An Introduction to Computer Programming (with MATLAB) for Scientific Computing

Weizhang Huang[1]

January 15, 2020

[1]Department of Mathematics, The University of Kansas, Lawrence, KS 66045 USA.

Preface

These notes are prepared for students with basic knowledge on calculus and matrix theory. Students are not required to have prior knowledge on the numerical methods discussed in these notes. Students who are interested in numerical analysis and scientific computing are encouraged to take courses and/or read textbooks in the area. An example is Burden, Faires, and Burden [1].

Contents

Chapter 1

Programming Basics

In this chapter we study programming basics, including decimal and binary number systems and their conversion, floating-point number representations in a computer, and common components among computer languages.

1.1 Decimal and binary number systems and conversion

Numbers we use in daily life are called decimal numbers. Examples include 0, 12, 3.1415926 and 1234.56. Take 1234.56 as an example. We know that 1 is the digit for thousands, 2 for hundreds, 3 for tens, 4 for ones, 5 for tenths, and 6 for hundredths. Indeed, the number is read as one thousand two hundred thirty four point five six. Translating this into mathematics, we can express the number into a series of powers of 10, i.e.,

$$1234.56 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times \frac{1}{10^1} + 6 \times \frac{1}{10^2}.$$

For this reason, decimal numbers are also called base-10 numbers. The digits we use are 0, 1, ..., and 9.

On the other hand, most modern computers use the so-called binary number system, which uses base 2 and the digits 0 and 1. An example is 11011.01. To distinguish numbers in binary and decimal systems, we denote 11011.01 in the binary number system by $(11011.01)_2$. It can be expressed into a series of powers of 2 as

$$(11011.01)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2}.$$

This explains the meanings of the digits. Moreover, it can be used to convert a binary number into a decimal number. Indeed, carrying out the calculations of the right-hand side terms, we have

$$(11011.01)_2 = 16 + 8 + 0 + 2 + 1 + 0 + 0.25 = 27.25$$

Exercise 1.1.1. Convert the following binary numbers into decimal numbers: $(1010101)_2$, $(111)_2$, and $(0)_2$. □

In the following example we explain how to convert decimal numbers into binary form.

Example 1.1.1. Convert $x = 11$ into binary form.

Since $2^3 < x < 2^4$, we know that x can be expressed as

$$x = a \times 2^3 + b \times 2^2 + c \times 2^1 + d \times 2^0,$$

where $a = 1$ and b, c, and d are to be determined. From the above expression, we have

$$x - 1 \times 2^3 = b \times 2^2 + c \times 2^1 + d \times 2^0.$$

On the other hand, $x - 1 \times 2^3 = 11 - 1 \times 2^3 = 3$. Recall that b can take 0 or 1. If $b = 1$, then the right-hand side of the above equation is at least 4, which is greater than the left-hand side whose value is 3. Thus, b must be zero. With this, we have

$$x - 1 \times 2^3 - 0 \times 2^2 = c \times 2^1 + d \times 2^0$$

and

$$x - 1 \times 2^3 - 0 \times 2^2 = 11 - 1 \times 2^3 - 0 \times 2^2 = 3.$$

Comparing these equations, we get $c = 1$. Inserting this into the above two equations yields

$$x - 1 \times 2^3 - 0 \times 2^2 - 1 \times 2^1 = d \times 2^0,$$

$$x - 1 \times 2^3 - 0 \times 2^2 - 1 \times 2^1 = 11 - 1 \times 2^3 - 0 \times 2^2 - 1 \times 2^1 = 1,$$

which lead to $d = 1$. Thus, we obtain

$$11 = (1011)_2.$$

Now we use a slightly different procedure. First, we find the integer m such that

$$2^{m-1} \leq x = 11 < 2^m.$$

It is not difficult to see $m = 4$. Then, x can be expressed as

$$x = \left(a \times \frac{1}{2^1} + b \times \frac{1}{2^2} + c \times \frac{1}{2^3} + d \times \frac{1}{2^4} \right) \times 2^4 = (0.abcd)_2 \times 2^4.$$

The values of a, b, c, and d are to be determined.

Define $x_0 = x \times 2^{-4}$. Then, $x_0 = 11 \times 2^{-4} = 11/16$ and $2x_0 = 22/16 = 11/8$. Moreover, $2x_0$ has the expression

$$2x_0 = a + b \times \frac{1}{2^1} + c \times \frac{1}{2^2} + d \times \frac{1}{2^3}.$$

Since $2x_0 \geq 1$, we know that $a = 1$.

Next, we define $x_1 = 2x_0 - 1$. We can find that $x_1 = 11/8 - 1 = 3/8$ and the expression of $2x_1$ is

$$2x_1 = b + c \times \frac{1}{2^1} + d \times \frac{1}{2^2}.$$

Since $2x_1 = 6/8 < 1$, we have $b = 0$.

Repeating this process, we can find $c = 1$ and $d = 1$. Thus, we obtain the binary form of 11 is

$$11 = (0.1011)_2 \times 2^4,$$

which is the same as what we obtained before.

The second procedure is summarized in Table 1.1. □

Table 1.1: The summary of the second procedure to convert $x = 11$ into binary form.

x_n	test	binary digits
$x_0 = x \times 2^{-4} = 11/16$	$2x_0 = 22/16 \geq 1$	$a = 1$
$x_1 = 2x_0 - a = 3/8$	$2x_1 = 6/8 < 1$	$b = 0$
$x_2 = 2x_1 - b = 3/4$	$2x_2 = 6/4 \geq 1$	$c = 1$
$x_3 = 2x_2 - c = 1/2$	$2x_3 = 1 \geq 1$	$d = 1$
$x_4 = 2x_3 - d = 0$	$2x_4 = 0$	computation stops

Exercise 1.1.2. Convert the following decimal numbers into binary form using the procedure shown in Table 1.1: 15, 3.25, 8 and 4. □

When x is nonzero, there exists an integer m such that $2^{m-1} \leq x < 2^m$. Then, x can be expressed as

$$x = \pm \left(a_1 \times \frac{1}{2^1} + a_2 \times \frac{1}{2^2} + \cdots + a_n \times \frac{1}{2^n} + \cdots \right) \times 2^m \tag{1.1}$$

$$= \pm (0.a_1 a_2 \cdots a_n \cdots)_2 \times 2^m, \tag{1.2}$$

where $a_1 = 1$ and $a_n = 0$ or 1 for $n = 2, 3, \ldots$. Moreover, the procedure in Table 1.1 can be translated into Algorithm 1.1.1 to convert any nonzero decimal number into binary form (1.2).

Exercise 1.1.3. Explain why the zero cannot be expressed into the form (1.2). □

Exercise 1.1.4. Explain why $a_1 = 1$ in the binary form (1.2) for any nonzero decimal number. □

Exercise 1.1.5. Show that Step 2(a) of Algorithm 1.1.1 is correct for $n = 1$. □

Interestingly, (1.2) can be used to show how (real) numbers are represented in a computer. We first notice that a computer has a finite amount of storage units so a (real) number on a computer can only have a finite number of fractional digits (called the mantissa) and its exponent m is in a bounded range. Thus, a number in a computer looks like

$$x = \pm (0.a_1 a_2 \cdots a_n)_2 \times 2^m, \tag{1.3}$$

Algorithm 1.1.1 Convert nonzero decimal numbers into the binary form (1.2).

1. Initialization: Determine the sign of x and the integer m such that $2^{m-1} \le |x| < 2^m$. Set $x_0 = |x| \times 2^{-m}$.
2. For $n = 1, 2, \cdots$ do
   (a). If $2x_{n-1} \ge 1$, set $a_n = 1$; otherwise, $a_n = 0$.
   (b). Compute $x_n = 2x_{n-1} - a_n$.
   (c). If $x_n = 0$, stop the computation.

where n is a positive integer, $a_1 = 1$, and m (which is also saved in binary form) is an integer in a bounded range. Each of the digits is stored in a bit, with 8 bits being grouped into a byte. As an example, we now consider two most common number representations in a computer. According to the 754-2008 standard of the Institute of Electrical and Electronics Engineers (IEEE), the 32-bit floating-point format (*single precision*) uses

- 1 bit for sign,
- 23 bits for the mantissa (and $n = 24$),
- 8 bits for the exponent.

This is shown in Fig. 1.1. One may notice that a_1 is not saved since it is known to be 1 for any nonzero number. Moreover, the largest integer that can be stored for the exponent is $(11111111)_2 = 255$. In order to represent negative exponents, the actual exponent is obtained by subtracting 127 from what is stored in the exponent part. Thus, the bits are converted into a numeric value as

$$< \text{sign} > \times (0.1 < \text{mantissa} >)_2 \times 2^{<\text{exponent}>-127}.$$

Note that zero cannot be expressed in the above form. Special specifications are defined for zero and several other special values including NaN (Not a Number).

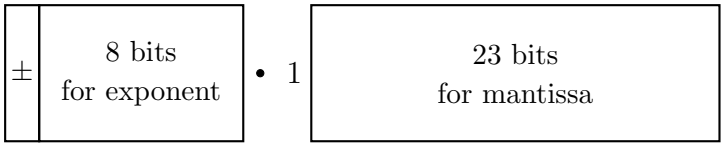

Figure 1.1: IEEE 754-2008 format for 32-bit floating-point numbers.

The IEEE 754-2008 specification of 64-bit floating-point (*double precision*) numbers is shown in Fig. 1.2. The bits are converted into a numeric value as

$$< \text{sign} > \times (0.1 < \text{mantissa} >)_2 \times 2^{<\text{exponent}>-1023}.$$

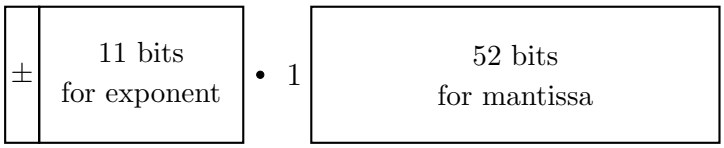

Figure 1.2: IEEE 754-2008 format for 64-bit floating-point numbers.

To conclude this section, we take a look at basic hardware of a computer; see Fig. 1.3. The central processing unit (CPU) is the brain of a computer, which carries out all of basic operations on data. Data are transferred between CPU and the main memory and between the main memory and the hard disk. It is pointed out that numbers are stored and operated in CPU using more bits than they are stored in the main memory or the hard disk. Roundoff error can occur when they are transferred between CPU and the main memory. Roundoff error can also occur when numbers have longer fractional digits than those in the mantissa. Although it is very small ($\sim 10^{-16}$ for double precision), roundoff error can accumulate and be amplified greatly when algorithms are not carefully designed (in this case the algorithms are unstable) or the problem at hand is ill-conditioned. While stability of algorithms and conditioning of problems are two important topics considered in scientific computing, we will not go into detail due to the scope of this course.

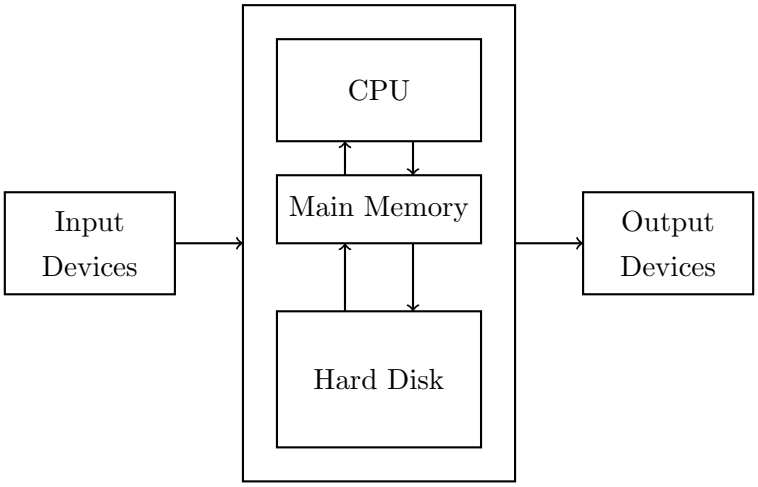

Figure 1.3: Basic hardware for a computer.

1.2 Common components among computer languages

Computer CPUs only understand instructions written in binary (machine code). It is extremely difficult for humans to write programs in machine code. Computer languages have been developed over the years so people can write programs more easily. Those languages serve more or less as translators that translate what people write into something computers

understand. Commonly used computer languages include C, C++, Fortran, Visual Basic, Python, and JavaScript. In this course, we will focus on MATLAB[1], which "combines a desktop environment tuned for iterative analysis and design processes with a programming language that expresses matrix and array mathematics directly" (as quoted from the MATLAB website). Another very useful language is R, which is also a combined language and environment but is used more for statistical computing and graphics.

High-level computer languages share many common features. In this section we overview some of those features, including data types, basic operators, selection and loop statements, and functions. These features will be discussed in detail in the next chapter for MATLAB.

1.2.1 Data types

Data types are classifications that specify what variables or objects can hold in programming, and must be referenced and used correctly. Common examples of data types are

- Float (for single precision real numbers; recall from the previous section that single precision numbers usually occupy 32 bits in computer memory)
- Double (for double precision real numbers; recall from the previous section that single precision numbers usually occupy 64 bits in computer memory)
- Integer (for integer numbers)
- Character (for single characters such as `a`, `4`)
- String (for groups of characters such as `abcd`, `a3c4#`)
- Boolean (`true` or `false`)

1.2.2 Basic operators

Each computer language provides a number of operators. Common examples of basic operators are

- **Assignment Operators** are used to assign the result of an expression to a variable. For example,

  ```
  variable = expression
  ```

  means that the computer evaluates the expression on the right-hand side and then stores the result in the memory unit assigned to the left-hand side variable. Languages may provide various shorthand assignment operators.

- **Arithmetic Operators**: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). Many languages also provide modulo division.

[1] MATLAB® is a trademark of The MathWorks, Inc., Natick, MA 01760.

- **Relational Operators** are used to make comparison: `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), and `!=` or `~=` (not equal to).

- **Logical Operators** are used when more than one conditions are to be tested: `&&` (logical AND), `||` (logical OR), `!` or `~` (logical NOT). The result of any logical expression is either TRUE or FALSE.

1.2.3 Selection and loop statements

In addition to assignment statements, most commonly used statements include selection and loop statements. Brief information and basic formats of these statements are given here. Detailed explanation and examples for the selection and loop statements in MATLAB will be given in §2.4.

A selection statement selects among a set of statements depending on the value of a controlling expression. Selection statements typically include `if` and `switch` statements. Their syntaxes look like

```
if (conditional-expression)
     {
     statements
     }
elseif (conditional-expression) (optional)
     {
     statements
     }
else (optional)
     {
     statements
     }

switch (expression)
     case (constant-expression)
          {
          statements
          }
          ...
     default: (optional)
          {
          statements
          }
```

Common examples of loops include `for` and `while` loops. Their syntaxes looks like

```
for (initialization Statement; test Expression; increment Statement)
     {
     statements
     }

initialization statement
while (condition)
     {
     statements
     increment/decrement statement
     }
```

The `break` statement is often used with loop statements to break the loop.

1.2.4 Functions

Each computer language provides a large set of functions including elementary functions in mathematics. At the same time, it also allows us to define our own functions. Generally speaking, this can be done ***inline*** (embedded in the code) or ***in file*** (programs in file). In addition to the rules for defining these functions, we need to pay special attention to the ***scope*** of the functions in file, for instance, where functions are visible, in the same file or in the same folder, public or private. See §2.5 for MATLAB functions.

1.3 Flowchart

Flowchart is a graphical representation of an algorithm. It is a very useful tool to plan a program because it indicates the flow and steps of information and processing. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Here we do not attempt to learn how to draw flowcharts using common symbols. Instead, we want to emphasize the importance of ***planning (steps and flow)*** for an algorithm before we begin to write the code. We will address this issue in every lab.

Chapter 2

An Introduction to MATLAB

MATLAB, standing for MATrix LABoratory, combines an interactive desktop environment with a matrix-based computer language and provides an abundance of built-in commands, math functions, and tool boxes for numerical computation, visualization, and programming. This chapter presents an introduction to MATLAB. We study its basics such as data types, arithmetic operations, relational operations, selection and loop statements, functions, and graphics. Our goal is to get ourself ready in a reasonably short time for further studies of programming principles for scientific computing in later chapters. It is worth reminding that the best way to learn a computer language is with our hands. So try as many examples as possible on the computer.

2.1 Data representations and types

MATLAB needs to be installed on the computer you will be using. After it is installed, you can start MATLAB by double clicking its icon.

Once MATLAB is started, you can find the command window with the prompt sign `>>`. Then try commands `pwd` and `help`. (In the following block, the left column are MATLAB commands while the right columns are explanations.)

```
>> pwd              This shows the path of the current directory/folder
>> help help        This gets help for help
>> help plot        This gets help for plot
```

Note that MATLAB is case-sensitive. For example, `plot` is different from `Plot`.

A distinct feature of MATLAB (from many other languages such as C or C++) is that there is no need to declare data types for variables before their use. A variable's data type is determined from its first assignment. Nevertheless, it is often more efficient (in terms of memory use and CPU time) to preallocate arrays (including vectors and matrices) before their first use. This can be done by assignment statements. For example, the command

```
>> A = zeros(100, 200);
```

preallocates a 100-by-200 block of memory for variable `A` and initializes it to be zero.

MATLAB's data representations are all interpretations of one basic structure, the matrix. This is a rectangular array of numbers, stored and manipulated internally using the computer's floating point format and operations. The basic matrix data structure can have special interpretations in a number of situations.

```
>> x = 3.14;                    x is a real number (a 1-by-1 matrix)
                                (double precision by default)
>> x = 3.14                     what is the difference?
>> v = [1, 2, 3, 4];            v is a row vector
                                (or a 1-by-4 matrix)
>> v = [1 2 3 4];               what is the difference?
>> u = [1; 2; 3; 4];            u is a column vector
                                (or a 4-by-1 matrix)
>> A = [1, 2; 3, 4];            A is a 2-by-2 matrix
>> b = 0.3 + 0.4*i;             b is a complex number
>> C = [0.1 0.2; 0.3 0.4] ...   what ...  is used for?
>>   + [0.5 0.6; 0.7 0.8]*i;    C is a complex matrix
>> ch = 'a';                    This creates a character 'a' and
                                assigns it to variable ch
>> st = 'Hello World';          st is a string or a character array
>> sm = ['Hello'; 'World'];     sm is a string matrix
>> aa = {'a', 2; 'b', 3};       aa is a cell that uses { }
```

It is important to point out that while being displayed according to their defined sizes and shapes, arrays are actually stored in memory as a single column of elements. Consider the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

Matrix A can be accessed via matrix indexing as $A(3,2) = 8$. On the other hand, matrices are actually stored ***column by column***; in other words, A is stored in memory as a vector containing the sequence of elements 1, 4, 7, 2, 5, 8, 3, 6, 9. As a result, A can also be accessed via the so-called ***linear indexing*** (denoted as $A(:)$); for example, we have $A(2) = 4$ and $A(6) = 8$. Understanding how matrices are stored in memory is often useful in making the code more efficient in implementation.

The MATLAB data types commonly used in scientific computing are listed as follows.

- **Numeric Types** include integers, floating-point data (single precision, double preci-

sion (default), and complex).

- **Logical Data Type** represents `true` or `false` states using the numbers `1` and `0`, respectively.
- **Characters and Strings** include character arrays such as `c = 'Hello World'` and string arrays, such as `str = "Greetings friend"`.
- **Cell Arrays** are arrays that can contain data of varying types and sizes, such as `myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}`.
- **Structures** are arrays with named fields that can contain data of varying types and sizes, such as `s.a = 1; s.b = {'A','B','C'};`
- **Function Handles**: A function handle is a data type that stores an association to a function. To create a handle for a function, precede the function name with an `@` sign. For example, if we have a function called `myfunction`, we can create a handle named `f` as: `f = @myfunction;`

Exercise 2.1.1. Give two examples of variables in each of numeric, logical, character, and cell types. □

2.2 Arithmetic operations

MATLAB has two different types of arithmetic operations: matrix operations and array operations. They are used to perform numeric computations, for example, adding two numbers, raising the elements of an array to a given power, or multiplying two matrices.

Matrix operations follow the rules of linear algebra.

- `+` (matrix addition) and `-` (matrix subtraction). Although they are named matrix addition and subtraction, they should be more precisely called array addition and subtraction (see discussion on array operations below). On the one hand, they act like ordinary matrix addition and subtraction when the operands have the same size. For example,

```
2 + 3 = 5
[1, 2; 3, 4] - [5, 6; 7, 8] = [-4, -4; -4, -4]
```

  On the other hand, when operands have ***compatible sizes*** but not necessarily the same size each input is implicitly expanded as needed to match the size of the other during execution of the computation. We give several examples in the following. The reader can find more information on compatible sizes by searching Compatible Array Sizes for Basic Operations.

(1) `10 + [1, 2; 3, 4]` In this example, the first operand is expanded into `[10, 10; 10, 10]` to match the size of the second operand during execution of the calculation. Thus, we have

$$10 + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Longrightarrow \begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}$$

(2) `[1, 2; 3, 4] + 10` The second operand is expanded into `[10, 10; 10, 10]` to match the size of the first operand during execution of the calculation, i.e.,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 10 \Longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix} = \begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}$$

(3) `[1, 2; 3, 4] + [5, 6]` The second operand is expanded into `[5, 6; 5, 6]` to match the size of the first operand during execution of the calculation. This gives

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 8 & 10 \end{bmatrix}$$

(4) `[1, 2; 3, 4] + [5; 7]` The second operand is expanded into `[5, 5; 7, 7]` to match the size of the first operand during execution of the calculation, which yields

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 \\ 7 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 5 \\ 7 & 7 \end{bmatrix} = \begin{bmatrix} 6 & 7 \\ 10 & 11 \end{bmatrix}$$

(5) `[1;2;3] + [1, 2; 3, 4]` The operands do not have compatible sizes.

- `*`: multiplication. It works for matrices under the rules of linear algebra, with scalars and vectors being considered as special forms of matrices. For example, `A*B` makes sense when the number of columns of `A` is equal to the number of rows of `B`. For any scalar `alpha`, `alpha*A` or `A*alpha` is a standard multiplication of matrices and scalars. For examples,
  `[1, 2; 3, 4]*5 = [5, 10; 15, 20]`
  `[1, 2; 3, 4]*[5, 6]` (error using `*`)
  `[1, 2; 3, 4]*[5; 6] = [17; 39]`
  `[1, 2; 3, 4]*[5, 6; 7, 8] = [19, 22; 43, 50]`

- `/`: right division. This is used mostly for scalars, i.e., a scalar divided by another scalar, for example,
  `3/4 = 0.75`
  `(2*10)/5 = 4`

  It can also be used for matrices (although rare), for example, `A/B` is calculated via `(A'\B')'`, where `'` is the matrix transpose and \ is the left division that is to be explained below.

- `^`: power. For example, `3.1^2.5` represents the 2.5th power of 3.1. Note that the 3rd power of -2 should be expressed as `(-2)^3`. Moreover, `A^2.1` stands for the 2.1th power of matrix `A`. For example,
  `[1, 3; 2, 1]^2 = [7, 6; 4, 7]`

- \: Backslash or left matrix divide. `A\B` is the matrix division of `A` into `B`, which is roughly the same as `INV(A)*B`, except it is computed in a different way. If `A` is an m-by-n matrix with $m < n$ or $m > n$ and `B` is a column vector with m components, or a matrix with several such columns, then `A\B` is the solution in the least squares sense to the under- or overdetermined system of linear system $Ax = B$. For example,

$$\begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}.$$

  The solution for this system is given by
  `x = [1, 2; 1, -1]\[3; 0] = [1; 1]`

On the other hand, **array operations** execute ***element by element*** operations on corresponding elements of vectors, matrices, and multidimensional arrays. More specifically, if the operands have the same size, then each element in the first operand gets matched up with the element in the same location in the second operand. If the operands have compatible sizes, then each input is implicitly expanded as needed to match the size of the other. The period character "`.`" distinguishes array operations from matrix operations. Moreover, MATLAB does not provide `.+` and `.-` because, as mentioned above, `+` and `-` already serve as array addition and subtraction.

- `.*`: array multiplication.
  `[1, 2; 3, 4].*[5, 6; 7, 8] = [5, 12; 21, 32]`
  `[1, 2; 3, 4].*[5, 6] = [5, 12; 15, 24]`
  In this example, the second operand is expanded into `[5, 6; 5, 6]` to match the size of the first operand during execution of multiplication.
  `[1, 2; 3, 4].*[5; 7] = [5, 10; 21, 28]`
  In this example, the second operand is expanded into `[5, 5; 7, 7]` to match the size of the first operand during execution of the calculation.

- `./`: right-array division. This operation is similar to the array multiplication. For example,
  `[1, 2; 3, 4]./[5; 7] = [1/5, 2/5; 3/7, 4/7]`
  where the second operand is expanded into `[5, 5; 7, 7]` to match the size of the first operand.

- `.^`: array power. This operation is similar to the array multiplication. For example,
  `[1, 2; 3, 4].^[5; 7] = [1^5, 2^5; 3^7, 4^7]`
  where the second operand is expanded into `[5, 5; 7, 7]` to match the size of the first

operand. `[1, 2; 3, 4].^3 = [1^3, 2^3; 3^3, 4^3]` where the second operand is expanded into `[3, 3; 3, 3]` to match the size of the first operand.

Exercise 2.2.1. Do the calculations by hand and check your answers with MATLAB.

(1) `[1, 2; 3, 4] - 20` and `[1, 2; 3, 4] - [1;2]`

(2) `[1;2]*[1, 2; 3, 4]` and `[1;2].*[1, 2; 3, 4]`

(3) `[1, 2; 3, 4]^2` and `[1, 2; 3, 4].^2`

(4) `[1;2;3].*[1, 2; 3, 4]`

□

Exercise 2.2.2. (1) Solve the following linear system by hand,

$$\begin{cases} x + y = 15 \\ x - y = 5 \end{cases} \quad \text{or} \quad Au = b \quad \text{where} \quad A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 15 \\ 5 \end{bmatrix}.$$

(2) Input A and b in MATLAB and then find the solution u using the \ operator. Verify your answer with the one obtained by hand.

(3) Try `inv(A)`, `det(A)`, `[V,D] = eig(A)`, and `A'*A`. What are they?

□

2.3 Relational and logical operators

The relational and logical operators perform ***element-wise comparisons*** between two arrays. The arrays must have compatible sizes to facilitate the operation. Arrays with compatible sizes are implicitly expanded to be the same size during execution of the calculation. The entries of the resulting array are either `1` (`true`) or `0` (`false`).

Relational operators in MATLAB include

- `<` (less than)
- `<=` (less than or equal to)
- `>` (greater than)
- `>=` (greater than or equal to)
- `==` (equal to)
- `~=` (not equal to)

For example,

- `[6, 6; 6, 6] > [5, 6; 7, 8] = [1, 0; 0, 0]`
- `[6, 6; 6, 6] > [5, 6] = [1, 0; 1, 0]` (the second operand is expanded)
- `[6, 6; 6, 6] > [5; 6] = [1, 1; 0, 0]` (the second operand is expanded)
- `[1, 2; 3, 4] > 2.5 = [0, 0; 1, 1]` (the second operand is expanded)

The following are commonly used **logical operators**.

- `&` (`and`): `A & B` performs a logical `AND` of arrays `A` and `B` and returns an array containing elements set to either logical `1` (`true`) or logical `0` (`false`). An element of the output array is set to logical `1` if both `A` and `B` contain a nonzero element at that same array location. Otherwise, the array element is set to `0`. For example,
  `4 & 0 = 0`
  `0 & 0 = 0`
  `4 & 3 = 1`
  `[1, 2; 1, 0] & [0, -1; 2, 3] = [0, 1; 1, 0]`
  `[1, 2; 1, 0] & [0, -1] = [0, 1; 0, 0]` (the second operand is expanded)
  `true & false = 0`
- `|` (`or`): `A | B` performs a logical `OR` of arrays `A` and `B` and returns an array containing elements set to either logical `1` (`true`) or logical `0` (`false`). An element of the output array is set to logical `1` if either `A` and `B` contain a nonzero element at that same array location. Otherwise, the array element is set to `0`. For example,
  `4 | 0 = 1`
  `0 | 0 = 0`
  `4 | 3 = 1`
  `[1, 2; 1, 0] | [0, -1; 2, 3] = [1, 1; 1, 1]`
  `[1, 2; 1, 0] | [0, -1] = [1, 1; 1, 1]` (the second operand is expanded)
  `true | false = 1`
- `~` (`not`): `~A` returns a logical array of the same size as `A`. The array contains logical `1` values where `A` is zero and logical `0` where `A` is nonzero. For example,
  `~[1, 2; 1, 0] = [0, 0; 0, 1]`

Exercise 2.3.1. Compute the following expressions and check your answer with MATLAB.

(1) `[1, 2; 3, 4] == 3`

(2) `[1, 2; 3, 4] >= [1, 2]`

(3) `[1, 2; 3, 4] ~= [1; 3]`

(4) `[1, 2; 3, 4] & [0, 5]`

(5) `[1, 2; 3, 4] | [0; 5]`

(6) `(3 > 2) & (1 >= 2)`

(7) `(3 > 2) | (1 >= 2)`

□

2.4 Selection and loop statements

Selection statements in MATLAB include `if` and `switch` statements. The syntax of `if` statement is

```
if conditional-expression
     statements
elseif conditional-expression (optional)
     statements
else (optional)
     statements
end
```

Consider the step function

$$y = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0. \end{cases} \tag{2.1}$$

The `if` statement for this function is

```
x = input('Enter a number:  ');
if x<0
     y = 0;
else
     y = 1;
end
disp(y)
```

Here, we have used `input` and `disp`. Check out their usage using `help`.

The syntax of `switch` statement is

```
switch switch_expression
     case case_expression
          statements
     case case_expression
          statements
     ...
```

```
    otherwise (optional)
        statements
end
```

Here is an example for `switch`.

```
grade = input('Enter a letter grade:  ');
switch (grade)
    case {'A', 'a'}
        disp('Excellent');
    case {'B','b'}
        disp('Very Good');
    case {'C','c'}
        disp('You Passed');
    case {'D','d'}
        disp('Close.  Try It Again');
    case {'F','f'}
        disp('Better to Try It Again');
    otherwise
        disp('Invalid Grade Entered');
end
fprintf(' Your grade is %s\n', grade)
```

Notice that the input should be a character or a string, such as `'a'` and `'abcd'`. Check out the usage of `fprintf`. Here, `%s` is the format specification for character arrays (`%d` for integers, `%e` and `%f` for real numbers) and `\n` means moving to a new line.

The MATLAB loop statements include the `for` and `while` statements. The syntax of the `for` statement is

```
for index = values
    statements
end
```

As an example, we want to compute the sum $\sum_{n=1}^{N} n$ for a given $N = 100$. The `for` statement is

```
N = 100;
sum = 0;
for n = 1:N
    sum = sum + n;
end
fprintf('sum = %d\n', sum);
```

Notice that `n = 1:N` is the short notation for `n = 1:1:N` with the middle '1' standing for increment 1. If we want to compute the sum of positive even integers less than or equal to $N = 100$, the code becomes

```
N = 100;
sum = 0;
for n = 2:2:N
     sum = sum + n;
end
fprintf('sum = %d\n', sum);
```

Notice that `2:2:N` is a vector of the form $[2, 4, ..., N]$ for even N or $[2, 4, ..., N-1]$ for odd N, and 2 at the middle represents the increment.

The syntax of the `while` statement is

```
initialization statement
while conditional_expression
     statements
     increment/decrement statement
end
```

The example of computing the sum $\sum_{n=1}^{N} n$ can be written into the `while` loop as

```
N = 100;
sum = 0;
n = 1;
while (n <= N)
     sum = sum + n;
     n = n + 1;
end
fprintf('sum = %d\n', sum);
```

Exercise 2.4.1. Write a short script for computing the factorial $N!$ for a given non-negative integer N using the `for` and `while` statements. Compare your results with those obtained with the MATLAB function `factorial`. Notice that you may need the `if` statement to test the situations $N = 0$ and $N > 0$. □

2.5 Functions

MATLAB provides numerous built-in functions including elementary functions in mathematics. In the meantime, it also allows the user to create several types of functions, including anonymous functions, local functions, nested functions, and private functions. We focus on

the first two that are more commonly used.

- **Anonymous Functions**: An anonymous function is a function that is not stored in a program file, but is associated with a variable whose data type is function handle. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement. These functions are similar to inline functions in C or C++. For instance,
  `f = @(x) sin(2*pi*x);` (try `f(1)`)
  `f = @(x, y) x^2 + y^4;` (try `f(1,0)`)
  `f = @(x) x.^2;` (try `f(1)` or `f([1,2])`)
  `f = @(x) [x(1)+x(2); x(1)*x(2)];` (try `f([1,2])`)

- **Local Functions**: MATLAB program files can contain code for more than one function. In a function file, the first function in the file is called the main function. This function is visible to functions in other files, or can be called from the command line. Additional functions within the file are called local functions or subfunctions, and they can occur in any order after the main function. Local functions are only visible to other functions in the same file.

We now discuss ***how to create functions in files***. First of all, files can be created and edited with the MATLAB Editor using the command `>> edit` *`filename`* or by clicking the `New Script`, `New`, or `Open` tab. Files can also be created and edited using other text editors installed on the computer. The syntax of functions is

```
function OutputVariables = FunctionName( InputVariables )
    statements
end
```

If this is the main function in the file, the name of the file should match the name of the function. The extension of the file should be `.m`, i.e., the name of the file should look like `FunctionName.m`. Moreover, it is not always required to have the `end` statement in the file. Nevertheless, it is good programming practice to have it in the file so it is clearer where the function ends.

As a first example, we consider the step function example in §2.4. Create a file called `step_fun.m` in the current folder and enter the following statements,

```
function y = step_fun( x )
    if x<0
        y = 0;
    else
        y = 1;
    end
end
```

On the command line, type `y = step_fun(10)` and see what you get.

The second example is `sign_fun.m` which does not have output variables.

```
function sign_fun( x )
    if x<0
        disp('x is negative');
    else
        disp('x is non-negative');
    end
end
```

The third example, `prod_sum.m`, has two input variables and two output variables.

```
function [prod, sum] = prod_sum( x, y )
    prod = x*y;
    sum = x + y;
end
```

To conclude this section, we remark that a file can have multiple functions. The input variables can be any type of data, including functions.

Exercise 2.5.1. Write a function for computing the sum $\sum_{n=1}^{N} n$. Create a file `sum_fun.m` with the first line as `function sum = sum_fun( N )`. □

2.6 How to plot data in MATLAB

Please try all these commands and see what you get on your computer.

```
>> t = 0:4*pi/125:4*pi      This creates a vector named t with entries
                            [0,4*pi/125,8*pi/125,...,4*pi]
                            Another way is to use linspace(0,4*pi,126)
```

Notice that this entire list of numbers was displayed on the screen, which is not generally what you want. To suppress the display, end the line with a semi-colon ";".

```
>> x = sin(t);              This creates a vector named x with entries
                            [sin(0),sin(4*pi/125),...,sin(4*pi)]
>> plot(x)                  Notice the units in horizontal axis
>> plot(t,x)                What is the difference?
>> y = sin(t+.25);          Phase shift
>> z = sin(t+.5);
>> hold on                  The next graph drawn will not erase
                            the present one.
>> plot(t,y,'--r')          r for color red
>> plot(t,z,'.b')           b for blue
```

```
>> title('sine function with phase shift')
>> xlabel('t')
>> ylabel('sin(t+)')
>> hold off
>> plot(x,z)                    What is this?
>> subplot(3,1,1), plot(t,x)
>> subplot(3,1,2), plot(t,y)
>> subplot(3,1,3), plot(t,z)    subplot(m,n,p) creates an m-by-n array
                                of plots on a single screen, and refers
                                to the pth one, counting from upper left
```

To save a figure, you can go to the figure window and choose `file` and `save as`. You can also use line commands.

```
>> print('a.eps', '-depsc')    Command to save the current image as a
                                color eps file with the name a.eps.
>> print('b.pdf', '-dpdf')     Command to save the current image as a
                                pdf file with the name b.pdf.
```

You can go to a specific figure (say figure (2)):

```
>> figure (2)
>> clf                          clear all figures in figure (2)
```

2.7 A few functions useful to know

In the previous sections, we have used `help`, `disp`, and `fprintf` several times. Once again, check out their usage. In this section, we study two more functions, `find` and `accumarray`, in detail.

The function `find` finds indices and values of nonzero elements.

- `find(X)` returns a vector containing the linear indices of nonzero elements in array `X`. For example, `find([1, 2; 0, 3])` returns `[1; 3; 4]`.
- `[row, col] = find(X)` returns vectors `row` and `col` containing the row and column indices of nonzero elements in `X`, respectively. For the previous example, `[r, c] = find([1, 2; 0, 3])` returns `r = [1; 1; 2]` and `c = [1; 2; 2]`.
- Another use of `find` is to find indices of elements satisfying certain conditions. For example, `find([1, 2; 0, 3] >= 2)` returns `[3; 4]`.
- ***Logical indexing:*** Let `A = [1, 2; 0, 3]`. Then, `B = A(A>=2)` returns `B = [2; 3]`. The condition `A>=2` in the index is called logical indexing. It functions the same as `find(A>=2)`. Therefore, `A(A>=2)` is equal to combined `I = find(A>=2)` and `B = A(I)`. Try `C = A(A(:,1)>=1,:)`. The answer is `C = [1, 2]`.

The second function is `accumarray` which is used to construct array with accumulation. This function is particularly useful for assembling matrices in finite element computation. The simplest use of this function is

```
A = accumarray(subs, val)
```

where `val` is a column vector of values, `subs` is a column vector consisting of (possibly repeated) indices of `val`, and both `subs` and `val` have the same length. The outcome `A` is a vector of length `max(subs)`, with its elements defined as `A(i)=sum(val(subs(:)==i))` for `i = 1, ..., max(subs)`. (The logical indexing `subs(:)==i` in subscripts is equivalent to `find(subs(:)==i)`, i.e., the indices of the elements of `subs` with their values equal to `i`.) Elements of `A` whose subscripts do not appear in `subs` (i.e., `find(subs(:)==i)` is an empty set) are equal to `0`.

We now consider an example.

```
>> val = [0.1; 0.2; 0.3; 0.4; 0.5];
>> subs = [1; 4; 1; 3; 4];
>> A = accumarray(subs, val);
```

This function can be understood in two different ways. For the first, notice that the assignment without accumulation is: `A(subs(i)) = val(i)`, `i = 1, ..., 5`, i.e.,

$$\texttt{A}\left(\begin{bmatrix}1\\4\\1\\3\\4\end{bmatrix}\right) = \begin{bmatrix}0.1\\0.2\\0.3\\0.4\\0.5\end{bmatrix},$$

or

$$A(1) = 0.1, \quad A(4) = 0.2, \quad A(1) = 0.3, \quad A(3) = 0.4, \quad A(4) = 0.5.$$

Then the repeated entries are accumulated. This gives

$$A(1) = 0.1 + 0.3 = 0.4, \quad A(3) = 0.4, \quad A(4) = 0.2 + 0.5 = 0.7.$$

The next step is to set $A(2) = 0$. Thus, the outcome is `A = [0.4; 0; 0.4; 0.7]`.

The second way is based on `A(i)=sum(val(subs(:)==i))`. First, the length of `A` is `max(subs) = 4`. For `A(1)`, `find(subs(:)==1) = [1; 3]`. Thus, `A(1) = val(1) + val(3) = 0.4`. For `A(2)`, `find(subs(:)==2) = empty` and thus `A(2) = 0`. Repeating this, we can find that `A(3) = 0.4` and `A(4) = 0.7`.

Often it is desired that `A` has the same length as `subs` and `val`. This can be done by augmenting `subs` and `val`:

```
>> val = [0.1; 0.2; 0.3; 0.4; 0.5];
>> subs = [1; 4; 1; 3; 4];
>> A = accumarray([subs; length(val)], [val; 0]);
```

Then we have `A = [0.4; 0; 0.4; 0.7; 0]`.

To conclude this chapter, we emphasize that you can get help with `help` command within MATLAB or by searching online with words like "MATLAB plot". It is also useful to check `See also` typically at the end of help menu. Furthermore, the taps `Analyze Code` and `Run and Time` on the command bar of MATLAB are very useful and worth trying at some point.

Chapter 3

Lab 1: Convert Decimal Numbers into Binary Form

In this lab, we consider to write a function (in file) to convert decimal numbers into binary form using Algorithm 1.1.1. We shall go through the steps of problem describing, planning, coding, testing, and reporting.

3.1 Problem description

It is important to have a good understanding of the underlying problem and the goals. It is important to gather all mathematical formulas before we code.

The goal of this lab is to write a function (in file) to convert any decimal number x into binary form using Algorithm 1.1.1. When x is not zero, it can be written in the binary form as

$$x = \pm(0.a_1a_2\cdots a_n\cdots)_2 \times 2^m, \tag{3.1}$$

where the sign (denoted by s), m, and a_1 $(= 1)$, a_2, ... are to be determined. Since zero cannot be cast in the above form, we will need a special treatment for zero. Otherwise, m satisfies

$$2^{m-1} \le |x| < 2^m. \tag{3.2}$$

Taking natural logarithm on all terms, we get

$$m - 1 \le \frac{\ln|x|}{\ln 2} < m,$$

which implies that $m - 1$ is the integer part of $\ln|x|/\ln 2$.

For convenience, we copy Algorithm 1.1.1 into Algorithm 3.1.1.

3.2 Planning

Having had a good understanding of the underlying problem and gathered all of the mathematical formulas, we can now do the planning.

Algorithm 3.1.1 Convert decimal numbers into binary form.

1. Compute m and let $x_0 = |x| \times 2^{-m}$.
2. For $n = 1, 2, \cdots$ do
   (a). If $2x_{n-1} \geq 1$, set $a_n = 1$; otherwise, $a_n = 0$.
   (b). Compute $x_n = 2x_{n-1} - a_n$.
   (c). If $x_n = 0$, stop the computation.

The first thing is to determine the ***input and output variables***. An obvious input variable is x. For output variables, basically we want to know the right-hand side of (3.1). Then we need to decide if we want a printout of the form or just the values of s, m, and a_1, a_2, ... At this point, we may realize that numbers can have long or a infinite number of digits. We may ask the user to provide a maximum (denoted by N) on the number of the output digits. Thus, we modify the input variables to be x and N. Going back to the output variables, we consider here to output both the form and the values of s, m, and a_1, ..., a_n (with $n <= N$). To summarize, we have

Input variables:	x and N
Output variables:	printout of form (3.1)
	s, m, and a_1, ..., a_n, with $n <= N$

The next step in the planning is to decide the name and the first line of the function. We call it `Decimal2Binary` and define the first line of the function as

```
function [s, m, a] = Decimal2Binary(x, N)
```

We now add the first line to Algorithm 3.1.1 and make a "flowchart" (a more detailed algorithm) into Algorithm 3.2.1. One can see that several changes were made here:

- Added the first line of the function with the input and output variables,
- Changed the `for` loop with an upper limit,
- Added Step 3 (for printout),
- Changed "stop the computation" to "break the loop".

3.3 Coding

At this step we begin to fill in technical detail. To start with, it is a good idea to keep in mind that a well written code should allow other people to use it. To this end, we should

Algorithm 3.2.1 Convert decimal numbers into binary form (with more detail).

```
function [s, m, a] = Decimal2Binary(x, N)
```

1. Compute m, s, and let $x_0 = |x| \times 2^{-m}$.
2. For $n = 1 : N$ do
   (a). If $2x_{n-1} \geq 1$, set $a_n = 1$; otherwise, $a_n = 0$.
   (b). Compute $x_n = 2x_{n-1} - a_n$.
   (c). If $x_n = 0$, break the loop.
3. Print out the form of (3.1).

try to make the code (i) easy to read, (ii) easy to use, (iii) easy to modify, and (iv) efficient (in terms of memory use and CPU time). Tips for good programming include

- Try to use the same names for variables and constants as in the mathematical formulas,
- Give brief and necessary comments,
- Use proper indents,
- Avoid unnecessary storage.

Following these tips, we find that there is no need to define x_0, x_1, ..., x_n as a vector in Algorithm 3.2.1 since x_{n-1} is not needed after x_n is calculated. Thus, all of x_0, x_1, ..., x_n can be replaced by a single variable, say, x_0. On the other hand, we do need to use a vector to save a_1, ..., a_n. With this change, we rewrite Algorithm 3.2.1 into Algorithm 3.3.1, where we have used `:=` for computer assignments to avoid confusion with mathematical assignments.

Algorithm 3.3.1 Convert decimal numbers into binary form (with more detail).

```
function [s, m, a] = Decimal2Binary(x, N)
```

1. Compute m, s, and let $x_0 := |x| \times 2^{-m}$.
2. For $n = 1 : N$ do
   (a). If $2x_0 \geq 1$, set $a_n = 1$; otherwise, $a_n = 0$.
   (b). Compute $x_0 := 2x_0 - a_n$.
   (c). If $x_0 == 0$, break the loop.
3. Print out the form of (3.1).

At this point, we are ready to code. We strongly suggest that the reader try to code according to Algorithm 3.3.1 and then compare your program with the one listed in Algorithm 3.3.2. Recall that the file should be named as `Decimal2Binary.m`. You may use functions `sign`, `floor`, and `fprintf`. The code is explained line by line in the following.

- Line 2: MATLAB treats all the information after `%` on a line as a comment.
- Line 4: The MATLAB built-in function `sign` is used to compute the sign of x (`+1` for positive x, `-1` for negative x, and `0` for $x = 0$).
- Line 5-10: For the case $x = 0$, we simply set $m = 0$ and $a = []$ (empty), display the message `x = 0` on the screen using `fprintf`, skip the rest of the statements, and `return` to where this function is called.
- Line 11: Function `floor(A)` rounds each element of `A` to the nearest integer less than or equal to that element.
- Line 13: This preallocates a 1-by-`N` block of memory for vector `a` and initializes it to be zero. There is no need to repeatedly reallocate memory when its size grows but stays no larger than `N`. The code runs much faster in this way.
- Line 14-25: This is the main loop.
- Line 23: This terminates the execution of the `for` loop. Notice that the index `n` keeps the current value.
- Line 26: Whether the loop is terminated when `n` reaches the limit `N` or through `break` when `x0 == 0`, `n` records the actual number of digits in vector `a`. Since `n` can be smaller than `N`, we need to remove the possible extra digits in `a`.
- Line 29-37: Print the result in the binary form (3.1). `%e` and `%d` specify the format for real numbers (in scientific form) and integers, respectively. `%1d` specifies the integer format with one-digit width. Notice that the displays from Lines 30/32, 35, and 37 are on the same line on screen.

3.4 Testing

We would like to test the code to see if it is correct. MATLAB provides the code analyzer (under the tap `Analyze Code` on the command bar) and the profiler (under the tap `Run and Time`) which are very useful. A simpler way is to try several test problems for which we know the solutions. Here are a few examples.

- `[s, m, a] = Decimal2Binary(0, 20)` leads to `x = 0`, `s = 0`, `m = 0`, and `a = []`. Obviously, that is what we wanted.

- `[s, m, a] = Decimal2Binary(11, 20)` leads to `x = 1.100000e+01 = + (0.1011)_2 x 2^ 4`, `s = 1`, `m = 4`, and `a = 1011`.
- `[s, m, a] = Decimal2Binary(-1/3, 20)` leads to

  ```
  x = -3.333333e-01 = - (0.10101010101010101010)_2 x 2^ -1
  ```

  and `s = -1`, `m = -1`, and `a = 10101010101010101010`.

3.5 Reporting

There is no unique way how to write or what to include in a report. Generally speaking, a reasonable report should involve the problem description, the numerical method, discussion and analysis of obtained numerical results, conclusions, and a copy of the code or a brief description of the code. It is a good idea to present numerical results in tables and/or figures. One may wonder what results should be included. If there are given requirements, we should present required numerics. If there are no such requirements or we want to provide more results than required, we can consider to provide results that support a point we would like to make or demonstrate an interesting phenomenon we observe.

It is common that we do not know what to say or do not have much to say on the tables and figures we provide. Well, maybe we should ask why we provide those tables and figures in the first place. We may want to tell people what the tables/figures are, what they show, and what is new in them. Do not think everything is trivial. Maybe it is trivial to you, but it is not necessarily trivial for other people.

Algorithm 3.3.2 Convert decimal numbers into binary form: the code. The numbering in the first column is for easy reference; it is not part of the code.

```
 1: function [s, m, a] = Decimal2Binary(x, N)
 2: % this function converts decimal number x into binary form.
 3:
 4:     s = sign(x);
 5:     if (s == 0)
 6:         m = 0;
 7:         a = [];
 8:         fprintf('x = 0\n');
 9:         return;
10:     end
11:     m = 1 + floor(log(abs(x))/log(2));
12:     x0 = abs(x)*2^(-m);
13:     a = zeros(1,N);
14:     for n = 1:N
15:         x0 = 2*x0;
16:         if (x0 >= 1)
17:             a(n) = 1;
18:         else
19:             a(n) = 0;
20:         end
21:         x0 = x0 - a(n);
22:         if (x0 == 0)
23:             break;
24:         end
25:     end
26:     a = a(1:n);
27:
28:     % for printfout of binary form
29:     if (s > 0)
30:         fprintf('x = %e = + (0.', x);
31:     else
32:         fprintf('x = %e = - (0.', x);
33:     end
34:     for i=1:n
35:         fprintf('%1d', a(i));
36:     end
37:     fprintf(')_2 x 2^%d\n', m);
38:
39: end % end of Decimal2Binary()
```

Chapter 4

Lab 2: Piecewise Linear Interpolation for a Given Data Set

Interpolation for given data sets is a very important subject in scientific computing. MATLAB built-in functions in this subject include `interp1`, `interp2`, `interp3`, `interpn`, `griddedInterpolant`, and `scatteredInterpolant`.

4.1 Problem description

In this lab we consider a simple scenario – the problem of using piecewise linear polynomials to interpolate a given data set. Denote the data set by

x_1	x_2	$\cdots$	x_n
y_1	y_2	$\cdots$	y_n

For simplicity, we assume that the points x_1, ..., x_n are distinct and has been sorted in the ascending order, i.e.,

$$x_1 < x_2 < \cdots < x_n.$$

We also image that the set defines a function $y = f(x)$ with $y_i = f(x_i)$, $i = 1, ..., n$. We do not need to know what f is. It is easier to refer to the data set when we have a function name associated with it.

The task of this lab is to write a function (in file) to perform linear interpolation for the given data set. The piecewise linear polynomial is defined as

$$p(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i} y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1}, \quad x \in [x_i, x_{i+1}], \quad i = 1, ..., n-1. \tag{4.1}$$

We want to write a function to take the data set and a number x, check if x is in $[x_1, x_n]$, and if so, compute the value $p(x)$.

4.2 Planning

At this step we would like to figure out the input/output variables and the flowchart/algorithm.

First of all, one may realize that we have an issue with symbols. We may want to use `x` for the vector $(x_1, ..., x_n)$ but we may also want to use `x` for the point we want to find the interpolation value. To avoid this, we decide to use `X` and `Y` for $(x_1, ..., x_n)$ and $(y_1, ..., y_n)$, respectively. Then the input/output variables are

Input variables: `X`, `Y`, x
Output variables: `p` $(= p(x))$

Next, we call this function `LinearInterpolation1D`. Thus, the first line of the code is

```
function p = LinearInterpolation1D(x, X, Y)
```

The algorithm (or flowchart) is given in the following algorithm.

Algorithm 4.2.1 Perform piecewise linear interpolation for a given data set.

`function p = LinearInterpolation1D(x, X, Y)`

1. Check if x is in $[X_1, X_n]$. If not, display the error message and return.
2. Find the index i such that $x \in [X_i, X_{i+1}]$.
3. Compute
$$p(x) = \frac{X_{i+1} - x}{X_{i+1} - X_i} Y_i + \frac{x - X_i}{X_{i+1} - X_i} Y_{i+1}$$

4.3 Coding

The coding for Algorithm 4.2.1 is relatively straightforward. Give a try to write the code before you look at the code in Algorithm 4.3.1. You may need to use MATLAB built-in function `find`.

Here are the comments for the code.

- Line 4: We use function `find` to execute Steps 1 and 2 in Algorithm 4.2.1. It returns an empty set when the condition is not met.
- Line 6: Function `error` displays the message and stops the running of the code.
- Line 8: Function `find` may return more than one indices. We choose the smallest one.

Algorithm 4.3.1 Perform piecewise linear interpolation for a given data set: the code.

```
function p = LinearInterpolation1D(x, X, Y)
% this function performs piecewise linear interpolation for (X,Y).

    I = find((X(1:end-1) <= x) & (x <= X(2:end)));
    if isempty(I)
        error('x is outside the domain [X(1), X(end)]');
    end
    i = min(I);
    p = ((X(i+1)-x)*Y(i)+(x-X(i))*Y(i+1))/(X(i+1)-X(i));

end % end of LinearInterpolation1D()
```

4.4 Testing

To test the code, we recall that piecewise linear interpolation is exact for linear polynomials. This means that if we choose $f(x) = x$ and generate values $Y = f(X)$, then we can expect that the error, defined as $|p(x) - f(x)|$, is zero for any x inside the domain. Thus, we can use this fact to test the code. An example of the script is

```
>> f = @(x) x;
>> X = linspace(0,1,10);
>> Y = f(X);
>> p = LinearInterpolation1D(0.5,X,Y);
>> err = abs(p-f(0.5));
```

The result is `err = 0`, which is consistent with our expectation.

Next, it is known in scientific computing that linear interpolation has second-order convergence for smooth functions. This means that the error decreases like $1/n^2$ as n (the number of data points) increases. To see this, we consider $f(x) = \sin(\pi x)$ and define `X` as `linspace(0,1,n)` with `n` = 10, 20, 40, 80 (and thus 0.5 is not a data point). An example of the script is

```
>> f = @(x) sin(pi*x);
>> X = linspace(0,1,10);
>> Y = f(X);
>> p = LinearInterpolation1D(0.5,X,Y);
>> err = abs(p-f(0.5));
```

The results are listed in the following table.

n	10	20	40	80
$\lvert p(0.5) - f(0.5)\rvert$	1.52e-2	3.40e-3	8.11e-4	1.98e-4
ratio		4.5	4.2	4.1

From the table, one can see that as n is doubled, the error decreases by a factor of about 4, confirming that the convergence order is second.

The above testing suggests that the code is correct.

Chapter 5

Lab 3: The Method of Least Squares

The method of least squares is one of the most important methods in scientific computing. It is a standard approach in regression analysis to approximate the solution of overdetermined systems and has important applications in data sciences. For example, MATLAB's Curve Fitting Toolbox uses the method of least squares when fitting data. After a parametric model that relates the response data to the predictor data has been chosen, the method of least squares minimizes the summed square of residuals to obtain the coefficient (parameter) estimates.

5.1 Problem description

In this lab we consider the simplest least squares fitting – the linear least squares fitting. We assume that we are given a set of observation data

x_1	x_2	$\cdots$	x_n
$\hat{y}_1$	$\hat{y}_2$	$\cdots$	$\hat{y}_n$

We want to fit a linear model to the data; see Fig. 5.1. A linear model is a linear function of x, i.e.,

$$y = p_1 + p_2 x, \tag{5.1}$$

where p_1 and p_2 are the parameters to be determined. The residues are defined as the difference between the observation data and the predictor data (by the model), that is,

$$r_i = \hat{y}_i - y_i = \hat{y}_i - (p_1 + p_2 x_i), \quad i = 1, ..., n.$$

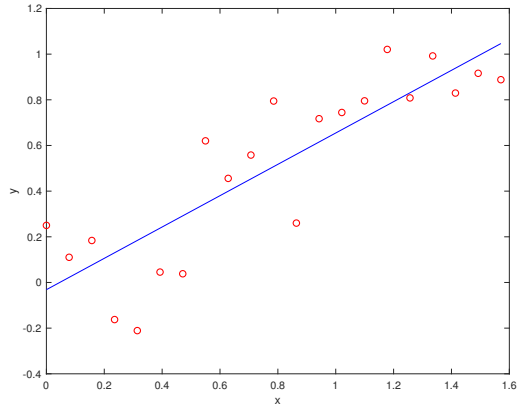

Figure 5.1: Linear least squares fitting for a given data.

The method of least squares is to minimize the summed square of residues,

$$\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \Big[\hat{y}_i \ - (p_1 + p_2 x_i)\Big]^2 \equiv g(p_1, p_2),$$

to find the estimates of p_1 and p_2. By setting the partial derivatives of the sum with respect to p_1 and p_2 to be zero, we obtain

$$\frac{\partial g}{\partial p_1} = 2\sum_{i=1}^{n} \Big[\hat{y}_i \ - (p_1 + p_2 x_i)\Big] \cdot 1 = 0,$$
$$\frac{\partial g}{\partial p_2} = 2\sum_{i=1}^{n} \Big[\hat{y}_i \ - (p_1 + p_2 x_i)\Big] \cdot (-x_i) = 0.$$

After some algebraic manipulation, we get

$$\left(\sum_{i=1}^{n} 1 \cdot 1\right) p_1 + \left(\sum_{i=1}^{n} x_i \cdot 1\right) p_2 = \left(\sum_{i=1}^{n} \hat{y}_i \cdot 1\right),$$

$$\left(\sum_{i=1}^{n} 1 \cdot x_i\right) p_1 + \left(\sum_{i=1}^{n} x_i \cdot x_i\right) p_2 = \left(\sum_{i=1}^{n} \hat{y}_i \cdot x_i\right).$$

If we define

$$A = \begin{bmatrix} 1 & x_1 \\ \vdots & \\ 1 & x_n \end{bmatrix}, \quad Y = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_n \end{bmatrix}, \quad p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix},$$

we can rewrite the above system into

$$A^T A p = A^T b, \tag{5.2}$$

where A^T is the transpose of A. The estimates to the parameters can be obtained by solving this system.

5.2 Planning

Once again, we need to consider the input/output variables, the first line of the function, and the chart of the algorithm. We put these together in Algorithm 5.2.1.

Algorithm 5.2.1 The linear least squares fitting.

```
function p = LinearLeastSquaresFitting(X, Y)
```

1. Determine the length of X and preallocate A
2. Check if X is a row/column vector and form A, $B := A^T A$.
3. Check if Y is a row/column vector and form $b := A^T Y$.
4. Solver $Bp = b$.

5.3 Coding

The reader is strongly encouraged to code according to Algorithm 5.2.1. An example of the code is listed in Algorithm 5.3.1.

Algorithm 5.3.1 The linear least squares fitting: the code.

```
1: function p = LinearLeastSquaresFitting(X, Y)
2: % this computes the linear least squares fitting to data (X, Y).
3:
4:     A = ones(length(X),2);
5:     if isrow(X)
6:         A(:,2) = X';
7:     else
8:         A(:,2) = X;
9:     end
10:    B = A'*A;
11:    if isrow(Y)
12:        b = A'*Y';
13:    else
14:        b = A'*Y;
15:    end
16:    p = B\b;
17:
18:    end % end of LinearLeastSquaresFitting()
```

5.4 Testing

The following script gives a test for the code,

```
X = linspace(0,pi/2,21)';
Y = sin(X);
Y = Y + randn(size(Y))*0.2; (adding the noise)
p = LinearLeastSquaresFitting(X,Y); (calling the function)
y = p(1) + p(2)*X;
plot(X,Y,'or',X,y,'-b')
```

A plot is shown in Fig. 5.1.

Chapter 6

Lab 4: The Composite Gauss-Legendre Quadrature

MATLAB built-in functions for numerical integration include `integral`, `integral2`, `integral3`, and `trapz`.

6.1 Problem description

In this lab we consider to write a function (in file) for numerically integrating integrals in the form $\int_a^b f(x)dx$ using the composite two-point Gauss-Legendre quadrature rule. The two-point Gauss-Legendre quadrature rule reads as

$$\int_{-1}^{1} g(s)ds \approx g(-\frac{1}{\sqrt{3}}) + g(\frac{1}{\sqrt{3}}), \tag{6.1}$$

where $-1/\sqrt{3}$ and $1/\sqrt{3}$ are the roots of the Legendre polynomial of degree two. It is known that the rule is exact for polynomials of degree 3 or less. This can be readily verified by comparing the values of both sides with $g(s) = 1,\ s,\ s^2,\ s^3$, and s^4.

To develop a composite rule for the integral $\int_a^b f(x)dx$, we divide the interval (a, b) into $n-1$ subintervals of same length, i.e.,

$$x_1 = a < x_2 < \cdots < x_n = b \quad \text{with} \quad x_i = a + (i-1)h, \quad i = 1, ..., n, \quad h = \frac{b-a}{n-1}.$$

The integral can be broken up into

$$\int_a^b f(x)dx = \int_{x_1}^{x_2} f(x)dx + \int_{x_2}^{x_3} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx.$$

To apply the Gauss-Legendre quadrature rule to each of the integrals in the above equation, we use the coordinate transformation $(x = x_i + (s+1)h/2)$ and get

$$\int_{x_i}^{x_{i+1}} f(x)dx = \frac{h}{2} \int_{-1}^{1} f\left(x_i + \frac{1}{2}(s+1)h\right) ds.$$

Applying (6.1) to this integral with

$$g(s) = f\left(x_i + \frac{1}{2}(s+1)h\right)$$

and combining this with the previous equation, we have

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n-1}\left[f\left(x_i + \frac{1}{2}(1-\frac{1}{\sqrt{3}})h\right) + f\left(x_i + \frac{1}{2}(1+\frac{1}{\sqrt{3}})h\right)\right]. \tag{6.2}$$

For easy reference, we denote

$$I(f) = \int_a^b f(x)dx,$$
$$I_h(f) = \frac{h}{2}\sum_{i=1}^{n-1}\left[f\left(x_i + \frac{1}{2}(1-\frac{1}{\sqrt{3}})h\right) + f\left(x_i + \frac{1}{2}(1+\frac{1}{\sqrt{3}})h\right)\right].$$

We define the error as

$$E_h(f) = I(f) - I_h(f). \tag{6.3}$$

We now use a "guessing" method to find the expression for $E_h(f)$. We start with the error term for the two-point Gauss-Legendre quadrature rule (6.1). Define

$$E_G(g) = \int_{-1}^{1} g(s)ds - \left[g(-\frac{1}{\sqrt{3}}) + g(\frac{1}{\sqrt{3}})\right]. \tag{6.4}$$

Recall that the rule is exact for polynomials of degree 3 and less, i.e.,

$$E_G(1) = 0, \quad E_G(s) = 0, \quad E_G(s^2) = 0, \quad E_G(s^3) = 0. \tag{6.5}$$

Suggested from the remainder term in Taylor's theorem, we guess that $E_G(g)$ has a form

$$E_G(g) = C\frac{d^4g}{ds^4}(\hat{s}), \tag{6.6}$$

where C is a constant independent of g and $\hat{s}$ is a point in $(-1, 1)$. Here we have assumed that the fourth-order derivative of g is continuous on $(-1, 1)$. Notice that $E_G(g)$ in the form (6.6) satisfies (6.5). Moreover, we can find C by taking $g = s^4$. Indeed, from (6.6) we have

$$E_G(s^4) = C\,4!.$$

On the other hand, from (6.4) we get

$$E_G(s^4) = \int_{-1}^{1} s^4 ds - \left[(-\frac{1}{\sqrt{3}})^4 + (\frac{1}{\sqrt{3}})^4\right] = \frac{2}{5} - \frac{2}{9} = \frac{8}{45}.$$

Combining the above results, we obtain $C = 1/135$ and thus,

$$E_G(g) = \frac{1}{135}\frac{d^4g}{ds^4}(\hat{s}).$$

Substituting this into (6.4) yields

$$\int_{-1}^{1} g(s)ds = \left[g(-\frac{1}{\sqrt{3}}) + g(\frac{1}{\sqrt{3}})\right] + \frac{1}{135}\frac{d^4g}{ds^4}(\hat{s}). \tag{6.7}$$

We now are ready to find $E_h(f)$. Recall that in the derivation of the composite rule, we applied the two-point Gauss-Legendre quadrature rule to

$$\int_{x_i}^{x_{i+1}} f(x)dx = \frac{h}{2}\int_{-1}^{1} f\left(x_i + \frac{1}{2}(s+1)h\right) ds$$

with

$$g(s) = f\left(x_i + \frac{1}{2}(s+1)h\right).$$

If we use (6.7) instead and notice that

$$\frac{d^4g}{ds^4}(\hat{s}) = \frac{h^4}{16}\frac{d^4f}{dx^4}(\hat{x}_i), \quad \hat{x}_i = x_i + \frac{1}{2}(\hat{s}+1)h,$$

we have

$$\int_a^b f(x)dx = \frac{h}{2}\sum_{i=1}^{n-1}\left[f\left(x_i + \frac{1}{2}(1-\frac{1}{\sqrt{3}})h\right) + f\left(x_i + \frac{1}{2}(1+\frac{1}{\sqrt{3}})h\right)\right] + \frac{h^5}{4320}\sum_{i=1}^{n-1}\frac{d^4f}{dx^4}(\hat{x}_i),$$

which gives

$$E_h(f) = \frac{h^5}{4320}\sum_{i=1}^{n-1}\frac{d^4f}{dx^4}(\hat{x}_i).$$

If we assume that the fourth-order derivative of f is continuous on (a,b), then there exists a number $\hat{x} \in (a,b)$ such that

$$\sum_{i=1}^{n-1}\frac{d^4f}{dx^4}(\hat{x}_i) = (n-1)\frac{d^4f}{dx^4}(\hat{x}).$$

Using this, we can simplify $E_h(f)$ into

$$E_h(f) = \frac{h^5(n-1)}{4320}\frac{d^4f}{dx^4}(\hat{x}).$$

From $h = (b-a)/(n-1)$, we finally get

$$E_h(f) = \frac{(b-a)^5}{4320(n-1)^4}\frac{d^4f}{dx^4}(\hat{x}). \tag{6.8}$$

This indicates that the error is reduced roughly by a factor of $2^4 = 16$ when n is doubled, i.e., the method has fourth-order convergence.

6.2 Planning

We start with the input/output variables. We will need to input the problem parameters: f (the function), a, and b. We also need n for the quadrature rule. For the output, we want the approximation: I_h. Thus,

Input variables: a, b, f, n
Output variables: I_h

Next, we decide the function name and the first line:

```
function Ih = GaussLegendre2P(f, a, b, n)
```

Finally, we decide the algorithm/flowchart. After several rounds of modification on scratch paper, we have the final version of the algorithm in Algorithm 6.2.1.

Algorithm 6.2.1 The composite two-point Gauss-Legendre quadrature.
`function Ih = GaussLegendre2P(f, a, b, n)`

1. Set $h = (b-a)/(n-1)$, $s_1 = (1 - 1/\sqrt{3})/2$, $s_2 = (1 + 1/\sqrt{3})/2$
2. Initialization: Set $x := a$ and $Ih := 0$
3. for $i = 1 : (n-1)$
   (a). $Ih := Ih + f(x + s_1 * h) + f(x + s_2 * h)$
   (b). $x := x + h$
4. $Ih := Ih * h/2$

6.3 Coding

The coding of Algorithm 6.2.1 is relatively straightforward. Once again, we recommend that the reader give it a try first. An example is given in Algorithm 6.3.1, where `tic` and `toc` are used to show the CPU time used in the computation and the function `f` is used an argument for the function `GaussLegendre2P`.

We note that the code in Algorithm 6.3.1 is loop-based and scalar-oriented. MATLAB is optimized for operations involving matrices and vectors and this runs much faster for vectorized code. To show this, we give a vectorized version of Algorithm 6.3.1 in Algorithm 6.3.2. One may see that there is no loop in the code. Line 13 computes all of the points $x_1, ...x_n$ and saves it in the vector `x`. Line 14 executes several tasks. For example, it computes the function values at the points $(x_1 + s_1 * h, ..., x_{n-1} + s_1 * h)$ and $(x_1 + s_2 * h, ..., x_{n-1} + s_2 * h)$ in vectorization and sums the values. This requires the function be defined in vectorization

Algorithm 6.3.1 The composite two-point Gauss-Legendre quadrature rule: the code.

```
function Ih = GaussLegendre2P(f, a, b, n)
% this function performs the composite 2-point Gauss-Lagendre rule.
% note that the function should be defined in the form y = f(x)

tic
    h = (b-a)/(n-1);
    s1 = 0.5*(1-1/sqrt(3));
    s2 = 0.5*(1+1/sqrt(3));
    x = a;
    Ih = 0;
    for i = 1:(n-1)
        Ih = Ih + f(x+s1*h) + f(x+s2*h);
        x = x + h;
    end
    Ih = Ih*h*0.5;
toc

end % end of GaussLegendre2P()
```

form (using array operations). For example,

```
Ih = GaussLegendre2Pvec(@(x) x.^3, 0, 1, 100)
```

for integral $\int_0^1 x^3 dx$.

6.4 Testing

We use two examples to test the code. The first is $\int_0^1 x^3 dx = 1/4$ for which the quadrature rule should give the exam value. The call for this example is:

```
Ih = GaussLegendre2P(@(x) x^3, 0, 1, 10)
```

which does give 0.25.

The another example is show the convergence order. We choose $\int_0^\pi \sin(x) dx = 2$. An example call is

```
Ih = GaussLegendre2P(@(x) sin(x), 0, pi, 10)
```

The results are given in the following table. We can see that the error reduction ratio

Algorithm 6.3.2 The composite two-point Gauss-Legendre quadrature rule: the vectorized code.

```
function Ih = GaussLegendre2Pvec(f, a, b, n)
%
% this is the vector version of GaussLegendre2P().
%
% this function performs the composite 2-point Gauss-Lagendre rule.
% note that the function should be defined in the form y = f(x)
% and using array operations.

tic
    h = (b-a)/(n-1);
    s1 = 0.5*(1-1/sqrt(3));
    s2 = 0.5*(1+1/sqrt(3));
    x = linspace(a,b,n)';
    Ih = sum(f(x(1:end-1) + s1*h)) + sum(f(x(1:end-1) + s2*h));
    Ih = Ih*h*0.5;
toc

end % end of GaussLegendre2Pvec()
```

is close to 16, which is consistent with the method's fourth-order convergence.

n	10	20	40	80	160
$\|E_h\|$	6.90e-6	3.46e-7	1.95e-8	1.15e-9	7.06e-11
ratio		19.9	17.7	17.0	16.3

6.5 The composite Simpson's rule

In this section we record the composite Simpson's rule. The interested reader can use it as a practice problem for programming with loops or in vectorization.

The rule reads as

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{3}\left[f(x_1)+4f(x_2)+2f(x_3)+4f(x_4)+2f(x_5)+\cdots+4f(x_{n-1})+f(x_n)\right]\\ &= \frac{2h}{3}\left[f(x_1)+2f(x_2)+f(x_3)+2f(x_4)+f(x_5)+\cdots+2f(x_{n-1})+f(x_n)\right]\\ &\quad -\frac{h}{3}\left[f(x_1)+f(x_n)\right], \end{aligned} \tag{6.9}$$

where $n \geq 5$ is an odd integer, $h = (b-a)/(n-1)$, and $x_i = a + (i-1)h$ $(i = 1, ..., n)$. The

error is given by

$$E_h(f) = -\frac{(b-a)^5}{180(n-1)^4}\frac{d^4 f}{dx^4}(\hat{x}),$$

where $\hat{x}$ is a number in (a, b). From the above error we can see that the method has fourth-order convergence and is exact for all polynomials of degree 3 and less.

Chapter 7

Lab 5: Euler's Scheme for Solving Ordinary Differential Equations

A large number of schemes have been developed for numerical solution of ODEs. Key words in characterizing those schemes include explicit, implicit, fully implicit, semi-explicit, simply implicit, high-order, variable step, adaptive stepping, and Runge-Kutta. There are also numerous books in this subject, for example, Hairer et al. [2, 3] and Shampine et al. [4].

MATLAB provides a suite of ODE solvers, including `ode15s` and `ode45`. The reader is referred to Shampine and Reichelt [5] or the corresponding MATLAB documents.

7.1 Problem description

In this lab we study to write a code for Euler's scheme for numerically solving the initial value problem (IVP) of ordinary differential equations (ODEs) in the form

$$\begin{cases} \frac{dy}{dt} = f(t, y), & \text{for } t_0 < t \le t_{final} \\ y(t_0) = y_0, \end{cases} \tag{7.1}$$

where $f(t, y)$ (a function), t_0, y_0, and t_{final} are given, and

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad f(t, y) = \begin{bmatrix} f_1(t, y_1, ..., y_m) \\ \vdots \\ f_m(t, y_1, ..., y_m) \end{bmatrix}.$$

For a given time step, Δt, Euler's scheme for (7.1) is given by

$$y^{n+1} = y^n + \Delta t\, f(t_n, y^n), \quad n = 0, 1, ... \tag{7.2}$$

where $t_n = t_0 + n\Delta t$, y^n is an approximation of $y(t_n)$, i.e., $y^n \approx y(t_n)$. Euler's scheme is explicit (in the sense that there is no need to solve a system of algebraic equations

when computing the new approximation y^{n+1}) and is known to have first-order convergence (meaning that the error $e^n = y^n - y(t_n) = \mathcal{O}(\Delta t)$). Euler's scheme is one of the simplest schemes for numerically solving (7.1). Its advantages lie in its simplicity and relative ease to program. Its disadvantages include its low-order convergence and restrictive stability condition which may force to use a very small Δt (problem dependent) to ensure the stability of the computation.

7.2 Planning

We start with figuring out the input/output variables. We need the variables from the problem, f (the right-hand side function), t_0, t_{final}, and y_0. For the numerical scheme, we need the value of Δt. For output, the code should return the approximations y^n, $n = 0, 1, ...$ and the corresponding time instants. Thus,

Input variables: f, t_0, t_{final}, y_0, Δt
Output variables: $(T, Y) = \{(t_n, y^n),\ n = 0, 1, ...\}$

Next, we decide the first line of the function as

```
function [T, Y] = ExplicitEuler(f, t0, tfinal, y0, dt)
```

We can now decide the flowchart/algorithm. The reader is encouraged to give a try and then compare yours to the one given in Algorithm 7.2.1.

Algorithm 7.2.1 Euler's scheme for ODEs.

```
function [T, Y] = ExplicitEuler(f, t0, tfinal, y0, dt)
```

1. Set $N =$ integral part$((t_{final} - t_0)/\Delta t)$. If $N\Delta < t_{final}$, let $N := N + 1$.
2. Initialization: $y = y_0$, $t = t_0$.
3. Preallocate T and Y: $m = length(y_0)$, $T = zeros(N, 1)$ and $Y = zeros(N, m)$. Set $T(1) = t$ and $Y(1, :) = y^T$.
4. for $i = 2 : N$
   (a). $y := y + \Delta t * f(t, y)$
   (b). $t := t + \Delta t$
   (c). $T(n) = t$
   (d). $Y(n, :) = y^T$

7.3 Coding

An example of the code for Algorithm 7.2.1 is given in Algorithm 7.3.1. Note how the function `f(t,y)` should be defined. This is especially important when dealing with ODE systems (versus scalar ODEs). One may notice that both `T` and `Y` are preallocated on Lines 23-24. Finally, pay special attention to the form how the approximations of y are saved in `Y`. To help the user, comments are given on Lines 4-14 for the output variables.

7.4 Testing

It is known that Euler's method is exact for linear polynomials (such as $y = t$). Thus, we consider $f(t, y) = 1$, $t_0 = 0$, $t_{final} = 1$, and $y_0 = 0$, and take $\Delta t = 0.01$. We expect that the approximation is the same as the exact solution $y = t$. A call to the function is

```
[T, Y] = ExplicitEuler(@(t,y) 1, 0, 1, 0, 0.01);
```

A plot of `(T, Y)` shows a diagonal line connecting (0,0) and (1,1), which indicates the approximation is `Y = T`.

The next example is the Lorentz system that was first developed by Edward Lorenz in 1963 as a simplified mathematical model for atmospheric convection. It reads as

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x), \\ \frac{dy}{dt} = x(\rho - z) - y, \\ \frac{dz}{dt} = xy - \beta z. \end{cases} \tag{7.3}$$

Here, we take the values used by Lorentz, $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$, for which the system exhibits chaotic behavior. A file, named `fun_Lorentz.m`, is created to define the right-hand side function; see Algorithm 7.4.1. A call to `ExplicitEuler` for this system is given in the following, where $t_0 = 0$, $t_{final} = 500$, $y_0 = (1, 1, 1)^T$, and $\Delta = 0.01$.

```
[T, Y] = ExplicitEuler(@fun_Lorentz, 0, 500, [1;1;1], 0.001);
```

The resulting trajectory obtained with `plot3(Y(:,1),Y(:,2),Y(:,3),'-')` is shown in Fig. 7.1.

7.5 Heun's and RK4 schemes

We record here two schemes so the interested reader can use them to practice programming. The first one is Heun's scheme which reads as

$$k_1 = \Delta t f\left(t_n,\, y^n\right),$$

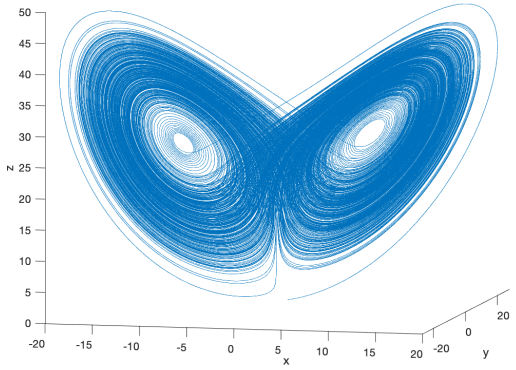

Figure 7.1: A trajectory of the Lorentz system.

$$k_2 = \Delta t f\left(t_n + \Delta t,\, y^n + k_1\right),$$
$$y^{n+1} = y^n + \frac{1}{2}\left(k_1 + k_2\right).$$

This method is known to have second-order convergence.

The second scheme is the Runge-Kutta method of order 4 (RK4). It is given by

$$k_1 = \Delta t f\left(t_n,\, y^n\right),$$
$$k_2 = \Delta t f\left(t_n + \frac{1}{2}\Delta t,\, y^n + \frac{1}{2}k_1\right),$$
$$k_3 = \Delta t f\left(t_n + \frac{1}{2}\Delta t,\, y^n + \frac{1}{2}k_2\right),$$
$$k_4 = \Delta t f\left(t_n + \Delta t,\, y^n + k_3\right),$$
$$y^{n+1} = y^n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).$$

This method is known to have fourth-order convergence.

Algorithm 7.3.1 Euler's scheme for ODEs: the code.

```
function [T, Y] = ExplicitEuler(f, t0, tfinal, y0, dt)
% this function implements Euler's scheme for ODEs.
%
% Input:
% f:  the right-hand side function handle.  it should be defined
%     in the form y = f(t,y) and returned with a column vector
%     of values.
% y0:  the initial solution in column vector.
%
% output:
% T: size of N-by-1, for time instants, t_n is saved in T(n)
% Y: size of N-by-m, for approximations, jth column, Y(:,j)
%     saves the approximations of the jth component of y
%     at the time instants, i.e., Y(n,j) approximates y_j(t_n).

    N = floor((tfinal-t0)/dt);
    if N*dt < tfinal
        N = N + 1;
    end
    t = t0;
    y = y0;
    m = length(y0);
    T = zeros(N,1);
    Y = zeros(N,m);
    T(1) = t;
    Y(1,:)  = y';
    for n = 2:N
        y = y + dt*f(t, y);
        t = t + dt;
        T(n) = t;
        Y(n,:)  = y';
    end

end % end of ExplicitEuler()
```

Algorithm 7.4.1 The right-hand side function for the Lorentz system: the code

```
function f = fun_Lorentz(t, y)
% this defines the right-hand side function for Lorentz system.

    sigma = 10;
    beta = 8/3;
    rho = 28;
    f = zeros(3,1);
    f(1) = sigma*(y(2)-y(1));
    f(2) = y(1)*(rho-y(3))-y(2);
    f(3) = y(1)*y(2)-beta*y(3);

end % end of fun_Lorentz()
```

Bibliography

[1] R. L. Burden, J. D. Faires, and A. M. Burden. *Numerical Analysis*. Brooks Cole, 10th edition, 2015.

[2] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations. I*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993. Nonstiff problems.

[3] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations. II*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1996. Stiff and differential-algebraic problems.

[4] L. F. Shampine, I. Gladwell, and S. Thomson. *Solving ODEs with MATLAB*. Cambridge University Press, 2003.

[5] L. F. Shampine and M. W. Reichelt. The MATLAB ODE Suite. *SIAM J. Sci. Comput.*, 18:1–22, 1997.