equations \((5.11)\) instantly:

\[
\begin{align*}
  s &= \frac{1}{2}, \\
  u &= \frac{1}{6}, \\
  v &= \frac{2}{3}, \\
  w &= \frac{1}{6}
\end{align*}
\]

Applying this solution in \((5.9)\) yields the Simpson’s rule on \([0, 1]\)

\[
\int_0^1 f(x) \, dx \approx \frac{1}{6} f(0) + \frac{2}{3} f(0.5) + \frac{1}{6} f(1),
\]

which can be converted to the general Simpson’s rule using \((5.7)\):

\[
\int_a^b f(x) \, dx \approx \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{b-a}{2}\right) + f(b) \right].
\]

For mature Maple users, it may be more intuitive, and certainly more straightforward to solve the equations \((5.10)\) directly as follows.

\[
\begin{align*}
  &> f[1] := x \rightarrow 1: \quad \# \text{define the functions } f[1], f[2], f[3], f[4] \\
  &> f[2] := x \rightarrow x: \\
  &> f[3] := x \rightarrow x^2: \\
  &> f[4] := x \rightarrow x^3: \\
  &> \text{solve( \{ \text{seq( int(f[k](x), x=0..1) = u*f[k](0) + v*f[k](s) + w*f[k](1), k=1..4)}, u,v,u,s) \}; } \quad \# \text{the equations} \\
  &> \{ s = \frac{1}{2}, u = \frac{1}{6}, v = \frac{2}{3}, w = \frac{1}{6} \}
\end{align*}
\]

For further studies, readers can find some good numerical analysis textbooks and look up the sections on numerical integration.
### 5.1.5 Interpolations (Numerical Analysis)

Suppose we know the values of a function $f(x)$ at several points, say, the data in Table 5.1, how do we estimate $f(0.8)$, $f(1.6)$, etc.?

<table>
<thead>
<tr>
<th>$x$</th>
<th>0.6</th>
<th>0.9</th>
<th>1.4</th>
<th>1.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>0.56</td>
<td>0.78</td>
<td>0.99</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table 5.1: Approximate values of $\sin(x)$

This is an interpolation problem. The function is obviously not linear. A polynomial is a convenient choice for such an approximation. The classical Weierstrass Theorem ensures that every continuous function in a closed interval can be approximated by a polynomial within any given accuracy bound.

In Table 5.1 there are four known values of the function $f(x)$, it is expected that we can determine four parameters in an approximation model. If the model is a polynomial, we should be able to determine a cubic polynomial $p(x) = ax^3 + bx^2 + cx + d$ for its four coefficients. The general (polynomial) interpolation problem can be stated as follows: For given data of a function $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_n, f(x_n))$, find a polynomial

$$p(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_2 x + a_1$$

such that

$$p(x_1) = f(x_1), \quad p(x_2) = f(x_2), \quad \ldots, \quad p(x_n) = f(x_n).$$

Solving the system of $n$ linear equations in (5.13) for the unknowns $a_1, a_2, \ldots, a_n$ in (5.12) is easy today using personal computers equipped with software like Maple. However, it was difficult task not too long ago. So mathematical geniuses had to improvise interpolation methods in pre-computer ages. One of the clever solutions was given by Lagrange (1736-1813): For function values 0.56, 0.78, 0.99, 0.95, the interpolation polynomial can be written instantly as

$$p(x) = 0.56 p_1(x) + 0.78 p_2(x) + 0.99 p_3(x) + 0.95 p_4(x)$$

where

$$p_1(x) = \frac{(x-0.9)(x-1.4)(x-1.9)}{(0.6-0.9)(0.6-1.4)(0.6-1.9)}, \quad p_2(x) = \frac{(x-0.6)(x-1.4)(x-1.9)}{(0.9-0.6)(0.9-1.4)(0.9-1.9)},$$

$$p_3(x) = \frac{(x-0.6)(x-0.9)(x-1.9)}{(1.4-0.6)(1.4-0.9)(1.4-1.9)}, \quad p_4(x) = \frac{(x-0.9)(x-0.9)(x-1.4)}{(1.9-0.9)(1.9-0.9)(1.9-1.4)}.$$  

At any one of the point $x = 0.6, 0.9, 1.4$ or 1.9, the polynomial $p(x)$ takes the identical value of $f(x)$ since, say, $p(0.9) = 0.56 \cdot 0 + 0.78 \cdot 1 + 0.99 \cdot 0 + 0.95 \cdot 0 = 0.78 = f(0.9)$.

The data in Table 5.1 is taken from $\sin x$ rounded to two digits. Figure 5.1 shows that the Lagrange interpolation polynomial matches the data and approximates the underlying
function well in the interval $[0.6, 1.9]$. On the other hand, Figure 5.1 also indicates that the interpolation polynomial is practically meaningless at $x$ values far away from the given data. In philosophical sense, its futile to predict distant future based solely on current trend.

Besides Lagrange, mathematical legends such as Issac Newton (1643-1727), Carl Friedrich Gauss (1777-1855) and Friedrich Bessel (1784-1846) also developed several versions of finite difference linear system formulas that enabled finding interpolation polynomials and avoiding dreadful linear system solving for centuries. Other than displaying elegance and ingenuity, these formulas may no longer hold any advantage today over straightforward linear system solving in scientific computing. Furthermore, it does not take a genius to find interpolation polynomials by solving linear systems in more extensive cases like follows: Construct an interpolation polynomial $p(x)$ that matches a function $f(x)$ on the following data:

$$f(1) = f'(1) = 0.37, \quad f(2) = 0.54, \quad f(3) = 0.45, \quad f''(1) = -0.37, \quad f''(2) = -0.27. \quad (5.15)$$

These data come from an underlying function $f(x) = x^2e^{-x}$. We leave this problem as an exercise (Problem 7).

5.1.6 Cubic splines (Numerical Analysis)

Straightforward interpolations in §5.1.5 using high degree polynomial may lead to inaccurate approximations. For example, the data in Table 5.2 constitute approximate values of

$$f(x) = (x + 1)^2e^{-x^2}.$$

A routine interpolation requires a polynomial of degree 16 that is unpleasantly high. As a result, the interpolation polynomial deviates far from the underlying function, as shown in Figure 5.2

The results shows that we need a better approximation model when the number of known function values is high (e.g. 10+).
Table 5.2: Approximate values of \( f(x) = (x + 1)^2 e^{-x^2} \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>0.0</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>2.0</td>
<td>2.20</td>
<td>2.38</td>
<td>2.54</td>
<td>2.67</td>
<td>2.75</td>
<td>2.79</td>
<td>2.77</td>
<td>2.71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x )</th>
<th>0.9</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>1.4</th>
<th>1.5</th>
<th>1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>2.61</td>
<td>2.47</td>
<td>2.31</td>
<td>2.15</td>
<td>1.98</td>
<td>1.81</td>
<td>1.66</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Figure 5.2: The interpolation polynomial \( p(x) \) from the data in Table 5.2 compared with the underlying function \( f(x) = (x + 1)^2 e^{-x^2} \).

Now that a single interpolation polynomial of high degree fails, how about several low degree polynomials patch together as a piecewise function? That’s the idea of what is known as spline approach. A popular choice is the cubic spline model as follows, using the data in Table 5.2.

\[
s(t) = \begin{cases} 
  p_1(t) = a_1(t - x_0)^3 + b_1(t - x_0)^2 + c_1(t - x_0) + d_1 & \text{for } t \in [x_0, x_1] \\
  p_2(t) = a_2(t - x_1)^3 + b_2(t - x_1)^2 + c_2(t - x_1) + d_2 & \text{for } t \in [x_1, x_2] \\
  \quad \ldots \quad \ldots \quad \ldots \\
  p_{16}(t) = a_{16}(t - x_{16})^3 + b_{16}(t - x_{16})^2 + c_{16}(t - x_{16}) + d_{16} & \text{for } t \in [x_{15}, x_{16}] 
\end{cases} \tag{5.16}
\]

where the nodes \( x_0 < x_1 < \ldots < x_{16} \) are the \( x \) values in Table 5.2.

There are 64 parameters \( a_j, b_j, c_j, d_j, j = 1, \ldots, 16 \) to be determined for \( s(x) \) in (5.16). So we can have as many “wishes” for the model to accomplish:

- Each piece polynomial matches the underlying function at the relevant nodes. Namely
  \[
  p_j(x_{j-1}) = f(x_{j-1}), \quad p_j(x_j) = f(x_j), \quad j = 1, \ldots, 16. \tag{5.17}
  \]
  Wish count: 32.

- At each interior point \( x_j, \quad j = 1, \ldots, 15 \), the two relevant piece polynomials \( p_j(t) \) and \( p_{j+1}(t) \) join smoothly:
  \[
  p_j'(x_j) = p_{j+1}'(x_j), \quad j = 1, \ldots, 15. \tag{5.18}
  \]
  Wish count: 15.
5.1. SOLVING EQUATIONS FOR CONVENTIONAL SOLUTIONS

- At each interior point \( x_j, \ j = 1, \ldots, 15, \) the two relevant piece polynomials \( p_j(t) \) and \( p_{j+1}(t) \) join more smoothly:

\[
p_j''(x_j) = p_{j+1}''(x_j), \quad j = 1, \ldots, 15. \tag{5.19}
\]

Wish count: 15.

- We now have \( 32 + 15 + 15 = 62 \) wishes, two short of the 64 allowed. This shortage of equation may open the door for creativity. One of the standard approach to fill these two equations is the so-called *natural spline function* by setting zero curvature at the two ends

\[
p'_1(x_0) = p''_{16}(x_{16}) = 0. \tag{5.20}
\]

We can use Maple to construct such a cubic spline conveniently as follows.

**Step 1:** Set up the polynomial pieces:

\[
> \text{for } k \text{ from 1 to 16 do}
> \quad \text{p}[k] := a[k]*(t-x[k])^3+b[k]*(t-x[k])^2+c[k]*(t-x[k]) + d[k]
> \text{end do;}
\]

**Step 2:** Set up the 64 equations:

\[
> k := 0; \quad \text{# initialize the equation counter}
> \text{for } j \text{ from 1 to 16 do} \quad \text{# setup equations for matching function values}
> \quad k := k + 1: \text{eqn}[k] := \text{subs}(t=x[j],p[j]) = y[j]:
> \quad k := k + 1: \text{eqn}[k] := \text{subs}(t=x[j+1],p[j]) = y[j]:
> \text{end do;}
> \text{for } j \text{ from 1 to 15 do} \quad \text{# setup equations for matching derivatives}
> \quad k := k + 1: \text{eqn}[k] := \text{subs}(t=x[j],\text{diff}(p[j],t)) = \text{subs}(t=x[j],\text{diff}(p[j+1],t))
> \quad k := k + 1: \text{eqn}[k] := \text{subs}(t=x[j],\text{diff}(p[j-1],t,t)) = \text{subs}(t=x[j],\text{diff}(p[j+1],t,t))
> \text{end do;}
> \text{# the final two equations}
> k := k + 1: \text{eqn}[k] := \text{subs}(t=x[1],\text{diff}(p[1],t,t))=0:
> k := k + 1: \text{eqn}[k] := \text{subs}(t=x[16],\text{diff}(p[16],t,t))=0:
\]

**Step 3:** Solve the equations and assign solutions:

\[
> \text{solve( \{seq(eqn[k],k=1..64)\}, \{seq(op([a[j],b[j],c[j],d[j]]),j=1..16)\})};
> \text{assign(%)};
\]

**Step 4:** Assemble the spline as a piecewise function

\[
> s := \text{piecewise(seq(op([t<=x[j+1],p[j]]),j=1..15),p[16]);}
\]
Figure 5.3 shows that the cubic spline $s(x)$ matches the function $f(x)$ nicely, with maximum deviation 0.005. In comparison, the maximum error of the straightforward interpolation is about 0.75, which is 150 times larger than the spline's.

![Graph showing cubic spline and deviation](image)

**Figure 5.3**: The cubic spline $s(x)$ from the data in Table 5.2 matches the underlying function $f(x) = (x+1)^2e^{-x^2}$ with small error.

There are generally two equations short for cubic splines unless extra data are available and used. The two equations in (5.19) for the natural spline is quite arbitrary. Without them, the resulting linear system would be underdetermined. However, underdetermined linear system can be still be solved. In (5.24) we explore the strategy for computing splines without imposing the artificial equations.
5.2 Exploring scientific computing

5.2.1 Sums of powers (Elementary Algebra, Discrete Mathematics)

The following formulas for sum of consecutive powers are well known:

\[ \sum_{k=1}^{n} k^0 = 1 + 1 + \cdots + 1 = n \]  \hspace{1cm} (5.21)

\[ \sum_{k=1}^{n} k^1 = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \]  \hspace{1cm} (5.22)

\[ \sum_{k=1}^{n} k^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n \]  \hspace{1cm} (5.23)

The question is, can we construct such a formula for \(1^m + 2^m + \cdots + n^m\), the sum of any power \(m\)?

Looking closely at (5.21), (5.22) and (5.23), the formula is a polynomial of \(n\) with degree 1, 2, 3 for \(m = 0, 1\) and 2 respectively. So one can easily come up with a conjecture

**Conjecture.** The sum of \(n\) consecutive \(m\)-th power is a polynomial of \(n\) with degree \(m+1\). Namely:

\[ \sum_{k=1}^{n} k^m = a_{m+1}n^{m+1} + a_m n^m + \cdots + a_1 n + a_0. \]  \hspace{1cm} (5.24)

Assuming this conjecture is true, constructing formula (5.24) becomes finding coefficients \(a_0, a_1, \ldots, a_{m+1}\) by setting up \(m + 2\) equations using \(m + 2\) values of \(n\), say \(n = 1, 2, \ldots, m + 2\). In other words, this is a special interpolation problem for the function \(f_m(n) = \sum_{k=1}^{n} k^m\) using the values of \(f_m(1), f_m(2), \ldots, f_m(m + 2)\) calculated from the sum of powers. Since the coefficients and the right-hand sides are all integers and can be obtained exactly, the resulting linear system can be solved exactly by the symbolic computation that Maple is built for.

The program for constructing such a formula for sum of consecutive \(m\)-th powers will be left as an exercise (Problem 5), with the formulas to be presented in factored form such as

\[ \text{PowerSum}(9); \]

\[ \sum_{k=1}^{n} k^9 = \frac{1}{20}n^2(n^2 + n - 1)(2n^4 + 4n^3 - n^2 - 3n + 3)(n + 1)^2 \]
To complete the process of creating such a formula, a proof is required. A formula for a sum of powers can typically be proved by a mathematical induction process in the following example of computer assisted proof.

**Example.** Prove (5.23) by mathematical induction.

**Solution.** Mathematical induction for proving

\[ \sum_{k=1}^{n} f(k) = g(n) \]

from general input \( f \) and \( g \) is programmable. The following is a test execution of the program with sample print-out:

```maple
> f := k->k^3; # define the k-th term in the sum
> g := n -> n*(n+1)*(2*n+1)/6; # define the formula for the sum
> induction(f,g); # running the math induction program
```

Prove:

\[ \sum_{k=1}^{n} k^3 = \frac{1}{6}n(n+1)(2n+1) \]

Proof. For \( n = 1 \)

\[ \sum_{k=1}^{n} k^2 = 1 \]

\[ \frac{1}{6}n(n+1)(2n+1) = 1 \]

Therefore the formula is true for \( n=1 \). Assume for \( n=m \)

\[ \sum_{k=1}^{m} k^2 = \frac{1}{6}m(m+1)(2m+1) \]

Then for \( n = m+1 \)

\[ \sum_{k=1}^{n} k^2 = \sum_{k=1}^{m} k^2 + (m + 1)^2 \]

\[ = \frac{1}{6}(m(m+1)(2m+1) + (m + 1)^2 \]

\[ = \frac{1}{3}m^3 + \frac{3}{2}m^2 + \frac{13}{6}m + 1 \]

and

\[ \frac{1}{6}n(n+1)(2n+1) = \frac{1}{6}(m + 1)(m+2)(2m+3) \]

By Maple simplification,

\[ \frac{1}{3}m^3 + \frac{3}{2}m^2 + \frac{13}{6}m + 1 = \frac{1}{6}(m + 1)(m+2)(2m+3) \]

Therefore, the formula is true for all \( n \). QED

Writing such a program will be a part of an exercise (Problem 6). Combined with the computer assisted proof, the process of creating a formula will be completed under the assumption that the Maple symbolic computation is accurate.
5.2.2 Viète’s root-finding method (Numerical Analysis, Algebra)

Polynomial root-finding is one of the oldest problems in the history of mathematics, dating back to the third millennium B.C. One of the root-finding method, attributed to Viète, can be easily derived and implemented using a computer algebra system. Consider a given monic polynomial of, say, degree three:

\[ p(z) = z^3 - 4z^2 + 9z - 10. \]  

(5.25)

The objective of root-finding is, of course, to find the three roots \( x_1, x_2 \) and \( x_3 \) such that

\[ p(z) = (z - x_1)(z - x_2)(z - x_3) \]  

(5.26)

We can use Maple to expand (5.26) and collect its coefficients

```maple
> p := expand((z-x[1])*(z-x[2])*(z-x[3]));

p := \[z^3 - x_1 z^2 - x_2 z^2 - x_3 z^2 + x_1 x_2 z + x_1 x_3 z + x_2 x_3 z - x_1 x_2 x_3\]
```

(5.27)

Equating coefficients in (5.27) with those in (5.25), the roots \( x_1, x_2 \) and \( x_3 \) must satisfy the system of equations

\[
\begin{align*}
-x_1 - x_2 - x_3 &= -4 \\
x_1 x_2 + x_1 x_3 + x_2 x_3 &= 9 \\
-x_1 x_2 x_3 &= -10
\end{align*}
\]  

(5.28)

which can be shortened in vector form

\[ f(x) = 0 \quad \text{for} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \]  

(5.29)

the vector-valued function \( f \) generated by Maple

```maple
> a := [-4, 9, -10]:
> F := Vector(seq(coeff(p,z,3-k)-a[k],k=1..3));

\[ f := \begin{bmatrix} -x_1 - x_2 - x_3 + 4 \\ x_1 x_2 + x_1 x_3 + x_2 x_3 - 9 \\ -x_1 x_2 x_3 + 10 \end{bmatrix} \]
```

For the general system of nonlinear equations in the form of (5.29), Newton’s iteration

\[ x^{(k+1)} = x^{(k)} - J(x^{(k)})^{-1} f(x^{(k)}) \quad k = 0, 1, \ldots \]  

(5.30)

is a common choice of method, where \( J(x) \) is the Jacobian matrix of \( f(x) \) at \( x \) and attainable by the Maple command:
To carry out the Newton’s iteration (5.30) on the system (5.28), we can randomly choose an initial iterate, say, \( \mathbf{x}^{(0)} = [1.0, 2 + 3i, 2 - 3i]^\top \). The immediate subsequent iterate \( \mathbf{x}^{(1)} \) can be obtained as follows.

\[
J := \begin{bmatrix}
-1 & -1 & -1 \\
x_2 + x_3 & x_1 + x_3 & x_1 + x_2 \\
-x_2 x_3 & -x_1 x_3 & -x_1 x_2
\end{bmatrix}
\]

At the sixth Newton iteration step, we obtain the accurate root vector

\[
\mathbf{x}^{(5)} = \begin{bmatrix}
2. - 4.038678347315810^{-28}i \\
1. + 2i \\
1. - 2i
\end{bmatrix}
\]

If all the roots of the polynomial \( p(x) \) are simple, Newton’s iteration (5.30) always converge to those roots from a random initial iterate. Implementation of Viéte’s method is an exercise (Problem 9).

### 5.2.3 From Viéte to Durand-Kerner (Numerical Analysis)

Viéte’s method in §5.2.2 can also be carried out in symbolic computation that leads to rediscovery of Durand-Kerner method. For a few integer values of \( n \), say, \( n = 3, 4, 5 \), consider polynomial

\[
p(z) = z^n + a_1 z^{n-1} + \cdots + a_{n-1} z + a_n
\]
5.2. EXPLORING SCIENTIFIC COMPUTING

with symbolic coefficients \( a_1, a_2, \ldots, a_n \). Let the roots \( x_1, \ldots, x_n \) be variables such that

\[
p(z) = (z - x_1)(z - x_2) \cdots (z - x_n).
\] (5.32)

Expanding (5.32) and collecting its coefficients yield

\[
p(z) = z^n + p_1(x_1, \ldots, x_n) z^{n-1} + \cdots + p_{n-1}(x_1, \ldots, x_n) z + p_n(x_1, \ldots, x_n).
\] (5.33)

The root-finding problem for \( p(x) \) becomes solving the nonlinear system in vector form

\[
f(x) = 0, \quad \text{for } x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}
\] (5.34)

where

\[
f(x) = \begin{bmatrix} p_1(x_1, \ldots, x_n) - a_1 \\ \vdots \\ p_n(x_1, \ldots, x_n) - a_n \end{bmatrix}.
\] (5.35)

Use symbolic computation to carry out one step of Newton iteration

\[
y = x - J(x)^{-1} f(x).
\] (5.36)

Paying particular attention to \( J(x)^{-1} f(x) \) in terms of \( x_1, \ldots, x_n \) and \( a_1, \ldots, a_n \), one can rediscover the Durand-Kerner iteration, which is a clever way for carrying out Newton’s iteration for Viète’s method without solving the linear system

\[
J(x^{(k)})[x^{(k+1)} - x^{(k)}] = -f(x^{(k)}).
\]

5.2.4 Cubic splines, revisited (Numerical Analysis)

Without generating two extra equations in (5.19), the number of equations for constructing cubic splines is less than the number of variables. Such a linear system is said to be underdetermined.

There are effective methods for solving underdetermined linear systems even though they are apparently less well-known. One of such methods uses the so-called matrix QR decomposition, as we shall illustrate below using a simple example.

Consider the linear system \( Ax = b \) with

\[
A = \begin{bmatrix} -1 & 2 & -1 & -2 & 2 \\ 1 & 0 & 2 & 3 & -1 \\ 0 & 0 & 3 & 3 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix}.
\]

Assume \( A \) is of full rank.

Step 1: Make a QR decomposition of \( A^\top \).
> with(LinearAlgebra): # load the LinearAlgebra package
> Q, R := QRDecomposition(Transpose(A),fullspan);

\[
Q.R = \begin{bmatrix}
-0.26726 & -0.0849787 & 0.761497 & -0.532996 & 0.241526 \\
0.534520 & -0.623247 & 0.182104 & -0.144918 & -0.521227 \\
-0.267260 & -0.481603 & -0.595952 & -0.532101 & 0.241523 \\
-0.534520 & -0.566591 & 0.165541 & 0.604557 & 0.0190936 \\
0.534520 & -0.226637 & 0.066218 & 0.217372 & 0.781849 \\
\end{bmatrix}
\begin{bmatrix}
3.74167 & -2.93986 & -2.93987 \\
0. & -2.52134 & -2.91795 \\
0. & 0. & -1.35746 \\
0. & 0. & 0. \\
\end{bmatrix}
\]

Step 2: Extract matrix blocs \( Q_1, Q_2 \) and \( R \) from \( Q \) and \( R \):

> Q1 := Q[1..-1,1..3]: Q2 := Q[1..-1,4..5]: R1 := R[1..3,1..3]:
> Q1, Q2, R1;

\[
\begin{bmatrix}
-0.26726 & -0.0849787 & 0.761497 \\
0.534520 & -0.623247 & 0.182104 \\
-0.267260 & -0.481603 & -0.595952 \\
-0.534520 & -0.566591 & 0.165541 \\
0.534520 & -0.226637 & 0.066218 \\
\end{bmatrix}
\begin{bmatrix}
-0.532996 & 0.241526 \\
-0.144918 & -0.521227 \\
-0.532101 & 0.241523 \\
0.604557 & 0.0190936 \\
0.217372 & 0.781849 \\
\end{bmatrix}
\begin{bmatrix}
3.74167 & -2.93986 & -2.93987 \\
0. & -2.52134 & -2.91795 \\
0. & 0. & -1.35746 \\
\end{bmatrix}
\]

Step 3: Solve the lower triangular system \( R_1^T y = b \) by a forward substitution:

> y := ForwardSubstitute(Transpose(R1),b);

\[
y := \begin{bmatrix}
0.801781 \\
-1.72810 \\
4.18825 \\
\end{bmatrix}
\]

Step 4: Obtain the minimum norm solution \( x_0 = Q_1 y \)

> x0 := Q1.y;

\[
x0 := \begin{bmatrix}
3.12191 \\
2.26830 \\
-1.87803 \\
1.24388 \\
1.09756 \\
\end{bmatrix}
\]

The general solution is

\[
x = x_0 + Q_2 z \quad \text{for any} \quad z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}
\]

In fact, it can be proved and verified that columns of \( Q_2 \) form an orthonormal basis for the kernel of \( A \), and \( A = R_1^T Q_1^T \). The equation \( Ax = b \) thus becomes \( R_1^T (Q_1^T x) = b \), or equivalently

\[
R_1^T y = b \quad \text{and} \quad Q_1^T x = y
\]

The unique solution \( x_0 \) of \( R_1^T x = b \) is the minimum norm solution of \( Ax = b \) (prove it!).
5.3 Exercises

1. **Percent mixture** *(Chemistry, Elementary algebra)* Rewrite the **PercentMixture** program in §5.1.3 by converting the equation to matrix-vector form and by applying the command **LinearAlgebra:-LinearSolve**.

2. **Creating quadrature rule I** *(Numerical Analysis)* Suppose one needs to approximate the definite integral $\int_a^b f(x) \, dx$ where the function value is already known at the midpoint of the interval $[a, b]$. Create a quadrature rule using this value along with two additional function evaluations in the interval. Namely, determine the parameters $u, v, w, s, t$ in the following approximation model

$$\int_0^1 f(x) \, dx \approx u f(s) + v f(\frac{1}{2}) + w f(t)$$

and convert it to a formula for $\int_a^b f(x) \, dx$.

3. **Creating quadrature rule II** *(Numerical Analysis)* Recreate Milne’s quadrature rule using the approximation model

$$\int_0^1 f(x) \, dx \approx s f(0) + t f(\frac{1}{4}) + u f(\frac{1}{2}) + v f(\frac{3}{4}) + w f(1)$$

and determine the parameters $s, t, u, v$ and $w$.

4. **Creating quadrature rule III** *(Numerical Analysis)* Milne’s quadrature rule in Problem 3 require five function evaluations in the interval. By adjusting two evaluation points inside the interval, we can create a more accurate quadrature rule with the same number of function evaluations.

5. **Sum of Powers I** *(Algebra, Discrete Mathematics)* Write a program that, for input nonnegative integer $m$, constructs the formula for $\sum_{k=1}^n k^m$ in the form of (5.24), following the conjecture in §5.2.1.

6. **Sum of Powers II** *(Computer Science)* Write a mathematical induction program following the example in §5.2.1 and recreate similar print-out of the computer assisted proof. Then combine this program with the program in Problem 5 to generate a complete program for generating a formula for sum of powers with proof.

7. **Interpolation** *(Numerical Analysis)* Construct an interpolation polynomial $p(x)$ that matches a function $f(x)$ at its values and derivatives given in (5.15). Plot the polynomial $p(x)$ in comparison with the underlying function $f(x) = x^2e^{-x}$.

8. **Daylight hours** *(Numerical Analysis)* The following table shows the number of daylight hours $h(t)$ on the $t$-th day of each year:

<table>
<thead>
<tr>
<th>$t$</th>
<th>15</th>
<th>74</th>
<th>166</th>
<th>196</th>
<th>258</th>
<th>349</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(t)$</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>
Use the model
\[ h(t) \approx a + b \cos \left( \frac{2\pi t}{365} \right) + c \cos \left( \frac{2\pi t}{365} \right) \]
and the data to find the least squares solution for \( a, b, c \). Then use the result to estimate the number of daylight hours on August 8.

9. Viéte’s method (Numerical Analysis, Algebra) Write a program that implements the Viéte’s method described in § 5.2.2. More specifically, the program should accept a polynomial
\[ z^n + a_1 z^{n-1} + \cdots + a_{n-1} z + a_n \]
and an error tolerance \( \varepsilon \) as input, construct vector-valued function (5.29) and its Jacobian, choose a random initial iterate, execute Newton’s iteration (5.30) until
\[ \| x^{(k+1)} - x^{(k)} \|_2 < \varepsilon, \]
and output the components of \( x^{(k+1)} \) as computed roots.

10. Durand-Kerner method (Numerical Analysis) Follow the strategy described in § 5.2.3 and try to rediscover the Durand-Kerner method. Write a program to implement the method and compare with the program in Problem 9.

11. Natural cubic spline (Numerical Analysis) Following the elaboration of the cubic spline in § 5.1.6, write a program that generates the natural cubic spline (i.e. the piecewise cubic polynomial) the fits the input data vectors \( x \) and \( y \).