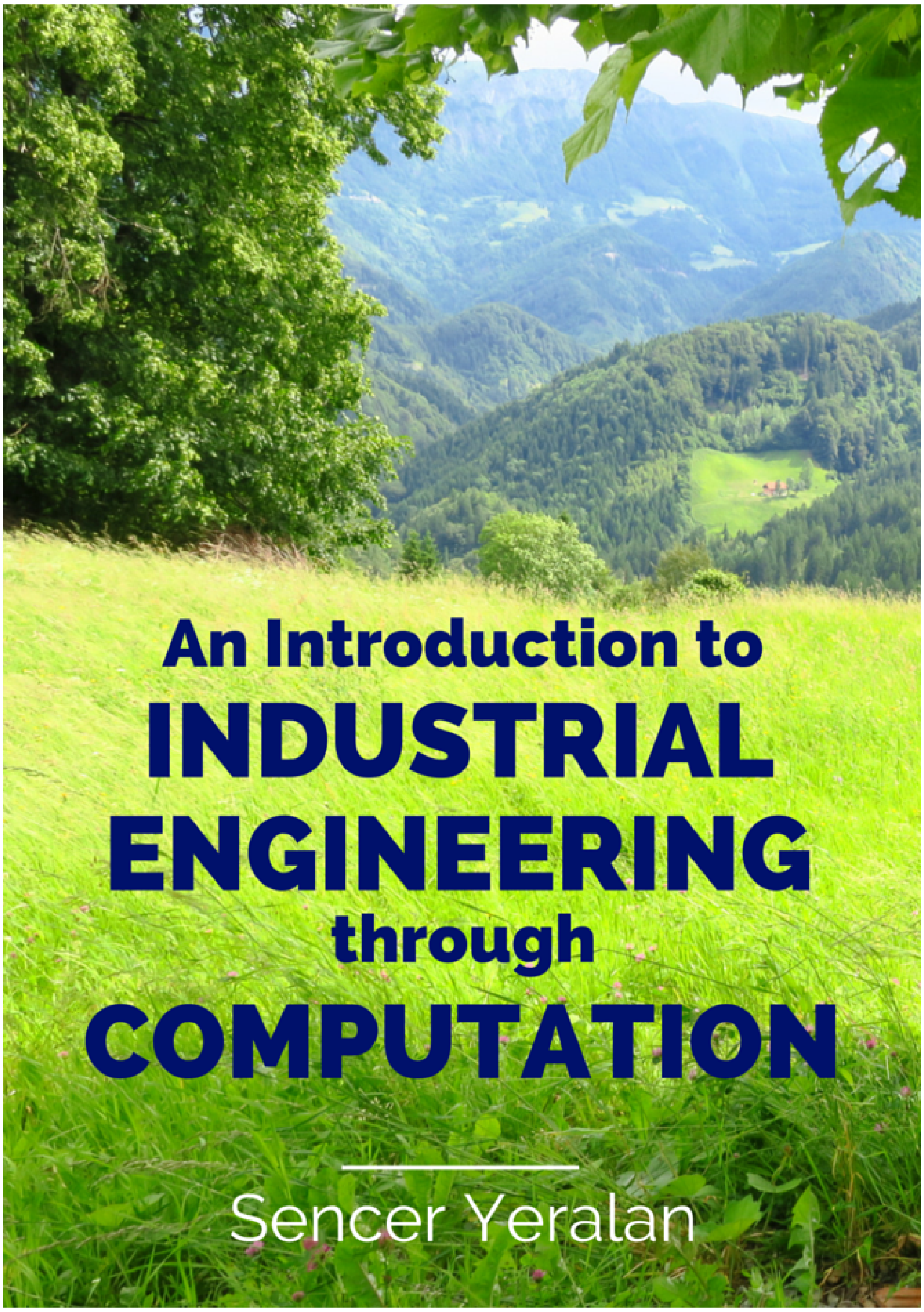


An Introduction to INDUSTRIAL ENGINEERING through COMPUTATION

Sencer Yeralan

An Introduction to Industrial Engineering through Computation

Sencer Yeralan

July, 2016

An Introduction to Industrial Engineering through Computation, First Edition, Sencer Yeralan, P.E., Ph.D., July 2016, Ed.: H. M. Emery.

ISBN 978-0-9633257-4-7

Books may be obtained by contacting the author by electronic mail or through the website, books@yeralan.org or books.yeralan.org.

Cover photograph: Davca, Slovenia, by the author.

...to The Great Professor.

Preface

This textbook is an outgrowth of a first-year course I have been teaching at Yasar University in Izmir. It is a somewhat nostalgic course for me. As a young student back in the early 1970s, I took the course called Engineering Sciences ES 100. There, the instructor told us about computation using a slide rule. The course was very informative. Our instructor was knowledgeable and entertaining. It provided a nice introduction to life as an engineer. I truly understood for the first time such concepts as accuracy, computational effort, modeling, approximation, order of magnitude, and error terms. What remains from the course is not computation, but rather, a general substrate of engineering. Then again, at the start of graduate school, I had a most insightful professor, who made an indelible impression upon me. This time we were free to use computers – any language we wanted: assembly, Fortran, or COBOL.

The days of slide rules have long past, although I keep mine handy. Nowadays we have wonderful software that takes the tedium out of computing, and leaves only its joy. In my courses, I use the Linux operating system and open-source software. This makes it attractive to students on a fixed budget or to those enterprising students who want to dig deep into the source code. I emphasize that the course is not a programming course. One may use any software, including a calculator, spreadsheets, or one may compute by hand.

Also gone are the days when we had to thumb through many books and magazines to just get an inkling of an idea. It was indeed an arduous task with a healthy dose of tedium -- unnoticed then, but probably most unpalatable to the reader today. Reading material has changed as well. One prefers concise and customizable instruction. Accordingly, this book is written as a minimalistic cookbook. It has the necessary ingredients to introduce the topic and give summary information. However, the full potential of the book is realized only when it is used with the accompanying example code. The combination of concise discussions and executable code is hopefully sufficient to whet the appetite and pave the road to understanding, and hopefully further, towards skillful mastery. The real benefit of example code is in making modifications and developing custom extensions. This reinforces the interplay between the visualized conceptual solution and the computational skills to be developed.

Computational skills are important to an engineer. Here, the term ”skill” should be emphasized. “It is a bit like riding a bicycle,” I tell my students. “It can be acquired only by practice.” After all, the theory of riding a bike is simple enough. One sits on the seat, puts feet on pedals and hands on the handlebar, alternatively pushes on the pedals while turning the handlebar to the intended direction. Simple enough, right? But just knowing the theory does not make one a rider. Nor is it possible to ride after watching someone else do it. Computation is also a skill. Computational skills are quite necessary for an engineer, to obtain numerical solutions and to develop insights. Just like riding a bike, though, one must practice computation to be good at it. This entails frequent mishaps and the occasional utterly painful failures. But there is really no other way. Also interesting is the fact that it is not easy, if not outright impossible, to fake such a skill. Just ask someone to ride ten paces on a bike. You can easily tell if the rider is a skilled cyclist or not. The same goes for computation. So, it is good advice to the young engineer to practice, practice, practice.

...as a textbook

An introductory course could be arranged around the this book. The book should be taken as a minimalistic guide, where supplementary material is added either in lectures or from on-line sources. The software will provide demos to facilitate active learning. In a typical course, the student is exposed to a series of industrial engineering models. The important concepts are discussed while simple, brute-force computational solutions are given. The objective is not to develop the theory of such models, nor is it to provide the most elegant or the most computationally efficient solution procedures. Rather, the objective is to introduce the concepts and start playing with the models at a very elementary level. Industrial engineering students will undoubtedly come across the theory and better solution procedures in their following years. Having been through this book, however, the industrial engineering student, when faced with such courses as optimization, location and layout, scheduling, queueing, etc., will have an idea of the conceptual ingredients of these topics. Moreover, if need be, the student could find solutions to some simplified problems by straight-forward computation. Such work is often the best means to gather insights into a new topic. Recall that much research in these fields start with entertaining example problems.

In my course, each chapter of the book takes one week to cover. I give reading assignments related to the topic, so that the students come to class (hopefully) prepare and knowledgeable. On a different day, the students meet for a laboratory session. There, the students are given a related numerical problem and asked to find a solution. I ask that students keep a good old-fashioned notebook and record their annotations and findings in a systematic manner. Students submit an engineering report at the end of the of the laboratory session. Engineering reports are to be concise and to the point, with rich engineering content but short on verbiage. As such, the course emphasizes experiential active learning and promotes good record-keeping skills.

There are several good textbooks on industrial engineering models, of which only a few are referenced at the end of the book. As needed, the student is encouraged to find on-line reading material and video tutorials related to the topics. Such material is not referenced, since it is voluminous and dynamically changing. A simple on-line query would return many good tutorials on any one of the topics covered. In this respect, the course also inaugurates lifelong learning paradigms and skills.

The experiments in this book will provide the necessary foundation for the student who wants to develop computational skills. Although the examples are from industrial engineering topics, no prior knowledge in industrial engineering is needed. This makes the book suitable not only for industrial engineering, but also for related disciplines, where computational skills may come in handy.

The example code given in the chapters is available on line at books.yeralan.org.

Sencer Yeralan
Izmir, July 2016

Table of Contents

I. ENGINEERING COMPUTATION SOFTWARE

Computation is central to engineering. Crunching numbers is not, however, the end goal of engineering. Engineers design and build products, in a responsible way, to serve practical needs. Responsibilities regard economical, environmental, and ethical issues. Engineering design relies on computation. Computation focuses the design effort so that, rather than trial and error, components and parameters are selected base on calculation. Engineers analyze systems, which also requires computation. Analysis often is needed to uncover problems, or to identify leverage areas where systems may be improved, so that next-generation products are better than the current ones.

The way engineers compute has changed over the years. Early on, all engineering computation had to be carried out by hand. The pace of computations, as well as its resolution, suffered. Over the years, many clever numerical techniques have been devised to make computation as efficient as possible. Some of these techniques are essential ingredients of the skill set of any engineer. The 20th Century saw the slide rule, followed by mechanical and electronic calculators, and finally the ubiquitous do-it-all computer. Today, the computer, including its many platforms (e.g. tablets) is the default device that is used in engineering computation. However -- and this is the crucial point – just because the computer solves many problems, engineers cannot forgo their responsibility. The engineers must be in the drivers seat, recognizing that the computer is only as effective as the engineer who understands the domain and uses the software.

There are many good engineering software products available to the engineer. Many of them are open-source software. The student should note that most of these programs share a similar syntax. Thus, switching from one platform to another is not much different than switching calculators. At least for a competent engineer.

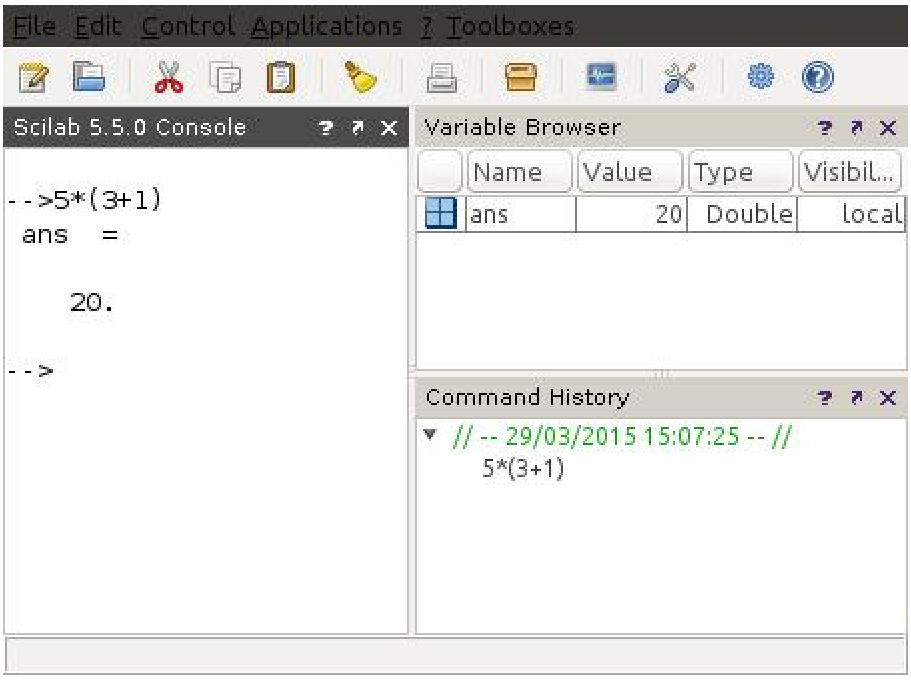
In this book, we will use mostly Scilab and Octave, two of the most widely used open-source software products. Commercial software products such as Matlab and Mathematica are also powerful platforms. However, the open-source products, being freely available on various operating systems, are more than sufficient for our purposes.

This book is written to serve as an introduction to students of industrial engineering. In this sense, it targets two equally worthy objectives. First, it intends to instill engineering skills, discipline, and intuition into the first year student. In addition, it aims to introduce the student to some of the basic models encountered in the field of industrial engineering. The topics are selected such that the industrial engineering student is resented with a preview of subjects to come. However, the topics are treated in a rather informal manner so that the non-industrial engineering student could also find the material interesting. This is done by a series of simple computational tasks. The student will be exposed to, by hands on experimentation, topics in industrial engineering, as well as concepts of computation, programming, and algorithms. The honing of general problem solving skills are encouraged by hands on simple computational tasks.

The student is expected to have a background in calculus and linear algebra. Following the contemporary modes of learning, the student is also expected to go through software manuals to identify the relevant software features that will come useful in tackling the computational tasks.

1. The Command Line

Simple computations can be carried out on a calculator. Most engineering software include a command-line interface where calculator-like instructions may be entered. The following figure shows the command interface of Scilab, which is called the Scilab Console.



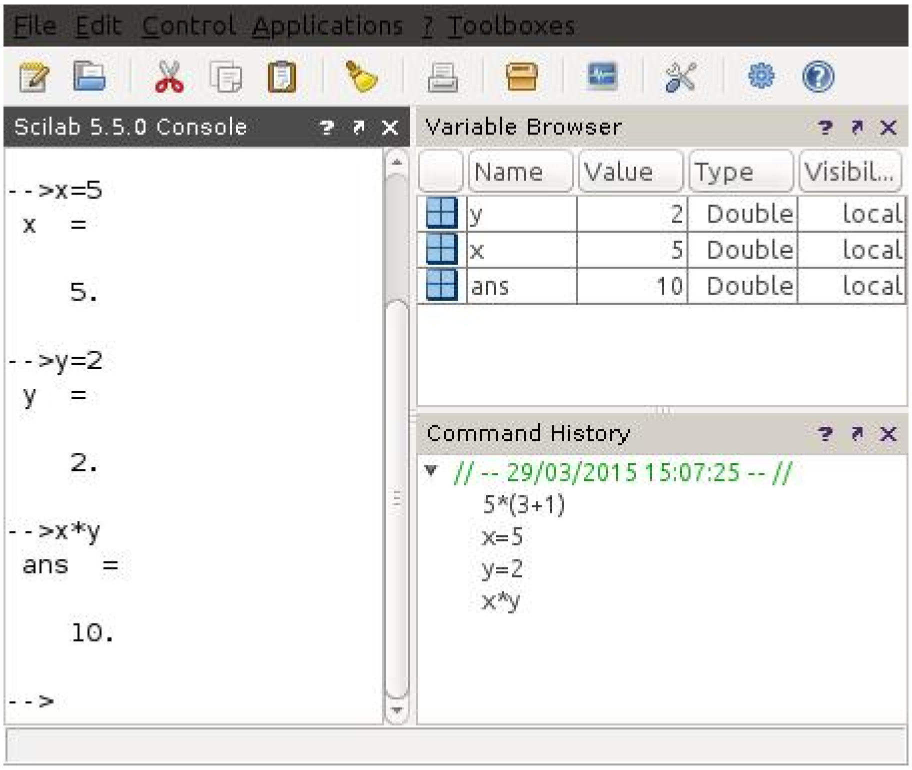
```
-->5*(3+1)
 ans  =

    20.

-->
```

	Name	Value	Type	Visibil...
	ans	20	Double	local

```
▼ // -- 29/03/2015 15:07:25 -- //
      5*(3+1)
```

Figure 1.1. The Scilab Console.

The expression 5*(3+1) is written into the Console window. When the user presses the “Enter” key, the answer (20) is displayed. Note that the Console has features above and beyond a calculator. For example, you may define variables and then carry out computations using these variables.



```
-->x=5
 x  =

    5.

-->y=2
 y  =

    2.

-->x*y
 ans  =

    10.

-->
```

	Name	Value	Type	Visibil...
	y	2	Double	local
	x	5	Double	local
	ans	10	Double	local

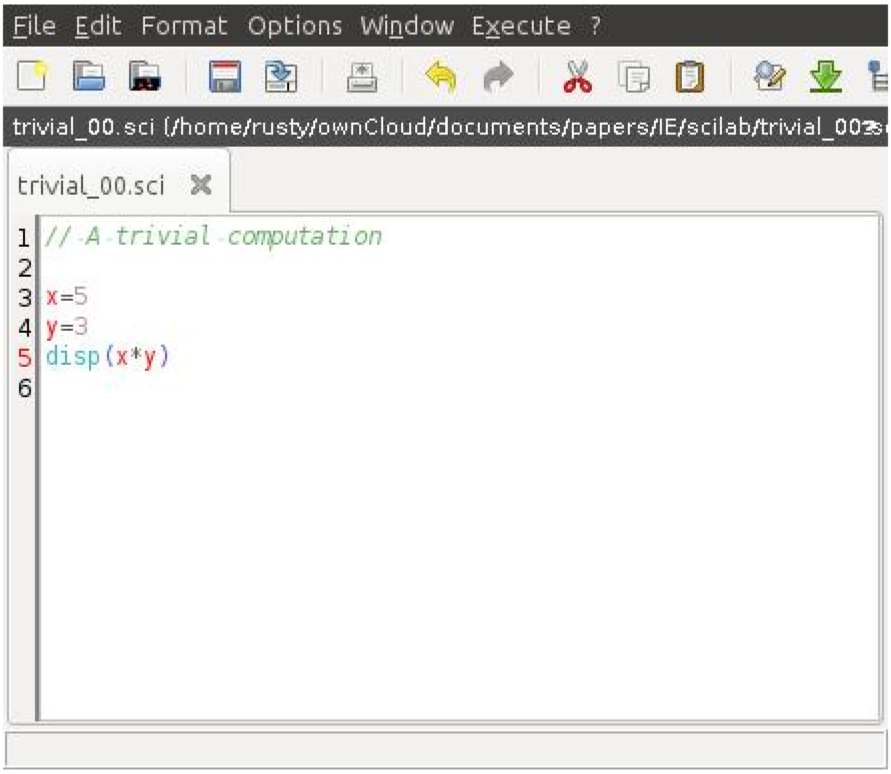
```
// -- 29/03/2015 15:07:25 -- //
    5*(3+1)
    x=5
    y=2
    x*y
```

Figure 1.2. Defining simple variables.

Note that Scilab keeps a history of the commands. You may double-click on the command in the "Command History" window to re-issue the command. Similarly, there is a "Variable Browser" which shows the defined variables and the results of the computations. Again, if you double-click on the variables in this window, further information about that variable is shown.

2. The Editor

Issuing commands on a instruction-by-instruction basis in the Console is adequate for the simpler calculations. If one wants to carry out detailed computations, then the "Editor" is more convenient. The "Editor" allows you to write your instructions one after another, and then save the file. You may run the instructions, or modify the file to update your instructions. This amounts to a programming effort, where the list of instructions saved by the Editor may be regarded as a program.


File Edit Format Options Window Execute ?
trivial_00.sci (/home/rusty/ownCloud/documents/papers/IE/scilab/trivial_00.s
trivial_00.sci

```
// A trivial computation

x=5
y=3
disp(x*y)
```

Figure 1.3. A simple script in the Scilab Editor.

The simple three-line script shown in Figure 1.3 defines x and y and then displays the result of x*y using the built-in disp() function[1].

The double slashes in the beginning of line 1 specify the beginning of a comment. Comments are short notes embedded in your code. They are ignored during execution. Good comments are essential to make your code readable.

Note that there are different options to execute the script. “Execute with echo” simulates the command console interface. Each time a line is executed, the result is echoed on to the screen. This is good for debugging your scripts. “Execute without echo” requires you to explicitly issue instructions to display the results.

Each of the engineering computation software has many utilities and features to be explored. It is left to the student to go through the menus and refer to the software manuals to become familiarized with

1 As a convention of the book, we give the scripts but not the detailed explanations of the built-in functions. The student is referred to the software manuals for information on these functions.

these. Software skills always come handy when you face a real-life engineering task.

3. Exercises

1. Define the constants C1=2, C2=4, and C3=9.
2. Using the constants defined in Exercise 1, compute $C1^{C2}$, and C2*(C2/C3).

II. FUNCTIONS AND GRAPHS

Functions may be expressed analytically, as lists, or graphically. Engineers like graphs, since they pack a lot of information in an easily observable manner. A list of independent variables, with corresponding dependent variables, is a good way to express a function empirically. Whereas analytical expressions are concise forms to express functions, especially if one is using calculus, a list of independent and dependent variable values is often sufficient for numerical work. Not surprisingly, engineering computation software includes many features that facilitate graphing functions.

1. A Simple Graph

The following Scilab code defines a vector x and a vector y, whose elements are the squared values of the vector x.

```
// A trivial function and its plot

x=1:0.1:10
y=x.^2
plot(x,y)
```

Figure 2.1 A simple function.

The independent vector is defined by the line

```
x=1:0.1:10
```

The colons are used to define ranges. Here, there are three values separated by the two colons. These are the start value (1), the step size (0.1) and the end value (10). You should view the details of vector x in the Variable Browser.

The next line defines the dependent variable (y) values.

```
y=x.^2
```

Each element of y is defined as the square of the corresponding element of x. The usual symbol for power is the circumflex (^). However, note that rather than x^2, we use the notation x.^2, that is (.^). The period before the symbol indicates that the power is to be computed element-wise. Otherwise, x^2 would indicate the multiplication of the vector x with itself. The vector x is a row vector.

The multiplication of a row vector and a row vector is not defined. You may multiply x with its transpose (i.e., the dot product) in which case, the result is a scalar. Element-wise operations are what we need here.

The last line plots the function as pairs of x and y values in a separate window. Scilab puts values on the two axes. Refer to the manuals to find out how you may add additional information, such as axis titles, to the plot.

```
plot(x,y)
```

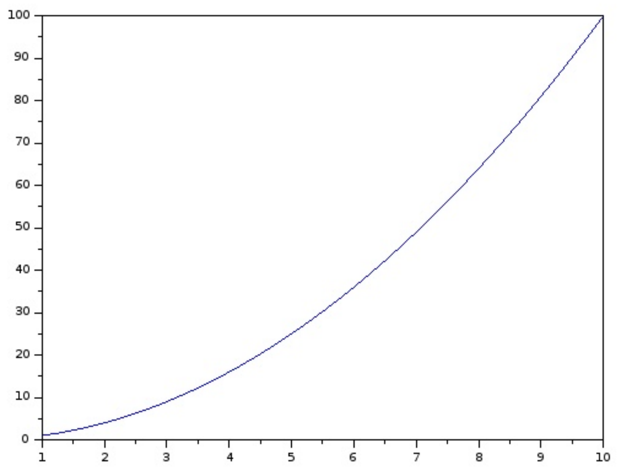


Figure 2.2 The plot of a simple function.

2. Exercises

1. Refer to the documentation of your engineering software to find information on the sin() function. Plot the function sin(x) for x in the range -4 to +4 radians.
2. It was mentioned that a dot product is possible provided that a row vector is multiplied by a column vector. The vector x used in the example is a row vector. Refer to the documentation of your engineering software to find out how you may transpose the vector x. Then find the dot product of x and x^T.

III. LINEAR EQUATIONS

In the previous chapter, we used vectors. Engineering applications also make extensive use of matrices. A matrix may be defined element by element. The syntax is fairly common.

```
A=[1 2; 3 4]
```

This defines the 2-by-2 matrix

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$$

Note that the elements of a row are separated by spaces. Placing commas also works in most software, as below.

```
A=[1, 2; 3, 4]
```

1. Matrix Operations

Engineering computation software allows operations that involve matrices and vectors. Again, remember that if you want the operations to be carried out element-wise, place a dot before the operator. Of course some matrix operations are already element-wise, for example A+B is already computed element-wise. However A*B refers to matrix multiplication, while A.*B multiplies the elements of A and B element-wise. The former matrix multiplication requires that the number of columns of A is the same as the number of rows of B. The element-wise operation is permitted only if A and B have the same dimensions.

2. An Example

A childhood riddle states that a farmer had chickens and sheep. The total number of heads were 6 while the total number of feet were 16. How many chickens did the farmer have?

Th riddle may be formulated by two linear equations in two unknowns. Let C and S be the number of chickens and the number of sheep.

$$C+S=6$$
$$2C+4S=16$$

In matrix notation, we have,

$$\begin{pmatrix}1 & 1\\ 2 & 4\end{pmatrix}\begin{pmatrix}C\\ S\end{pmatrix}=\begin{pmatrix}6\\ 16\end{pmatrix} \quad .$$

In the console, let us define the square coefficient matrix and the right-hand-side matrix.

```
-->A=[1 1;2 4]
 A  =

    1.    1.
    2.    4.

-->r=[6 16]'
 r  =

    6.
    16.

-->(A^-1)*r
 ans  =

    4.
    2.
```

Figure 2.1. Console Dump.

Figure 2.1 shows the definition of the coefficient matrix (A) and the right-hand-side vector (rhs) in Scilab. Note that vectors in Scilab are defined as row vectors. The tick mark (single quotation mark) is used to transpose the defined vector to a column vector. From then on, the solution is computed simply as the inverse of matrix A, or a to the power -1, multiplied by the right-hand-side vector. The console returns the solution vector $(4,2)^T$ indicating that the farmer has 4 chickens and 2 sheep.

3. Exercises

1. Write the steps in the example as a script and execute the script to find the solution.

2. Generalize your script in Exercise 1 to allow the user to input the number of heads and the number of feet. (Hint: Most software has an input() function that prompts the user and returns a value. Refer to the documentation of your software.)
3. Define 3-by-3 matrices A, B, and C. Then compute A*(B+C).
4. Using the matrices defined in Exercise 3, find (A*B) and (A .* B). How do these two differ?
5. Refer to the documentation of your software and find the determinant of a 3-by-3 matrix.
6. Refer to the documentation of your software and find out how you may define a N-by-N matrix whose elements are random numbers in the range 0 to 100.

IV. GROWTH MODELS

Growth models predict the population as a function of time. The so-called Malthusian model, named after Thomas Robert Malthus, is given by the simple differential equation

$$\frac{dP(t)}{dt}=r\,P(t)$$

Where P(t) is the population at time t, and r is the birth rate. The differential equation states that the change in population at time t is proportional to the population at time t.

A discrete version of the Malthusian growth can be written as,

$$\Delta P=P_{n+1}-P_n=r\,P_n$$

which yields,

$$P_{n+1}=(1+r)P_n \quad .$$

The Malthusian growth model is thus completely determined when the initial population (that is, P(0) or P_0) and the birthrate r are given. The resultant population grows exponentially (geometrically for the discrete case) without limit.

The growth is limited by the introduction of a carrying capacity by Pierre Francois Verhulst. The Verhulstian model assumes that K is the maximum population sustainable by the environment. The parameter K is called the carrying capacity. The model reduces the birthrate as the population approaches K.

$$\frac{dP(t)}{dt}=r\,P(t)(1-\frac{P(t)}{K})$$

Notice that, compared to the Malthusian case, the model has an additional term. As P(t) is small compared to K, the model behaves like the Malthusian model, since the additional term is close to unity. However, as the population approaches K, the additional term approaches zero, effectively cutting off the birthrate. The discrete case is similarly evaluated.

$$\Delta P=P_{n+1}-P_n=r\,P_n(1-\frac{P_n}{K})$$

or

$$P_{n+1}=P_n(1+r(1-\frac{P_n}{K})) \quad .$$

1. Graphing Growth

The following Scilab code computes the population for the next 100 years. The birthrate is taken as 0.01, and the initial population, p0=100.

The vector population is initialized to a single value of p0. A *for loop* is used to iterate through 100 years.

```
clc

// Malthusian growth
// --- parameters ---
p0=100;                       // initial population
birth_rate=0.01;          // birth rate
// --- population ---
population=[p0];        // population (count)

for year=1:100
// update population
population($+1)=..
                 (1+birth_rate)*population($);
end

scf(0);
plot(population);
legend('population');
xtitle('population');
```

Figure 4.1. Malthusian Growth (Scilab Code).

There are several interesting points to note.

1. The initial instruction *clc* clears the console.
2. The vector population is initialized to be a vector of size one, holding the value p0.
3. A loop is formed where the loop variable *year* is incremented, starting from 1, up to 100. The loop starts with the keyword *for* and ends with the keyword *end*.
4. The equation within the loop is executed 100 times, as specified with the *for* statement. The right-hand-side of the equation refers to the element of the vector *population($)*. Here, the '$' indicates the index of the last element of the vector. Thus, *population($)* is the last population available in the vector.

5. The population of the following year is similarly defined as *population($+1)*. Note that as new elements are defined, the length of the vector is incremented. Adjusting the size of vectors (arrays, in general) is a nice feature of such high-level languages.
6. The equation spans two lines. The double dots (..) at the end of the first line indicates that the next line is a continuation, and must be read by the interpreter before executing the instruction.
7. The Scilab function scf(N) selects figure window N. Here, we will draw our graph in window 0.
8. The Scilab functions xtitle() and legend() place meaningful labels on the horizontal axis and name the graph. If we want to plot multiple graphs in the same window, preferably in different colors, a legend for each graph would make reading the display more readable.

The resultant graph shows the typical geometric population growth.

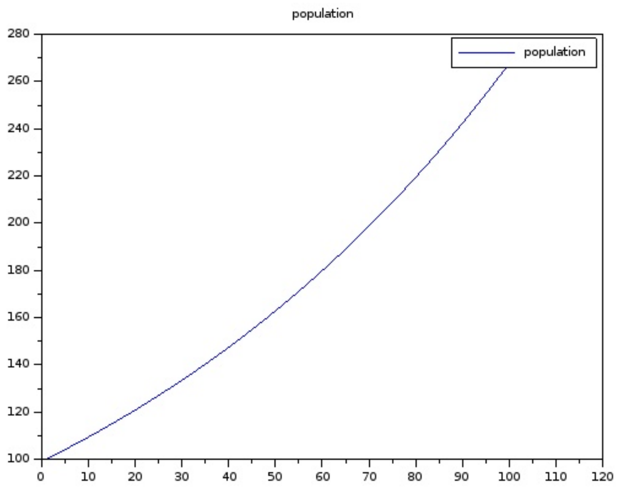


Figure 4.2. Malthusian Growth.

2. Graphing Limited Growth

The addition of a carrying capacity to the code in the previous section results in limited growth.

```
clc
// Verhulst growth
// --- parameters ---
```

```
p0=100;                                 // initial
population
birth_rate=0.01;                     // birth rate
K=1000;                                 // carrying capacity
// --- population ---
population=[p0];                     // population (count)

for year=1:1200
// update population
 pop_growth=..
      birth_rate*population($)*(1-population($)/K);
 population($+1)=population($)+pop_growth;
end

scf(0);
plot(population);
legend('population');
xtitle('population');

d=diff(population);
scf(1);
plot(d);
legend('population(n+1)-population(n)');
xtitle('population change');
```

Figure 4.3. Verhulst Growth (Scilab Code).

A carrying capacity of 1000 is used. The code is very similar to the previous case, except for the added term in the equation. The loop is run for 1200 years. Note also that a second window is used to plot the change in population. A new vector (d) is obtained by calling the Scilab function diff(population). The function diff() computes the successive differences of a given vector. Note that the resultant vector has one fewer element than the original vector.

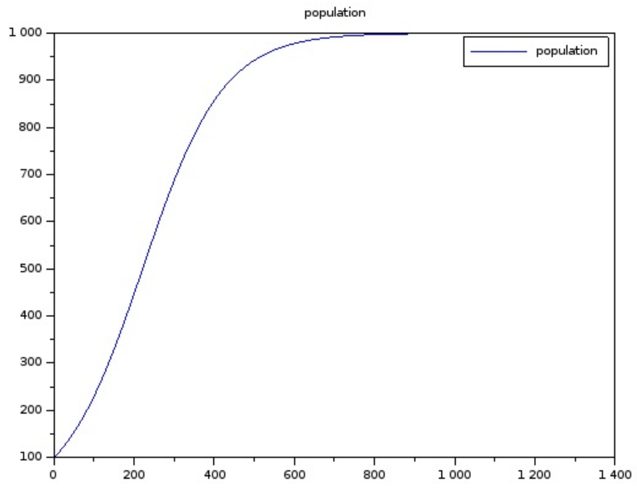


Figure 4.4. Verhulst Growth.

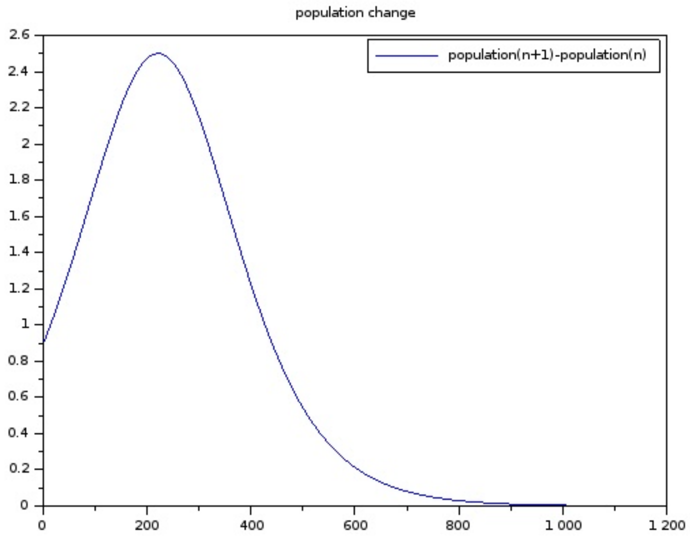


Figure 4.5. The Change in Population with Verhulst Growth.

Figure 4.5 shows the change in population over time. It is observed that the population growth is reduced to zero as the population approaches the carrying capacity.

3. Exercises

1. Growth models involving a predator and a prey are extensions of the models discussed in this chapter. The Lotka–Volterra equations are given below.

$$\frac{dR(t)}{dt} = \alpha R(t) - \beta R(t) F(t)$$

$$\frac{dF(t)}{dt} = \delta R(t) F(t) - \gamma F(t)$$

   Here, R(t) is the rabbit population at time t, and F(t), the fox population. The parameters are shown by the four Greek characters. The parameters specify the birthrates as well as the interdependency between the predator and the prey.

   Model the above growth equations and plot the populations as functions of times. Experiment with different parameters and starting populations.

V. CHAOS AND RANDOMNESS

The growth model discussed in the previous chapter has been the subject of some investigation into systems that display chaotic behavior. Specifically, the difference equation

$$p_{n+1} = r\, p_n (1 - p_n)$$

inspired by the Verhulst Growth Model, given appropriate values of r, generates a sequence of numbers p_n that behave chaotically. If the sequence starts with a value in the range [0, 1] and the parameter $r<4$, then all elements in the sequence will be limited to the range [0, 1]. This gives the idea that successive elements of the sequence may be used as pseudo-random numbers. Interestingly, for ranges of the parameter r, the sequence behaves nicely and converges to target values determined by the parameter value.

1. An Implementation

The following code implements the difference equation.

```
// chaotic sequences – (pseudo-random numbers?)
// p(n+1)=r*p[n]*(1-p[n])
// --- parameters ---
 r=3.99;                // rate
 p0=0.5;                // initial value
 p=[p0];                // initial population

 for i=1:1000
   p($+1)=r*p($)*(1-p($));  // update population
 end

 scf(0);
 plot(p, 'ro');
 xtitle(sprintf('RNG, r = %f', r));
 xlabel('n');
 ylabel('Pn');
```

Figure 5.1. Chaotic Sequence based on Verhulst Growth.

You may have noticed the use of the command-line

```
 xtitle(sprintf('RNG, r = %f', r));
```

to put a title on the graph. The built-in function xtitle() places a given string as a title, as for instance, in xtitle('My Title'). In our example, a constant string is not given. Instead, we use the sprintf() function to construct a string. The sprintf() function is well known to users of C languages. It prints a formatted string. Here we want to construct the string to reflect the value of the parameter r. The '%f' is the place holder for the value of a floating point number. The value is usually taken from a variable which follows as a function argument. The student is referred to documentation on C language library functions for more information on printf() and its many variations including sprintf().

The code, when run, plots the sequence of numbers by small red circles , as shown below.

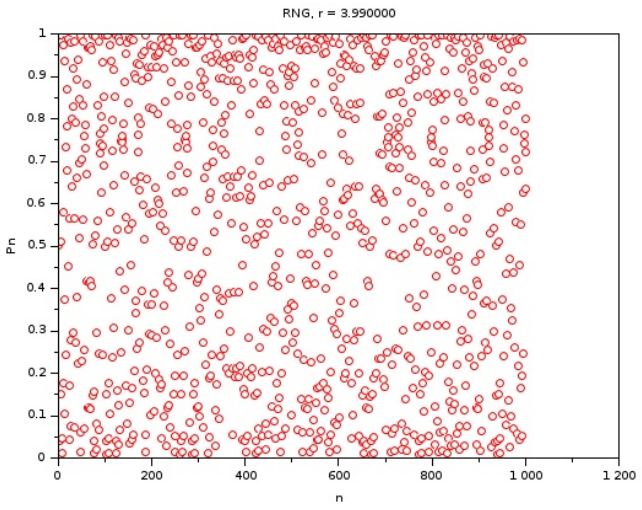


Figure 5.2. The Sequence.

Although the numbers look relatively well distributed over the range, the sequence is not suitable to be used as random numbers, since the sequence displays a very high degree of autocorrelation.

Autocorrelation describes how a sequence (or signal) is correlated with its time-shifted self. In the code below, we plot the values of p_{n+1} against the values of p_n. The plot reveals that if p_{n+1} is expected to be close to p_n.

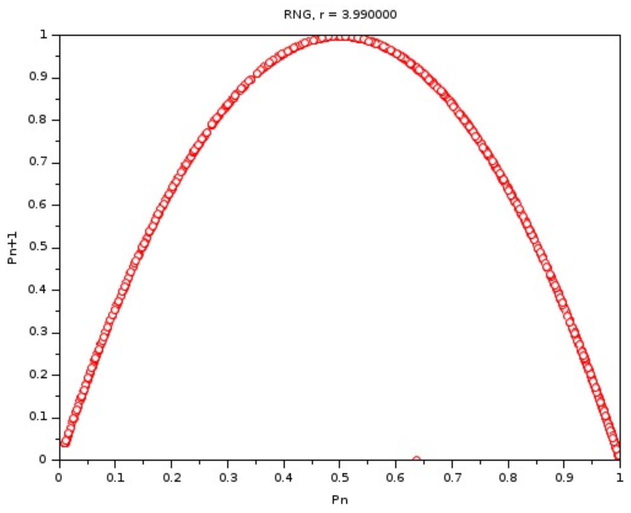


Figure 5.3. Successive Elements of the Sequence.

As can be seen from Figure 5.3, if p_n is close to the mid point, the next element is close to 1. Similarly, if p_n is close to either boundary, the next element is close to 0. Such predictability of the next element from the current disqualifies the difference equation from being used as a pseudo-random number generator, no matter how chaotic its output looks over time.

We repeat the experiment and produce a graph similar to that depicted in Figure 5.3, but this time using the built-in pseudo-random number generator function rand(). Each successive call to this function returns another pseudo-random number.

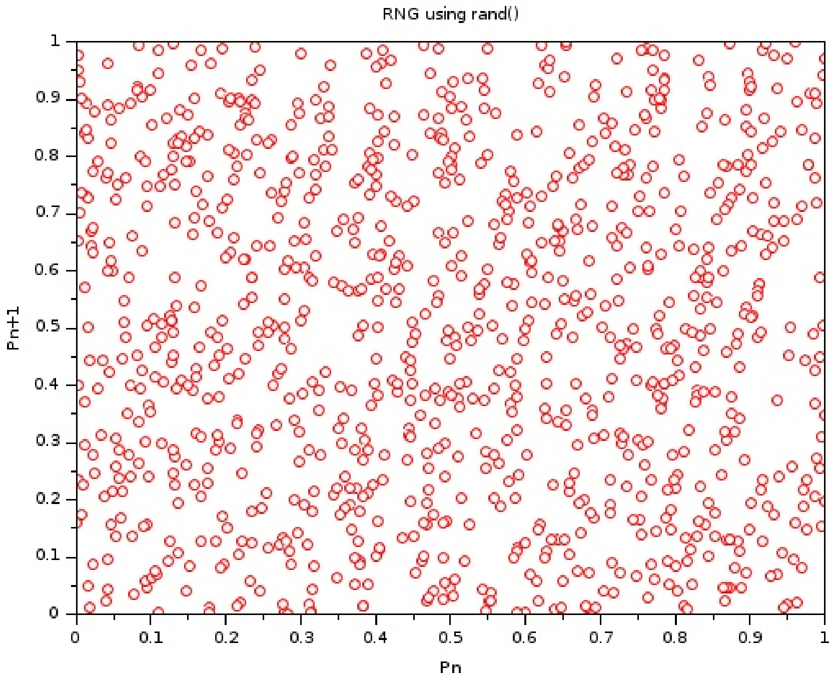


Figure 5.4. Successive Calls to rand().

Figure 5.4 shows that there is very little if any correlation between successive calls to rand(). That is, it is difficult to guess the value of P_{n+1} from the value of P_n where the elements of P_n are obtained from successive calls to rand().

2. Exercises

1. Try different values for parameter r and identify the ranges where the sequence behaves chaotically, and where it approaches a constant.
2. Suggest modifications to the difference equation to improve its usefulness as a pseudo-random number generator.

VI. SIMPLE NUMERICAL DIFFERENTIATION

Numerical differentiation and integration are essential engineering computations. Numerical differentiation is obtained simply by finding the difference between successive function values, separated by a given step size. Then, this difference in the function values divided by the step size approximates the slope of the function at those points.

1. An Example

Numerical differentiation is quite straightforward, as given by the following example.

```
// differentiate sin(x)
clc
clear

 h=0.1;                 // step size
 x=-%pi:h:%pi;          // -pi to pi in increments of h
 y=sin(x);

 // z numerical differentiation
 z=diff(y);
 z($+1)=z($);
 z=z./h;

 scf(0);                 // figure window 0
 plot(x, y, 'b');        // plot y in blue
 plot(x, z, 'r');        // plot z in red
 xlabel('x');
 ylabel('sin(x)');
 legend(['sin(x)';'d/dx sin(x)']);
 xtitle('numerical differentiation');
```

Figure 6.1. Numerical Differentiation of sin(x).

The code shown in Figure 6.1 plots sin(x) and its numerically computed differentiation.

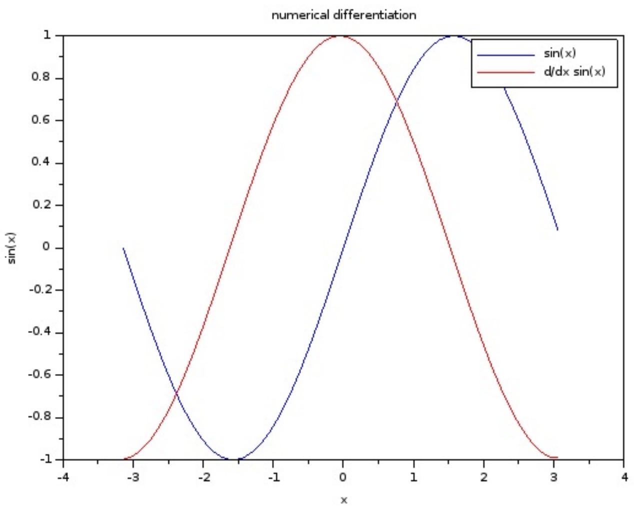


Figure 6.2. Graphical Output of the Example Code.

Note that the numerical differentiation looks as expected, that is, like cos(x).

2. Exercises

1. Try different values for the step size and re-plot the graphs. What values or range would you recommend for the step size? Why?
2. Select three common functions and plot their numerical derivative functions.

VII. SIMPLE NUMERICAL INTEGRATION

The numerical integration of a simple univariate function is useful to find the area under the function within a range of the independent variable.

1. An Example

The following example uses a function definition for the sake of code flexibility. By changing the function definition, you may quickly modify your code to integrate different functions.

```
clc
clear
// define the function -- try others
function [q]=f(a)
  q=sin(a)*cos(a);
endfunction

// lowerbound and upperbound define
// the interval
// h is the step size
// x is a vector of the independent
// variable values
// x has (upperbound-lowerbound)/h intervals
// x has 1+(upperbound-lowerbound)/h elements
// y is the vector of function values
// z is a vector of numerical
// integration values
// we compute the elements of z by
// incrementally adding the area under
// the function

h=0.1;
lowerbound=0;
upperbound=10;

// set the column x vector
x=(lowerbound:h:upperbound)';

// integral at lowerbound is 0
z=[0];
// function value at lowerbound
y=[f(lowerbound)];

for i=1:length(x)-1
```

```
    y(i+1)=f(x(i+1));
    z(i+1)=z(i)+f(x(i))*h; // rectangular
end

// plot the function and the
// numerical integration
scf(0);
plot(x, y, 'b');
plot(x, z, 'r');
xlabel('x');
ylabel('function and its integral');
legend(['f(x)';'the integral']);
xtitle('f(x) and its integral');
```

Figure 7.1. Numerical Integration.

The approach is to divide the function into narrow rectangles of width h. Here, 'h' is referred to as the step size. The rectangle at position x has an area h time the function value at x. The code simply loops through the range, computes the function values y(x) at x and accumulates the areas of the rectangles. The total area from the lower bound to x is stored as z(x), the value of function z at x. The function z(x) is thus the integral of the function y(x).

The code plots the function and its numerical integral, as shown below.

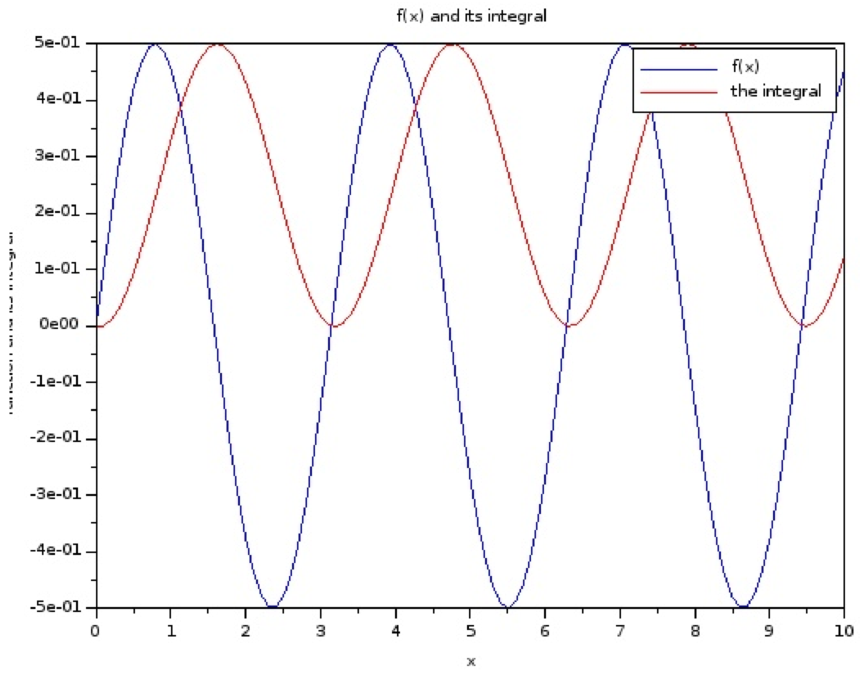


Figure 7.2. The function and its numerical integral.

2. Exercises

1. Try different values for the step size and re-plot the graphs. What values or range would you recommend for the step size? Why?
2. Select three common functions and plot their numerical derivative functions.

VIII. A SIMPLE ALGORITHM: THE BI-SECTION METHOD

Algorithms are ubiquitous computational tools in engineering. Although their use may extend to other domains, such as logic or signal processing, in engineering, they are used to obtain numerical solutions to problems. Perhaps a good way to understand what an algorithm is, is to understand what an algorithm is not. Take, for instance, finding the roots of a second degree polynomial. If the polynomial is a function, such as,

$$f(x) = ax^2+bx+c$$

and we are interested in finding the a value of x at which point makes f(x)=0, then we have the closed-form formula,

$$x = \frac{-b+\sqrt{(b^2-4ac)}}{2a}$$

which gives such a value of x. We call this, as mentioned, a “close-form” solution, since all one needs to do is plug in the given data (the coefficients) and perform the calculations.

In contrast, no such closed-form formula exists for a polynomial of degree 5. So, if an engineering problem requires a value of the independent variable at which point, a fifth degree polynomial yields the value of zero, then we resort to different computational approaches. An algorithm is such an approach.

An algorithm seeks solutions by an iterative method. At each step, computations are performed, whose results are hoped to converge to a solution. Whether such a convergence is guaranteed is an important question. A well-developed algorithm should not only produce results which converge to a solution, but should also do so in as few steps as possible. Some algorithms never find the exact solution. They only approach a solution. We discuss such an algorithm in the example below. The so-called bi-section algorithm is a flexible procedure that is applicable to a wide range of problems. Moreover, it has a good convergence rate, and is quite easy to implement.

1. An Example

Consider the polynomial

$$f(x) = x^5-2x^4+3x^3-4x^2+5x-6$$

We are interested in a value of x which makes f(x)=0. The bi-section algorithm starts with an interval of the independent variable, which is known to contain a solution. In this case we have a continuous and smooth function. The interval [-100, 100] is guaranteed to contain a solution, since f(-100)<0 and f(100)>0. Thus, someplace within the interval the function must have at least one point where it crosses the horizontal axis.

Each step of the algorithm reduces the interval by half. Hence the name of the algorithm. Each step cuts the interval in half.

Specifically, we pick the midpoint of the interval and compute the function value. If the function value is positive, we replace the current upper bound with the midpoint. That is, the midpoint becomes the upper bound for the next iteration. Otherwise, we replace the lower bound with the mid point. After N iterations, the interval is 1/(2^N) the width of the original interval.

We now give the code.

```
// find a zero of the polynomial
// f(x)=x^5-2x^4+3x^3-4x^2+5x-6
clc
clear

function [y]=f(x)
  y=x^5-2*x^4+3*x^3-4*x^2+5*x-6;
endfunction

LB=-100;  // initial lower bound
UB=100;   // initial upper bound
x=[];     // empty vectors
y=[];

for i=1:30          // 30 steps suffice

 x(i)=(UB+LB)/2;    // the current x value
 y(i)=f(x(i));      // the current y value

 // update the range [LB, UB]
 if y(i)*f(UB)>0 then UB=x(i);
  else LB=x(i);
 end

 printf("%d:%f %f\n",i, x(i), y(i));
end
```

Figure 7.1. The Bi-Section Algorithm.

The polynomial is implemented as a function for added flexibility. The remainder of the code is independent of the function. As such, the code may be used for other problems provided that the function, and possibly the initial interval are updated accordingly.

The initial interval is set to [-100, 100], defined by the variables LB and UB. Here, we implement the algorithm for 30 iterations. In effect, the width (200) of the initial interval is reduced to ($200/2^{30}=0.186\times10^{-6}$). The tail end of the output is shown below.

```
.
.
24: 1.491797 -0.000011
25: 1.491803  0.000056
26: 1.491800  0.000022
27: 1.491798  0.000005
28: 1.491798 -0.000003
29: 1.491798  0.000001
30: 1.491798 -0.000001
```

Figure 7.2. The Algorithm Output.

As seen, the algorithm converges to a solution (approximately 1.492).

2. Exercises

1. Improve the given code by implementing a terminating criterion. Define a tolerance, Epsilon with a value of 10^{-9}. Let the loop terminate when the width of the interval drops below Epsilon.
2. Add a check at the beginning of the code to make sure that the function value at the two ends of the interval are have opposite signs (i.e., one is positive and the other negative).
3. Modify the code to find a local extremum (maximum or minimum) of a given function.
4. Modify the code by implementing an initial step to ask the user for the initial interval. Plot the function in this interval and ask the user if the interval is appropriate, or if the user would like to issue a different initial interval.

IX. A SIMPLE ALGORITHM: THE SECANT METHOD

Before we continue with industrial engineering problems, we dwell on one more algorithm to rivet the ideas firmly in place. The Secant Method, like the Bi-Section Method already discussed, is a general-purpose algorithm. It has better convergence characteristics compared to the bi-section method.

The secant method uses the most recent two solutions to find the next. The most recent two solutions provide two points which are used to define a line. The value of the independent variable that makes this line cross the horizontal axis is use for the next solution. The method is summarized by a second-order difference equation, as given below.

$$x3 = x2 - f(x2)\frac{x2 - x1}{f(x2) - f(x1)}$$

Intuitively, the method has better convergence properties compared to the bi-section method, because it uses more information to pick the next solution. You may interpret this as using information on the slope, or as using a linear combination of the most recent two solutions. In contrast, the bi-section method simply picks the mid-point, irrespective of the change in the function values for the given independent variable values.

1. An Example

We use the same polynomial as in the previous chapter, namely,

$$f(x) \; = \; x^5 - 2x^4 + 3x^3 - 4x^2 + 5x - 6$$

and start with the same two solutions, x=-100 and x=100.

```
// find a zero of the polynomial
// f(x)=x^5-2x^4+3x^3-4x^2+5x-6

clc
clear

function [y]=f(x)
  y=x^5-2*x^4+3*x^3-4*x^2+5*x-6;
endfunction

epsilon=1e-8;     // tolerance
```

```
x1=-100;
x2=100;

for nCount=1:100
 if(abs(f(x1)-f(x2))<epsilon) then break; end
 x3 = x2 - (f(x2))*(x2 - x1)/(f(x2) - f(x1));
 x1=x2;
 x2=x3;
 disp([nCount x3 f(x3)])
end
```

Figure 8.1. The Secant Method.

The code is a straightforward implementation of the difference equation. The output is shown below.

```
.
.
 1.    1.9998       11.991803
 2.    1.9997999    11.991798
 3.    1.7071857    3.3174612
 4.    1.5952767    1.3550359
 5.    1.5180046    0.3072755
 6.    1.4953431    0.0402559
 7.    1.4919267    0.0014542
 8.    1.4917986    0.0000072
 9.    1.491798     1.314D-09
10.    1.491798     0.
```

Figure 8.2. The Algorithm Output.

Note that the algorithm terminates in ten steps with an error of less than the specified tolerance, Epsilon=10^{-8}. The loop is set to run for 100 steps. However, the loop is broken as soon as the error drops below Epsilon. The maximum loop count is a good idea, in case the loop never ends (for example, if one were to accidentally specify Epsilon to have a negative value).

2. Exercises

1. Add a check at the beginning of the code to make sure that the function value at the two initial solutions are well defined.
2. Is it possible to modify the code to find a local extremum (maximum or minimum) of a given function? If so, how would you accomplish this?

3. Pick another function and find its zero by the secant method and the bisection method. Observe and compare the convergence characteristics of these two methods.

X. AN INVENTORY MODEL

Inventory control is a fundamental topic in industrial engineering. Inventory control deals with ordering a batch of items to be stored in inventory. There may be differing modes of ordering the items. For example, there may be lead times (duration from the time an order is placed until when the order actually arrives) or orders may arrive instantaneously, probabilistic quantities, or various discount schemes. The items are drawn from the inventory by a prescribed operating policy as well. Here, there may also be different assumptions. Demand may be known and constant, or be probabilistic with a given distribution. Items may be backlogged (not immediately delivered but supplied to the customer at a later date. There may be multiple items that share the inventory, or multiple levels of inventories (e.g., a hierarchical system of distribution centers and local inventories). Moreover, items may have a limited self-life, as in blood stored in blood banks. Let us look at perhaps the simplest model, which still contains the minimal necessary components.

Consider a single item to be stored. The item is ordered in batches of quantity Q. Placing an order has a fixed costs. This may be a processing cost, or perhaps a shipping cost. Let each time an order is placed cost K dollars on top of the cost of the items. Let K be fixed, irrespective of the order quantity. Let there be a constant demand D. Finally, holding items in inventory has a holding cost h. This cost is per item per time unit.

In our model, we take demand, and the two costs as parameters. That is, D, K, and h are taken to be determined externally as requirements imposed on our system. These will be considered as parameters. The order quantity is a decision variable. We would like to find the best order quantity. When we say “best” we must also define how we compare one choice of the decision variable to another. This measure of “goodness” which will ultimately allow us to find “the best” is called an objective value. In most industrial engineering models, an objective function is constructed. It is a function, because the objective is measured as a function of the decision variables.

The dynamics for our inventory system is rather straightforward. If we were to consider the number of items in inventory over time, we would have a time dependent inventory level function as below.

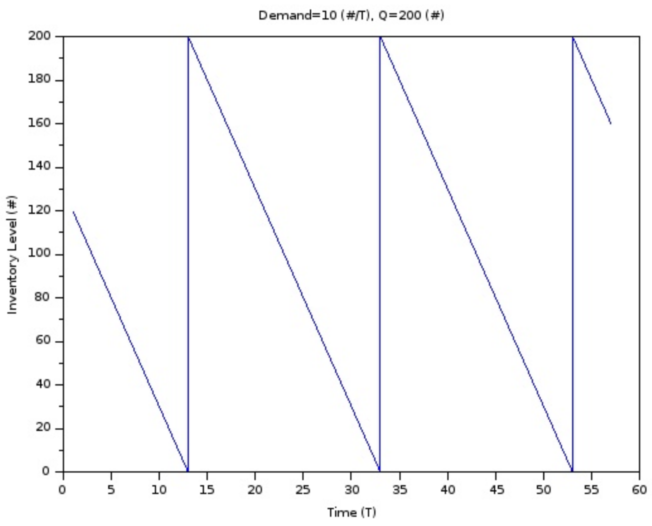


Figure 10.1. Inventory Level over Time

The inventory level shown in the figure above depicts the case where the demand is 10 items per time unit. The order quantity is 200 items, which arrive instantaneously. It follows that the order period is 20 time units.

A good – and often used – objective is to minimize cost. In this model, we have an infinite working horizon, so the total cost will be infinitely large. It is thus reasonable to consider the average cost per time period. Alternatively, we can look at the cost in a single order period and then divide this cost by the length of the order period.

1. The Model

As a first step to constructing a mathematical model, we first list the quantities as described in the previous section. In square parentheses, we give the unit of the variable. We use $ for cost, T for time and # for quantity (item).

```
Q: order quantity [#]
D: Demand [#/T]
K: Fixed ordering cost [$]
h: Inventory holding cost [$/(#T)]
```

Figure 10.2. Model Variables and Parameters

We made a special point about units being used. Engineering computations differ somewhat from the more abstract mathematics, such as number theory, in that we usually deal with quantities of tangible entities. These physical entities have not only a magnitude but also a quantity. In many cases, we even must make conversions from one unit to another. For example, the gravitational acceleration may be 9.81 m/s^2 or 32.17 ft/s^2 depending on whether we use meters or feet as our unit of length.

Our objective is to minimize cost per time in the long run. Looking at Figure 10.1, we remark on another fundamental concept in systems engineering. The inventory level starts at the order quantity and drops linearly until it reaches zero. Then the cycle repeats. We call this a recurrence. The order points may be regarded as renewal points, where the system simply repeats itself. Thus, any long-term property may be extracted from the properties of a single cycle. In other words, the single cycle captures all of the information regarding the system dynamics, irrespective of how long it is run. Things simply repeat. If we are to minimize the cost per time over the long run, it would suffice to study just a single period and minimize the cost over just one period.

2. Cost Components

There are two components of cost in our model, which we will call C_1 and C_2. Both are expressed as cost per time unit ([$/T]).

The fix cost K is incurred each cycle. A cycle repeats itself when demand D depletes the quantity Q. That is, a cycle lasts Q/D time units. Notice that Q/D has the correct units, that is, [#/(#/T)]=[T]. So the cost attributed to ordering is $C_1=K/(Q/D)=KD/Q$ per time unit ([$/T]).

Similarly, there is an inventory holding cost. The average inventory per cycle is Q/2 (why?). This amounts to a cost per unit time given by $C_2=h(Q/2)(Q/D)=hQ^2/(2D)$.

Total cost per time unit is thus $C=C_1+C_2$ in units of [$/T].

3. The Code

We now compute and plot C_1 and C_2.as well as their sum. The code is given below.

```
clc
h=10     // holding cost [$/(#T)]
K=50     // fixed cost   [$]
D=10     // demand       [#/T]
maxQ=100;
q=1:maxQ // order quantity
```

```
f=[]     // fixed cost
v=[]     // holding cost
c=[]     // total cost

// --- loop to compute costs for
// various values of q ---
for i=1:maxQ
 f(i)=K*D/i     // fixed cost
 v(i)=h*i/2     // holding cost
 c(i)=f(i)+v(i) // total cost
end

// --- plot costs versus order quantity ---
scf(0) // create figure 0
plot(q',f, 'g') // green : fixed cost/time
plot(q',v, 'b') // blue  : holding cost/time
plot(q',c, 'r') // red   : total cost/time
xtitle('cost/time', 'Order quantity (q)');
legend(['fixed cost/time',..
'inventory holding cost/time',..
'total cost/time']);
```

Figure 10.3. The Code.

The equations given in the previous section are implemented. The code simply computes the costs C_1 and C_2 for order quantity values from 1 to maxQ=100. The code also plots the costs C_1 and C_2 as well as their sum.

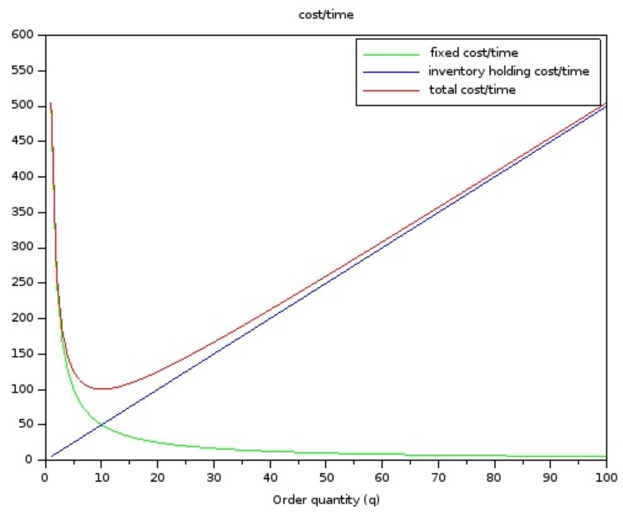


Figure 10.4. Cost Components.

The total cost per time is plotted in red. It is clearly seen that the total cost has a minimum around 10.

4. Exercises

1. Modify the given code to find the optimum order quantity. That is, the order quantity for which cost per time is minimum.
2. Modify the code to numerically differentiate the total cost per time function. Then use a search algorithm to find the order quantity for which the derivative of the cost per time function is zero.
3. Which of the two methods (Exercise 1 or 2) of detecting the minimum is better? Explain.
4. Industrial engineers speak of "sensitivity analysis" which are usually performed after an optimum is found. The idea is to investigate how sensitive the objective function value at the optimum solution is with respect to the given parameters or the decision variable. Compute the following:

   How much does the cost (C) change if we change the fixed ordering cost (K) by one unit? That is, give a numerical value for

$$\frac{\partial C(q,K)}{\partial K}$$

evaluated at the points K=50 and q=10. Note that K=50 is the parameter used in the example, and that q=10 is taken as the the ordering quantity that minimizes cost per time.

5. Repeat Exercise 4 for parameters h, and D.

XI. FACILITY LOCATION

Determining the optimum location of a facility is a typical industrial engineering task. For instance, we may be interested in finding the best location for a machining center somewhere on the factory floor. Similarly, we may be interested in placing a distribution center somewhere in the vicinity of several factories.

1. An Example.

An airport is to be built to service four cities. The city coordinates are given. As the airport is built, four roads must also be constructed linking each city to the airport. We assume that the terrain is rather flat, and that roads are built in straight lines. What is the best location for the airport, so that the total length of road to be constructed is minimized?

2. The Code

We place the cities on a two-dimensional grid. We store the x and y coordinates of the cities in vectors. The airport is to be located at one of the lattice points on the grid. A function is written to compute the total road length given an airport location. The code is given below. Note that Scilab underlines the functions in its editor.

```
clc
clear

// Four planar points, coordinates
// in the range of [0, 99]
x=[10, 80, 35, 36];
y=[42,  9, 91, 19];
// numerically find a point to place
// a new airport, such that the total
// distance from the cities to the airport
// is minimized.

// f(a,b) gives the total distance from
// points (xi, yi) to (a,b)
function [d]=f(a, b)
 d=0;
 for i=1:4
  d=d+sqrt((a-x(i))^2+(b-y(i))^2);
 end
endfunction

Z=zeros(100,100);
```

```
// generate objective function values
// and keep track of the best solution

bestZ=f(1,1);
bestX=1;
bestY=1;

for i=1:100
    for j=1:100
        Z(i,j)=f(i,j);
        if Z(i,j)<bestZ then
            bestX=i;
            bestY=j;
            bestZ=Z(i,j);
            end
    end
end

disp(bestX, bestY, bestZ);

[X,Y] = meshgrid(1:100, 1:100);
mesh(X,Y,Z)
```

Figure 11.1. The Code

A two-dimensional array (Z) is defined. Z(i,j) holds the objective function value for when the airport is to be located at point (i,j). Two nested loops, one for each dimension are used. The inner loop also keeps track of the objective function values. The minimum Z(i,j) value as well as the coordinates (indices I and j) for which the objective function attains its minimum, are displayed.

We must note that engineering software usually has many built-in functions to find the minimum or maximum value of a set of numbers. He we explicitly checked each value of the objective function and compared it to the best value so far. We did so explicitly to illustrate how such a search might be accomplished by a general programming language. See the exercises for a different implementation.

As an added feature, we create a three-dimensional plot. The function meshgrid() creates a two-dimensional matrix. The plot function mesh() is used to plot the objective function surface.

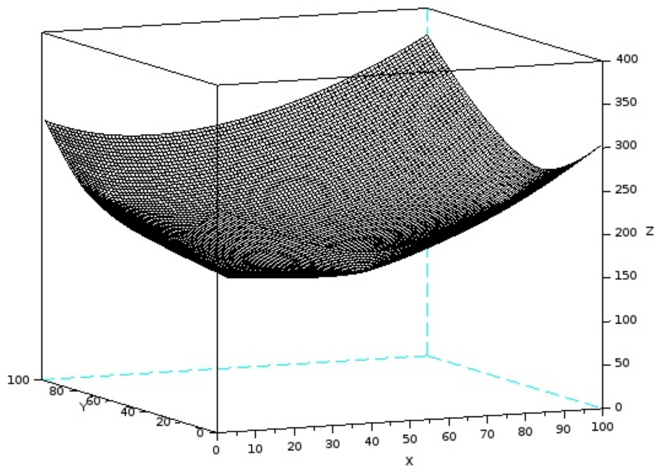


Figure 11.2. The Objective Function Surface.

For the given city coordinates, the best airport location is computed to be (30,36) while the total road length is approximately 150.

3. Exercises

1. Assume that the objective is to minimize the distance to the farthest city. Note that such objective functions are useful in minimizing the worst case. For example, if we were to place a fire station in proximity of a few neighborhoods, rather than minimizing the total road length, it would make more sense to minimize the farthest neighborhood.
2. In this example, all cities are placed on integer lattice points. Similarly, we only seek airport locations on integer lattices. Modify the code so that the grid resolution is higher. That is, the airport may be located between lattice points.
3. Rather than checking each objective function value against the best so far, use a built-in function to find the minimum value of the mesh (X,Y,Z).
4. A river runs through a 100 by 100 mile grid. The course of the river is given as N = 6* sqrt(E), where E is the east direction and N is the north direction (a bit like x and y). A city is located at coordinates [20,50]. A port is to be built on the river. Find the location on the river that is closest to the city.

XII. COMBINATORIAL OPTIMIZATION: THE KNAPSACK PROBLEM

The inventory model and the location model discussed in the previous chapters are good examples of industrial engineering optimization efforts. These two models involve smooth functions, albeit in our computations, we look at discrete points in the respective domains. For example, we search for the best airport location among points whose coordinates are integers. The student must have understood that seeking any point on the plane is also possible. This may require different techniques, but nonetheless possible. Our approach is a computational one. As an engineering approximation, we understand that finding the best airport location among lattice points is sufficient.

There is another important point to be made about these previous models. Consider the inventory model. The objective function (cost per time) as a function of the decision variable (order quantity) is a smooth function. It can be seen from the graph that shows total cost per time as a function of the order quantity, that small changes in the decision variable makes small changes in the objective function value. This concept is called “locality” which means if you are in the vicinity of the optimum solution, so will the objective function be in the vicinity of the optimum value.

Combinatorial optimization is somewhat different. It deals with sets and subsets, the inclusion or exclusion of the elements in these sets. Solutions are usually defined as sets. As such, the objective function is not a smooth function of the decision variables. Rather, there is a prescribed way of enumerating the objective function value for a given solution (set). Understandably, changing one element of an optimum solution set does not necessarily give an objective function value in the vicinity of the optimum. This makes combinatorial optimization problems qualitatively different. It also makes computations more demanding.

1. An Example

The *knapsack problem* is well known in industrial engineering. Many industrial engineering problems are formulated as knapsack problems. Moreover, the knapsack problem encapsulates all of the important aspects of combinatorial optimization in a concise manner.

Consider a knapsack to be filled with items. Each item has a specified weight. Each item also has a specified utility value. The knapsack has a weight limit, which we call its capacity. The problem is to determine which items to put into the knapsack in order to maximize the total value of the items chosen. Meanwhile, the total weight of the items must not exceed the given capacity. From a set theoretic viewpoint, the set of all items is given. We are asked to select a subset, so that the capacity constraint holds and the total value is maximized.

We first compute a solution by total enumeration. Consider the following code.

```
clc                  // clear the console
clear

w=[90 12 32 22 14 31];    // weight
v=[ 1 14 41 53 24 47];    // value
c=48;                     // capacity
n=[1 2 3 4 5 6];          // items

// number of possible solutions (ask why?)
count=2^6;
tv=[];              // solutions: total value
tw=[];              // solutions: total weight
s=zeros(count,6);

for i=1:count;
 k=i;
 tv(i)=0;
 tw(i)=0;
 for j=1:6
    s(i,j)=pmodulo(k,2);
    k=floor(k/2);
    if s(i,j) then
      tw(i)=tw(i)+w(j);
      tv(i)=tv(i)+v(j);
    end
 end
 if tw(i)>c then tv(i)=-1; end;
end

// start with solution 1 as the best
best_index=1;
best_value=tv(1);

for i=2:count
 if(tv(i)>best_value) then
  best_index=i;
  best_value=tv(i);
 end
```

```
end

disp("--- solution ---")
printf("items loaded: ");
for j=1:6
    if s(best_index,j) then
     printf("%d ", j);
    end
end
printf("\n");
printf("best value  : %d\n", best_value);
printf("total weight: %d (c=%d)\n",.. tw(best_index),
c);
```

Figure 12.1. The Code

The code gives an example with 6 items to be placed in a knapsack with capacity 48. The vector w contains the weights of the items. Similarly, the vector v contains the values. We generate all possible combinations. For this, consider that there is a separate decision for each item: to be placed in the knapsack or not. Thus, the total number of combinations is 2^6. We set the variable count=2^6. Similarly, a solutions array s is used. The array has count=2^6 rows and 6 columns. Each row of s corresponds to a possible combination. We compute combinations at row k successively dividing k by 2 and checking if it is odd or even. The Scilab function pmoluo() is used for this purpose. If odd, we assume that the item is to be loaded. In this case, we add its value and weight to vectors tv and tw, respectively. If the weight exceeds the capacity, we mark the value tv(i) as -1, so that it is never selected as a solution.

In a separate loop, we scan the values and identify the best one. Its index is also saved. Finally, the solution is printed out.

```
 --- solution ---
items loaded: 2 4 5
best value  : 91
total weight: 48 (c=48)
```

Figure 12.2. Code Output.

2. Exercises

1. The given example code fixes the number of items to 6. Modify the code so that the number of items is defined by a variable (N) at the top of the code. Generate random value and weights in the range of 1 to 100. Run the code for various values of (N). Measure how long it takes to find a solution. What is the practical limit of (N)?
2. Consolidate the given example code to keep track of the best value and the index to which this corresponds in the first loop. This way, the second loop may be eliminated. Does this improve run time?
3. Modify the code to print out alternative solutions, when they exist.

XIII. HEURISTICS FOR COMBINATORIAL OPTIMIZATION: THE KNAPSACK PROBLEM

We continue our discussions on the knapsack problem that was introduced in the previous chapter. As we observe, the exact solution by total enumeration requires the evaluation of 2^N different subsets. The computational difficulty of this approach grows exponentially with N. Note that 2^N eventually grows more rapidly than any polynomial in N, e.g. N^2 or N^3. Case in point, 2^N surpasses N^2 at N=5, and N^3 at N=10. Afterwards, the exponential quickly outstrips the polynomials. At N=20, for example, while N^2=400 and N^3=8000, 2^N reaches the value of 1,048,576. This means that sooner or later, we will reach a practical computational limit on N.

The good news is that in engineering applications, it often suffices to find good solutions, if not the best (optimum). Moreover, engineering applications usually concern physical systems, where the problem reflects this physical structure. For instance, it was mentioned that quantities in engineering, unlike abstract mathematics, almost always have units. The knapsack problem is a good example of this. We want to maximize the value packed into the knapsack with items of varying value and weight. This already suggests solution techniques that would yield better solutions than randomly selecting the items. Such intuitive solution techniques are called “heuristics.” We present such a heuristic in this Chapter.

1. A Heuristic for the Knapsack Problem

Since we are interested in packing as much value into the knapsack, we would like to pick items with high value but low weight. Both of these desires, low weight and high value, could be addressed if we compute the value-per-weight measure for each item. Then, we may proceed by stuffing the knapsack with items in decreasing order of value-per-weight, until the capacity is reached. Note that, although a sensible one, this procedure is not guaranteed to yield the optimum solution. Consider, for example the case of four items with weight and values given as follows.

Item	Weight	Value	Value/Weight
1	4	12	3
2	4	11	2.75
3	3	8	2.67
4	3	8	2.33

Figure 13.1. An Example.

If the knapsack has capacity 10, and we apply our heuristic, we pick items 1 and 2, since they have the highest value per weight. Their total weight will reach 8, and thus leave no room for the other items. The total value packed would be 12+11=23. This is clearly not the optimal solution. Packing items 1, 3, and 4 would result in a value of 28 and a combined weight of 10.

2. The Code

We now give sample code that implements the heuristic.

```
clc                    // clear the console
clear
N=30;
w=floor(rand(N,1)*10+1);     // weight
v=floor(rand(N,1)*100);      // value
c=72;                        // capacity

disp("--- heuristic ---")
disp("--- v ---")
disp(v)
disp("--- w ---")
disp(w)
r=v./w;  // element-wise operation
disp("--- r ---")
disp(r)

disp("--- included indeces ---")
l=0;
value=0;
for k=1:N   // repeat N times

  max_index=1;
  max_ratio=r(1);

  for i=2:N
```

```
    if(r(i)>max_ratio) then
        max_ratio=r(i)
        max_index=i;
     end
  end

  if (l+w(max_index)<=c) then
    l=l+w(max_index);
    disp(max_index);
    value=value+v(max_index);
  end
  r(max_index)=-1;
end

disp("--- value ---")
disp(value)
disp("--- load ---")
disp(l)
```

Figure 13.2. The Code.

The code solves the heuristic for N items. The item weights and values are randomly generated and held in vectors w and v, respectively. The vector r computes the value-to-weight ratio. The main loop is executed N times. At each iteration, we find the item with the highest value-to-weight ratio, and place it into the knapsack, provided that there is enough capacity left. Once an item is placed, or considered but disregarded, due to its weigh, we mark its ratio as -1. This prevents the item from being considered during the future iterations, since all N items have nonnegative ratios.

The code runs in reasonable time for N up to a few thousand in a few seconds.

3. Exercises

1. Run the heuristic for different values of N and measure the computation times. Plot the time the heuristic takes to solve problems as a function of N. What type of function is this?
2. Combine the code with the one given in the previous chapter. Solve a number of randomly generated knapsack problems, both to optimality, and using the heuristic. Find out how often the heuristic finds the optimum solution.
3. (Difficult) Are there special cases where the heuristic is guaranteed to find the optimum solution to the knapsack problem?

XIV. COMBINATORIAL OPTIMIZATION: THE TRAVELING SALESMAN PROBLEM

A salesman is to conduct a tour to visit a set of cities. Each city is to be visited once and only once. The salesman is to complete the tour by returning to the starting point. Given a set of N cities, say {1, 2, 3,.. ,N} it is easy to schedule such a tour. In fact, any permutation of the numbers 1 to N would be a tour. Since the salesman may start at any city, we can fix the starting point, say city 1. This gives us (N-1)! possible tours. The twist comes when city-to-city distances are of concern. If the salesman wants to complete the tour in the shortest possible total travel distance, then the problem becomes very difficult. In fact, one is almost forced to evaluate all (N-1)! tours and find the shortest. As the knapsack problem that required in the order of 2^N evaluations, the Traveling Salesman Problem (TSP) has a computational difficulty that surpasses any polynomial-time solution. For, N! will surpass any polynomial in N (say N^k for any k>1). Even N^{10}, which grows quite rapidly, is surpassed by N! at N=15.

The TSP is of interest to industrial engineers, since many scheduling problems reduce to the TSP. Consider a job shop that is to paint N automobiles, each a different color. Although each paint operation takes the same time, setting up the equipment depends on the colors. If a white automobile is to be painted after a black one, cleaning the paint guns takes more time compared to the case where a black automobile is to be painted after a white one. This is because a little dark paint left in the paint gun will show in the lighter color, while the effect in the reverse case will go unnoticed. Scheduling these N automobile paint jobs is like traveling through N cities, where each city is to be visited once.

1. The Code

The following code solves a TSP by total enumeration.

```
clc                         // clear the console
clear

N=4;                        // number of cities

// random [0,100] distance matrix
// from row r to column c
d=round(rand(N,N)*100);
disp("--- random distance matrix ---")
disp(d);

c=[1:N];              // cities
a=perms(c);           // permtations of cities

// number of possible solutions
count=factorial(N-1);

t=[];                 // total tour lengths

disp("--- possible tours : lengths ---")
best_tour=0;
min_length=%inf;

for i=1:count;
// compute tour length for tour i
 t(i)=d(a(i,N),a(i,1));
 for j=1:N-1
  t(i)=t(i)+d(a(i,j),a(i,j+1));
 end
 if(t(i)<min_length) then
     min_length=t(i);
     best_tour=i;
 end

disp([t(i), a(i,:)]);
end  // end of tour i

disp("--- solution ---")
printf("min tour length: %d\n", min_length);

disp(a(best_tour,:));
```

Figure 14.1. The Code

A random distance matrix is generated with integer values in the range of [0, 100]. The vector c contains the city indices, that is, c=[1, 2, ...N]. The array a holds all permutations of the city indices. Each row of array a is a permutation of the indices in c. All (N-1)! tours are evaluated and the corresponding tour lengths stored in vector t. As we evaluate tour lengths, we keep track of the shortest tour by saving its index in best_tour and its length in min_length. The code terminates by displaying the solution.

2. Exercises

1. Find examples that can be modeled as a TSP in everyday life.
2. Change N and make a few runs. Plot the average time it takes to solve the TSP as a function of N. What is a practical computational limit to N?

XV. A HEURISTIC FOR THE TRAVELING SALESMAN PROBLEM

The TSP is a computationally difficult problem. As the number of cities (N) increases, the effort to find the optimum solution becomes prohibitively expansive. Hence, one often resorts to heuristical solutions. A most intuitive heuristic is the so-called “nearest neighbor” approach. The idea is simple: start with a city, and go to the nearest city. At each step, find the nearest city not yet visited and go there next. Although this heuristic may not always find the optimum solution, it nonetheless finds good solutions, usually much better than a simple random ordering.

1. The Code

In an attempt to make the code more understandable, we make use of two functions. The function Visited(tour, i) takes a vector (tour) and an index (i). If the index is an element of the vector (tour), then the function returns 1, representing the Boolean value “TRUE”. Otherwise, it returns the value 0, representing “FALSE”.

```
// this function returns true (1)
// if city i is already in the tour
// else it returns false (0)
function result=Visited(tour, i)
result=0; // assume not visited
for j=1:length(tour)
 if(tour(j)==i)
  result=1;
  break;
 end;
end
endfunction
```

Figure 15.1. The Function “Visited()”.

The next function GetNearestCity(tour, d) takes a vector (tour) that contains the indices of the cities in the partial tour, and a distance matrix (d). It returns the index of the unvisited city that is nearest to the last city in the partial tour. The number of cities is deduced from the size of the distance matrix (d).

```
// this function returns the nearest
// unvisited city from the last city
// in the tour
```

```
function NearestCity=GetNearestCity(tour, d)
LastCity=tour($);
NearestCity=0;  // to be replaced
DistanceToNearest=%inf;

for i=1:size(d, 'r')
 if ~Visited(tour, i)
  if(d(LastCity, i)<DistanceToNearest)
   DistanceToNearest=d(LastCity,i);
   NearestCity=i;
  end
 end
end
endfunction
```

Figure 15.2. The Function “GetNearestCity()”.

Using functions that meaningfully encapsulate a logical task is central to structured programming. This not only improves code readability, but it positively affects code performance. It also facilitates code re-usability, since general-purpose functions may be imported to other code applications. In fact, writing general-purpose functions and combining these in libraries are common practice in software.

With the aid of the two functions, the main code becomes quite straightforward.

```
N=100;                  // number of cities

// random [0,100] distance matrix
// from row r to column c
d=floor(rand(N,N)*100);

tour=[1];    // start from city 1
for i=2:N
j=GetNearestCity(tour, d);
tour(i)=j;
end

// compute tour length for the tour
 t=d(tour(N),tour(1));
 for j=1:N-1
  t=t+d(tour(j),tour(j+1));
 end
```

```
// print city sequence for the tour and the tour length
 disp(tour);
 disp(t);
```

Figure 15.3. The TSP Heuristic Main Code.

The code runs for quite large problems, several hundred cities, in a few seconds. An exact solution by total enumeration for even 100 cities would be impossible, as 100! is about 10^{158}. Considering that the universe is about 10^{17} seconds old, and that the fastest current supercomputer can execute about 50 PLOPS (50×10^{15} floating-point instructions per second), it would take the fastest computer today 10^{124} times the age of the universe to go through all possible 100-city tours. Note that 10^{124} is a very large number indeed.

10^{124}=10,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000.

Our supercomputer must run this many times the age of the universe to evaluate all 100-city tours.

2. Exercises

1. Plot the tour length obtained from the nearest neighbor heuristic as a function of N.
2. Use the code to evaluate how well the nearest neighbor heuristic performs compared to random tours. First evaluate the length (R) of the random tour (the sequence 1, 2, 3,..N), and then evaluate the tour length obtained from the heuristic (H). Plot the ratio H/R as a function of N. Suggest a functional form for this curve.
3. Find out how fast the nearest-neighbor heuristic works as a function of the number of cities. Generate a several random problems with N in the range of 100 to 1000 and plot the CPU times against N.
4. (Difficult) Find information on other TSP heuristics and code these.

XVI. THE ASSIGNMENT PROBLEM

Not all combinatorial industrial engineering problems are difficult. A case in point is the so-called assignment problem. The problem tries to assign N jobs to N different machines. The cost of each job on each machine is known. The objective is to assign each job to a unique machine so that the total cost is minimized.

Although at first glance, it is a combinatorial problem. There are N! different ways we can order the jobs. Each ordering corresponds to an assignment. Nonetheless, compared to the TSP, the assignment problem has a rather simple solution. The best known algorithm for the assignment problem is the “Hungarian Algorithm.” Its computational difficulty is of polynomial order. More specifically, given N jobs, the number of computations required to find a solution is in the order of N^4 (recent improvements have further reduced the order). This order of complexity is much better than total enumeration by considering all N! assignments.

The reason why some combinatorial problems are computationally difficult (exponential-time algorithms) while others are not (polynomial-time algorithms) is intuitively addressed by considering the structure of the problem. The assignment problem has a polynomial-time algorithm. Also note that the assignment problem has a lot more structure as well. The number of jobs and the number of machines is equal. Each job is assigned to one machine, and each machine is assigned only one job. Such restrictions limit the feasible region of the decisions, and hence serve to limit the complexity of the solution process.

1. The Code

We use total enumeration to solve the assignment problem. Although total enumeration is way overkill, it is simple to code. Unfortunately, as N gets larger, total enumeration becomes computationally prohibitive.

```
clc                 // clear the console

// cost matrix, job at row r is assigned
// to machine at column c
c=[1 2 3 4 5;
   1 4 3 5 4;
   2 1 5 2 1;
   6 3 2 5 6;
   6 3 5 2 4];
```

```
j=[1 2 3 4 5];          // jobs
a=perms(j);             // permtations of jobs
count=factorial(5);     // number of solutions
s=[];                   // assignment costs

disp("--- assignment costs ---")
disp(c)

disp("--- possible assignments : costs ---")
best_index=1;
best_cost=%inf;

for i=1:count;
 s(i)=0;
 for j=1:5
  s(i)=s(i)+c(j, a(i,j))
 end

 if(s(i)<best_cost) then
  best_cost=s(i);
  best_index=i;
 end
 disp([a(i,:), s(i)]);
end

disp("--- solution ---")
printf("best cost: %d\n", best_cost);

// display alternative solutions
for i=1:count
 if(s(i)==best_cost) then
  disp(a(i,:));
 end
end
```

Figure 16.1. The Code.

The code works well for small N. In this case (N=5) the solution is readily computed in a fraction of a second.

```
--- assignment costs ---

    1.    2.    3.    4.    5.
    1.    4.    3.    5.    4.
    2.    1.    5.    2.    1.
    6.    3.    2.    5.    6.
    6.    3.    5.    2.    4.
 --- solution ---
best cost: 8

2.    1.    5.    3.    4.
```

Figure 16.2. The Output.

The solution indicates that jobs 1, 2, 3, 4, and 5 are assigned to machines 2, 1, 5, 3, and 4, respectively.

Job	Machine	Cost
1	2	2
2	1	1
3	5	1
4	3	2
5	4	2
Total		8

Figure 16.3. The Optimal Assignment.

2. Exercises

1. Rather than the costs, let the assignments yield different profits on different machines. That is, instead of a cost matrix, we now consider a profit matrix. Modify the code to find the maximum profit assignment.
2. Let the cost matrix give the machining times for each job on each machine. Once the assignment is made, let all machines start at the same time. Find the assignment that minimizes makespan. That is, minimize the time it takes all jobs to finish.
3. Modify the task in Exercise 2 to show all possible alternative solutions.
4. (Difficult) Investigate the Hungarian Algorithm to solve the assignment problem. Code the algorithm and investigate the performance advantages over the present code.

XVII. SCHEDULING: THE SINGLE MACHINE CASE

Scheduling problems in industrial engineering often lead to combinatorial optimization. The simplest scheduling problem is perhaps the single machine, N jobs case. Here, we have N jobs, each associated with a processing time, available at the start. The machine is to process all jobs. The decision involves the sequence in which the jobs are to be processed.

Let us also define a few other keywords. The so-called “flowtime” for a job is the total time it spends in the system. First the job waits for its turn on the machine, and then, it undergoes the operation. The latter is the processing time for the job, which is assumed to be constant and known. Sometimes jobs have “due dates”. The difference between the due date and the completion time is called lateness. Lateness can be positive (the job is finished after the due date) or negative (the job is finished before the due date), or zero (the job is finished exactly on the due date). Tardiness is also used to indicate if a job is late. If the job finishes on or before its due date, then tardiness is zero. If the job is finished after its due date, then tardiness is the same as lateness. If a job is completed before its due date, than the difference between the due date and the completion time is called the “earliness”.

1. Minimize Average Flowtime

The code given below defines jobs with processing times. As a brute-force attempt, we once again consider all possible sequences of the jobs. That is, N! possible schedules. Note that the total processing time does not change for these different schedules. The total processing time is computed once and saved as a global variable (tpt). A global variable is a variable that is visible to all parts of the code, including the functions.

```
clc                           // clear the console
clear
N=6;                          // number of jobs
MaxP=100;                     // max processing time
p=ceil(rand(N,1)*MaxP);       // processing times
tpt=0;                        // total process time

disp(p);
```

Figure 17.1. Generating Random Jobs.

For each, we compute an average flow time. The function

GetFlowtime() is defined to simplify the computations.

```
function ft=GetFlowTime(s, k)
ft=0;
    for j=1:N
        ft=ft+p(s(j));
        if s(j)==k then break; end
    end
endfunction
```

Figure 17.2. Computing the Flowtime of Job k for Schedule s.

The function takes two arguments. The first (s) is a vector of job indices in a given sequence. The sequence determines the schedule. The second argument is the index of the job whose flowtime is to be computed. Note that the number of jobs (N) and the processing times (p) are defined as global variables before the function definition. Thus, the function already has access to these variables.

We next define the jobs, the number of schedules, and a permutation matrix (a) whose rows are the possible schedules (permutations of job indices). The total processing job (tpt) was defined as a global variable with an arbitrary value. Here we compute its value and assign it to the variable.

```
jobs=[1:N];
a=perms(jobs);          // permutations of jobs
count=factorial(N);     // number of schedules
f=[];                   // average flow time

for i=1:N
tpt=tpt+p(i);
end
```

Figure 17.3. Computing the Flowtime of Job (k) for Schedule (s).

The remainder of the code is quite straightforward. We compute the average flow time for each schedule and keep track of the best average in the process of doing so. The code terminates with displaying the schedule that yields the lowest average flowtime.

```
// ---compute the average flowtime ---
best_ave_flowtime_index=0;
best_ave_flowtime=%inf;
```

```
for i=1:count;
 f(i)=0;
 for j=1:N
  f(i)=f(i)+GetFlowTime(a(i,:), j);
 end
 f(i)=f(i)/N;

 if (f(i)<best_ave_flowtime) then
   best_ave_flowtime=f(i);
   best_ave_flowtime_index=i;
  end
end

disp("--- solution ---")
disp(a(best_ave_flowtime_index, :));
disp(best_ave_flowtime);

disp("flow times: ");
 for j=1:N
  printf("%6.2f ",..    GetFlowTime(..
     a(best_ave_flowtime_index,:),..
     j));
 end
```

Figure 17.4. Finding the Schedule with Best Average Flowtime.

As our approach looks at all possible schedules, once again, the computational effort is quite high. We must look at N! possible schedules and select the best among them. The output of the code is shown below.

```
    29.
     9.
    63.
    35.
    71.
    53.

 --- solution ---

    2.    1.    4.    6.    3.    5.

    115.83333
 flow times:
38.00   9.00 189.00  73.00 260.00 126.00
```

Figure 17.4. Code Output.

There are 6 jobs with randomly generated processing times. The processing times are 29, 9, 63, 35, 71, and 53. The sequence that minimizes the average flowtime is 2-1-4-6-3-5. The minimum average flow time is computed as 115.83.

2. Exercises

1. You may notice that the sequence 2-1-4-6-3-5 that minimizes the average flowtime in the example also follows a certain logic. It is in the order of processing times, from the lowest to the highest. In scheduling, this sequence is called the “Shortest Processing Time” ordering, or SPT for short. Run several examples and observe if SPT minimizes the average flowtime in each trial.
2. Modify your code to generate random due dates. Then compute the average tardiness for each schedule. Does SPT also minimize average tardiness?
3. With randomly generated due dates. compute the maximum tardiness for each schedule. Does SPT minimize the maximum tardiness among jobs?
4. In Exercise 2 above, check if ordering the jobs according to their due dates minimizes the maximum tardiness.
5. (Difficult) Return to the problem of minimizing the average flowtime for N jobs. Suppose that we have two machines. In general, we would have different processing times for a given job on each machine. To start with, let us assume that the job processing times are the same for each machine. Write code to generate all possible schedules. Note that each job is processed either on machine 1 or machine 2. Among the possible schedules, find the one that minimizes the average flowtime.
6. (Difficult) Repeat Exercise 5, but with due dates. Find the schedule that will minimize the maximum tardiness.

XVIII. NETWORK MODELS: SHORTEST PATH

A network is a collection of nodes and arcs. A network is also called a graph, and hence the term “graph theory” in mathematics and operations research. Similarly, nodes and arcs are also known as vertices and edges. The nodes and the arcs may have associated attributes. Many industrial engineering models involve networks. For example, activities with precedence relations are easily modeled by networks. This becomes handy in project management. The Traveling Salesman Problem discussed in the previous chapters is also defined on a network of cities. Similarly, decision trees and probabilistic “what if” scenarios may be modeled as networks. A flowchart is essentially a network, where the nodes contain information about the operations performed. Even the assignment problem we discussed may be considered as a network problem. Here, the network connects elements of a set of jobs to the elements of a set of machines.

Here, we consider a simple network, one that is actually a map of cities and roads. The arc attributes are the lengths of the roads. We pick an origin and a destination. We want to find the best path to travel from the origin to the destination. Choosing the objective to be to minimize travel distance is reasonable. This problem is called the “shortest path” problem in networks. It is a fundamental problem that we encounter a lot in industrial engineering applications. For instance, the related problem, the “longest path” determines project completion time if the nodes are the various stages of the project, and the arc lengths represent time to reach a stage from a previous stage.

1. An Example

Consider the following example. Here we have 6 cities. The numbers associated with the arcs are the road lengths between the cities. Note that we use undirected arcs. In this case, the distance between adjacent cities is the same in either direction. It is also possible to consider different lengths for opposite directions. This is the case in air travel, where a tail wind or a head wind makes a difference.

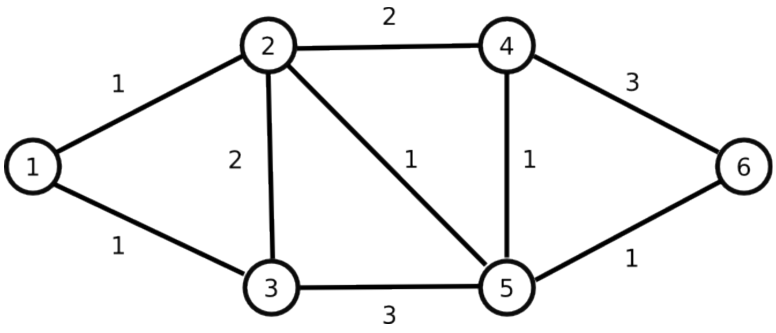


Figure 18.1. The Network (Origin 1, Destination 6).

Here, we want to find the path from the origin (City 1) to the destination (City 6) that has the shortest distance. The shortest path is easily detected by inspection. However, when the number of nodes and arcs increase, the task of finding the shortest path may no longer be so trivial.

2. An Algorithm

Dijkstra'a algorithm is used to find the shortest path. The algorithm is very intuitive. We shall use it without proving that it works. Throughout the algorithm, we work with two sets: the visited nodes, and the unvisited nodes. The two sets are complements of each other. In the beginning, the visited nodes contain only the origin. All other nodes are unvisited. Each node has a distance from the origin. This distance is zero for the origin. In the beginning, the distance from the origin is infinity for all unvisited nodes.

At each iteration, we consider all arcs from visited nodes to unvisited nodes. Essentially, we are considering how we can go to an unvisited node, from one of the visited nodes. Now, each visited node has a distance associated with it. It measures how far that visited node is from the origin. Suppose we travel on an arc i to j, where i is a visited node and j is an unvisited node. By doing so, we will have visited previously unvisited node (j). We can compute the total distance from the origin to this newly visited node (j) by adding to the arc length to the distance from the origin to node (i). We do not immediately take an arc, but consider all possible arcs i to j, where node i is visited and node j is unvisited. For each case, we compute and tentatively assign to the nodes j, their computed total distance from the origin. Among all possible arcs, we pick the one that takes us to a node j, whose total distance from the origin is the smallest. Intuitively, we next visit that previously unvisited node which is the closest to the origin.

Thus, at each iteration, we add one more to the set of visited nodes. This node is removed from the unvisited nodes. Given N nodes, since at each iteration, we visit another node, after N-1 iterations, the algorithm will terminate (not N but N-1, since the origin was already added to the set of visited nodes). The algorithm may be terminated sooner if we reach our destination, while some nodes remain unvisited.

3. The Code

We first define a distance matrix. Element (i,j) f the matrix is the distance from node i to node j.

```
clc                     // clear the console
clear

// distance matrix (symmetric)
d=[%inf    1    1 %inf %inf %inf;
      1 %inf    2    2    1 %inf;
      1    2 %inf %inf    3 %inf;
   %inf    2 %inf %inf    1    3;
   %inf    1    3    1 %inf    1;
   %inf %inf %inf    3    1 %inf];

disp(d);
N=size(d,1);
```

Figure 18.2. The Distance Matrix.

Note that in this case, the distance matrix is symmetric. The algorithm works equally well if the distance matrix is not symmetric.

```
origin=1;                // origin node
destination=6;           // destination node

visited=[origin];        // visited nodes

// unvisited nodes
for i=1:N
 if(i<>origin) then unvisited($+1)=i; end
end

// shortest distance from origin to the nodes
for i=1:N distance(i)=%inf; end
```

```
distance(origin)=0;
previous=zeros(N);
```

Figure 18.3. The Distance Matrix.

Next we specify the origin and the destination and construct the sets “visited” and “unvisited”. The vector “distance” whose elements give the distance from the origin to the nodes is initialized. Finally, the vector “previous” is initialized to all zeros. Element j of this vector will hold the city index i, if arc i-to-j is used to reach city j (that is, if arc i-to-j is on the shortest path).

After the initialization steps, we are now ready to implement the algorithm. The main loop executes as a new node is added to the set of visited nodes at each iteration.

```
while length(unvisited)
 best_i=0;
 best_j=0;
 best_distance=%inf;

 for i=1:length(visited)
  from=visited(i);
  for j=1:length(unvisited)
   to=unvisited(j);
   if distance(from)+d(from, to)..
            < best_distance then
       best_distance=..
              distance(from)+d(from, to);
       best_i=i;
       best_j=j;
   end
  end
 end
 from=visited(best_i);
 to=unvisited(best_j);
 previous(to)=from;
 distance(to)=best_distance;
 visited($+1)=to;
// remove node 'to' from unvisited,
// add it to the set visited
 unvisited(best_j)=[];
end
```

Figure 18.4. The Main Loop.

At each iteration, the main loop considers all possible arcs from visited nodes to unvisited nodes. The tentative total distance from the origin to the unvisited node is compute. The lowest such total distance is recorded (best_distance) as well as its source node (best_i) and its destination node (best_j). We also keep track of the node from which we have reached the newly visited node. The vector “previous” holds the index (best_i). The closest unvisited node is then removed from the set of unvisited nodes and added to the set of visited nodes. We run the algorithm until all nodes are visited.

```
// construct the path backwards
reverse_path=[destination];
prior=previous(destination);
while prior<>origin
 reverse_path($+1)=prior;
 prior=previous(prior);
end
reverse_path($+1)=origin;

path=reverse_path($:-1:1);
disp("shortest path");
disp(path);
disp("distance");
disp(distance(destination));
```

Figure 18.5. Constructing the Shortest Path.

When the main loop terminates, we have collected all the information needed to construct the shortest path. The vector “previous” gives the node from which we have reached a given node. Starting from the destination, we work backwards to construct the shortest path. The indices, working backwards, are stored in the vector “reverse_path”. This needs to be reversed to find and report the shortest path in the forward direction. The code terminates by reporting the shortest distance to the destination and the path by which to achieve this distance.

4. Exercises

1. Run the code for different distance matrices and different number of cities.
2. Generate random symmetric distance matrices and run the algorithm.

3. Generate random non-symmetric distance matrices and run the algorithm.
4. Run several random examples and plot average execution times versus the number of nodes, N.
5. Project Management involves a set of project stages that have a precedence relationship. The network model is similar to the example given in this chapter. However, the arcs are directed to indicate the precedences. The nodes represent the project stages. The arcs lengths are the time it takes from one project stage to the next. The project has an origin, as in this case, and a destination. The destination stage represents project completion. Unlike the example here, the project terminates after all stages are completed. The time to complete the project is thus the length of the longest path from the source to the destination. Modify the code to find the longest path, rather than the shortest one.

XIX. RECURSIVE NETWORK ALGORITHMS: FIND A PATH

Recursion is a powerful conceptual and practical concept. Simply stated, a function is recursive if it calls itself. Rather than a lengthy discussion, let us illustrate recursion with an example. Suppose you want to write a function that returns the factorial of a nonnegative integer. The following function is a straightforward implementation.

```
function nFact=getFactorial(n)
 nFact=1;
 for i=1:n
  nFact=i*nFact;
 end
endfunction
```

Figure 19.1. A Function to Compute Factorials.

The function implements a loop from 1 to the given integer n, and multiplies the series of number, 1,2,...n to find the factorial. Now, consider the following implementation.

```
function nFact=getFactorialRecursive(n)
 if(n<=2) then
     nFact=n;
    else
     nFact=n*getFactorialRecursive(n-1);
 end
endfunction
```

Figure 19.2. A Recursive Function to Compute Factorials.

The function in Figure 19.2 is a recursive function, since it calls itself. The logic is elegant. When the factorial of n is to be computed, if n is less than or equal to 2, then the factorial is simple n. Otherwise, the factorial of n is n times (n-1)!. In order to compute (n-1)!, the function calls itself. Thus the recursion.

If recursion appeals to you, then you have a good sense of engineering intuition. The advantages and disadvantages of both these approaches is an interesting topic of computer engineering, albeit outside the scope of our elementary textbook.

1. Find A Path From Source to Sink

We now use recursion in a more demanding application. Consider a network with undirected arcs through which there is a flow of given capacity. For example, this could be a road network, where the capacity denotes the number of vehicles that can travel through that arc in an hour. We use a “from-to”-type matrix to define the arc capacities. Element (i,j) of the capacity matrix is the capacity on the arc that goes from node i to node j. Note that if the arcs are undirected, then the capacity matrix will be a symmetrical matrix. If an arc does not exist between the nodes, then the corresponding capacity is simply 0.

The arc capacities are written as a matrix. Consider, for example, the following capacity matrix.

```
// capacity matrix (undirected/symmetric)
capacity=[  0    10    31     0     0     0;
           10     0    22    24    13     0;
           31    22     0     0    37     0;
            0    24     0     0    13    34;
            0    13    37    13     0    12;
            0     0     0    34    12     0];
```

Figure19.3. The Capacity Matrix.

The rest of the code relies on a single recursive function. Before we discuss the function, let us present the code.

```
origin=1;               // origin node
destination=6;          // destination node

visited=[origin];       // visited nodes

// unvisited nodes
unvisited=[];       // unvisited nodes
for i=1:N
 if(i<>origin) then
//      unvisited($+1)=i;
     unvisited=[unvisited,i];
 end
end

path=getPath(visited, unvisited,..
                       capacity, destination)

if(path($)==destination) then
  disp("path:");
  disp(path);
else
```

```
  disp("path not found!");
end
```

Figure19.4. The Main Code.

As seen, the code relies on the function getPath() to actually find a path from the origin to the destination. The code defines the origin and the destination as nodes 1 and 6, respectively. We make use of two vectors, namely “visited” and “unvisited”. These vectors partition the set of nodes. These vectors are initialized before we call getPath(). The vector “visited” initially includes only the origin, and the vector “unvisited” contains all nodes except the origin.

The function getPath() takes as arguments these two vectors, the capacity matrix, and the destination node. It returns the node indices of the path from origin to destination as a vector. The function is now given below.

```
function path=getPath(head, unvisited,..
                                   cap, dest)
 path=head;
 from=head($);
 for i=1:length(unvisited)
      to=unvisited(i);
      if(cap(from,to)==0) then continue; end
      if(to==dest) then
           path=[head,to]; // path found
           break;
       else
        newHead=[head,to];
        newUnvisited=unvisited;
        newUnvisited(i)=[];
        path=getPath(newHead, newUnvisited,..
                                   cap, dest);
        if(path($)==dest) then break; end
      end // if
 end
endfunction
```

Figure19.5. The Function getPath().

The function takes a partial path, named “head”, that starts from the origin. Any node that is not in the path is in the set “unvisited”. The capacity matrix and destination nodes are also specified. The function simply goes through the unvisited nodes and tries to append it to the partial path. These unvisited arcs are named “to” in the code. The last node in the partial path is named “from”. If the capacity of the arc (from, to) is zero, then the loop simply continues. In this case, the arc (from, to) does not exist, since the capacity is zero. If however, the arc exists, it is appended to the partial path. This augmented partial path is named “newHead”. Similarly, a new vector (newUnvisited) of unvisited nodes is formed, and the node “to” is removed from this vector.   If the node “to” is the destination node, we break out of the loop, since the path is now found. Otherwise, with a new partial path and a new vector of unvisited nodes, the function calls itself, and thus invokes a recursion. Again, if the end of the returned path is the destination, we are done. We break out of the loop. If not, we try another node as the “to” node and repeat.

You may have noticed that at the beginning of the function, we set path=head. This is done, so that, if no path is found, the function return value would be undefined. This way, the function returns the partial path. It is up to the calling program to check if the path terminates at node “destination”.

2. Exercises

1. Write a small program and implement the factorial functions given at the beginning of this chapter.
2. Change the capacity matrix and run the code. Also try a network where there is no path from the origin to the destination. Does the code run as intended?
3. Modify the function getPath() so that besides the path, it returns a Boolean variable which indicates whether or not a path has been found from the origin to the destination.
4. (Difficult) Write a program that finds a path from the origin to the destination but without the use of recursion. Compare the two versions, with and without recursion. Discuss the benefits of each approach.

XX. NETWORK MODELS: MAX FLOW

This chapter builds upon the previous chapter. We will use the code developed in the previous chapter to find the maximum possible flow from the origin to the destination. The method is referred to the Ford-Fulkerson algorithm in the literature. The idea is rather simple. We try to find a path, as in the previous chapter, from the origin to the destination. If such a path exists, we find the maximum flow along that path. As expected, the maximum flow is the minimum of the arc capacities along the path. We then remove the flow along the path from the capacities of the arcs on the path. A path with positive flow is called an “augmenting path”. We repeat until no further augmenting paths exist.

1. Support Functions

In addition to the function getPath() discused in the previous chapter, we make use of two other functions. The function getPathMaxFlow() takes the vector path whose elements are the indices of the nodes on the path, and the capacity matrix. The function traverses the path and returns the minimum capacity of the arcs on the path. Clearly, this is the maximum amount of flow we may put through the given path. The implementation is rather straightforward.

```
function flow=getPathMaxFlow(path, capacity)
 flow=%inf;
 from=path(1);
 for i=2:length(path)
  to=path(i);
  if(capacity(from, to)<flow) then
   flow=capacity(from, to);
  end;
  from=to;
 end
endfunction

function reducedCapacity =..
     getReducedCapacity(path, flow, capacity)
 reducedCapacity=capacity;
 from=path(1);
 for i=2:length(path)
  to=path(i);
  reducedCapacity(from, to) = ..
                  capacity(from, to)-flow;
  reducedCapacity(to, from)=reducedCapacity(from, to);
  from=to;
 end
endfunction
```

Figure 20.1. The Support Functions.

The next function updates the capacities of the arcs along a given augmenting path. The function getReducedCapacity() takes the vector "path", the "flow" by which the capacity matrix is to be modified, and the capacity matrix. It simply reduces the capacities of the arcs along the given path by the amount "flow".

2. The Main Program

The main program is given below. It consists of a loop that identifies an augmenting path, finds the maximum flow through that path, and updates the capacity matrix to reduce the capacities of the arcs along the augmenting path by the amount of flow. The loop runs until no further path from the origin to the destination is found.

```
// --- main loop ---
maxFlow=0;
found=1;
while(found)
 visited=[origin];
 unvisited=[];
 for i=1:N
  if(i<>origin) then
    unvisited=[unvisited,i];
  end
 end

 path=getPath(visited, unvisited,..
                          capacity, destination);
 found=(path($)==destination);
 if(found) then
      flow=getPathFlow(path, capacity);
      maxFlow=maxFlow+flow;
      capacity=getReducedCapacity(path,..
                                 flow, capacity);
 end
end
```

```
disp("Max flow:")
disp(maxFlow);
```

Figure 21.1. The Main Program.

The flow along the augmenting paths are cumulatively added to the value maxFlow, which is displayed at the end.

3. Exercises

1. Modify the main loop to display each augmenting path and its maximum flow.
2. Is it always possible to increase the maximum flow through a network by increasing the capacity of just one arc? Experiment and explain your findings.

XXI. LINEAR PROGRAMMING

Linear Programming is perhaps the best known modeling tool in optimization. Since its inception in the mid 50s, Linear Programming, or LP as it has come to be known, was responsible for much of the CPU time of the mainframes. The main concepts are easily demonstrated by an example.

Suppose we want to make two different types of cookies. The main ingredients in both types is essentially the same, albeit at different proportions. The following table shows how much of each ingredient, by weight, comprises each type of cookie.

	Cookie 1	Cookie 2	Price	Available
Sugar	3 kg	5 kg	2 TL/kg	260 kg
Butter	4 kg	3 kg	8 TL/kg	200 kg

The table also gives the cost of each ingredient and the available amounts on hand. Cookie 1 sells for 118TL per batch, and Cookie 2, for 106 TL per batch. How many batches of each type of cookie should the bakery make in order to maximize its profit?

1. The LP Model

We formulate the problem as an LP model. The following decision variables are defined.

X1: Number of batches of Cookie Type 1 to be baked

X2: Number of batches of Cookie Type 2 to be baked

X1 and X2 are called the decision variables of the LP. The objective is to maximize profit. The revenue is simply,

$$R=118*X1 + 106*X2.$$

The cost of the ingredients is computed as follows.

$$C=2*(3*X1+5*X2) +8*(4*X1+3*X2).$$

The objective is to maximize the profit (Z). Specifically,

$$Z=R-C=(118-6-32)*X1 + (106-10-24)*X2.$$

Simplifying the objective function, and desiring to maximize it, we write,

$$\max Z=80*X1 + 72*X2$$

There are two constraints, namely the availability of the two ingredients.

3*X1+5*X2 <= 260

4*X1+3*X2 <= 200

Each of the constraints above is associated with an ingredient. The first constraint, for instance, is related to the available sugar (260 kg). The left hand side of the inequality is the amount of sugar to be used, if we produce X1 and X2 batches of the types. The amount of sugar used cannot exceed the available amount on hand. The remaining constraint is for butter.

In addition, we write two non-negativity constraints.

X1 >= 0

X2 >= 0

These two constraints are needed to prevent the number of batches to be baked from assuming negative values.

2. A Naive Approach

A good engineer who has never seen an LP formulation may first consider a brute-force approach. Considering the available sugar, we have enough to produce 260/3 (fewer than 90) batches of Cookie 1, and 260/5 (fewer than 60) batches of Cookie 2. Similarly, the available butter limits us to 50 batches of Cookie 1 and fewer than 70 batches of Cookie 2. So, a most naïve approach would be to try all combinations of (X1, X2) for X1 and X2 in the range [0, 100]. For each combination, we would first check if the combination is feasible. That is, if the specified number of batches can be baked with the available ingredients. If feasible, we would compute the objective function value. Then, among all feasible solutions, we would pick the one with the highest objective function value.

The code is shown below.

```
function result=feasible(a, b)
 result=0;
 if a < 0 then return; end
 if b < 0 then return; end
 if 3*a+5*b > 260 then return; end
 if 4*a+3*b > 200 then return; end
 result=1;
endfunction

function result=objective(a, b)
```

```
 result=80*a+72*b;
endfunction

Z_grid=meshgrid(0:100,0:100);
best_X1=0;
best_X2=0;
best_Z=0;
for X1=0:100
    for X2=0:100
      if ~feasible(X1, X2) then continue; end;
      Z=objective(X1,X2);
      if Z>best_Z then
          best_Z=Z;
          best_X1=X1;
          best_X2=X2;
          end
    Z_mesh(X1+1,X2+1)=Z;
    end
end

printf("X1:%d X2:%d Z:%d\n",..
             best_X1, best_X2, best_Z);

mesh(Z_mesh);
```

Figure 21.1. The Code

Following the practice of structuring our code with the use of functions, we separate the feasibility check and the computation of the objective function value. These two are implemented as individual functions. This structure also makes the code somewhat easier to modify in the future. The main loop simply iterates X1 and X2. First we check the feasibility of the (X1, X2) combination. If feasible, we compute the objective function value. The best objective function value as well as the X1 and X2 that give this objective function value are kept. The code prints the best solution upon termination.

```
X1:20 X2:40 Z:4480
```

Figure 21.2. The Output

The solution indicates that we must produce 20 batches of Cookie 1 and 40 batches of Cookie 2. This combination yields the highest profit, 4480 TL.

The graph shows how the objective function behaves over the feasible solution space.

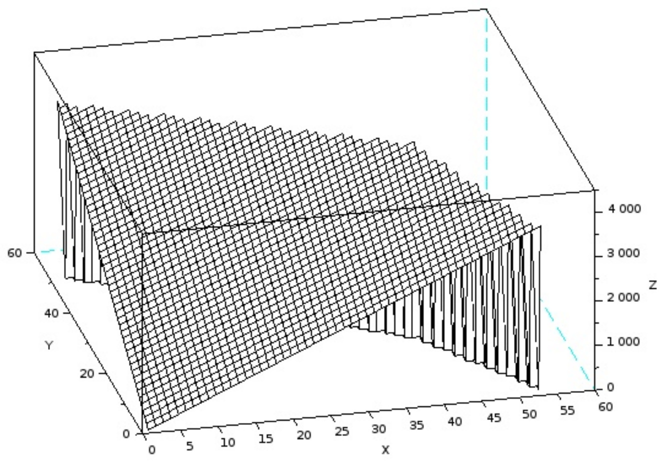


Figure 21.3. The Objective Function Value over the Solution Space.

3. A Smarter Approach

As discussed, a good engineer is expected to solve the problem, albeit in a naïve brute-force manner. This is a good illustration for the motivation for this book: develop computational skills to solve practical problems and gain further insights. There are insights we could develop from this exercise. In any case, LP will be covered in some depth in almost all industrial engineering curricula. It is seen that if there is a solution, it should occur on the boundary of the feasible solution space. Armed with this information, instead of checking all combinations of the decision variables, we could limit our checks to the extreme points. The extreme points are where the inequalities hold tightly. We have four inequalities and two unknown decision variables.

3*X1+5*X2 <= 260

4*X1+3*X2 <= 200

X1 >= 0

X2 >= 0

When we force the inequalities to hold tightly, we have an equation rather than an inequality. Any two equations of the four would uniquely determine the values of the decision variables X1 and X2.

For example, the two inequalities

4*X1+3*X2 <= 200

X1 >= 0

become

4*X1+3*X2 = 200

X1 = 0

which yield the solution X1=0 and X2=200/3=66.67. There are a total of four-choose-two or six different ways we can solve for the decision variables. Each one of these solutions would be at an extreme point. In matrix form, we have an overdetermined set of linear equations.

$$\begin{bmatrix} 3 & 5 \\ 4 & 3 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X1 \\ X2 \end{bmatrix} = \begin{bmatrix} 260 \\ 200 \\ 0 \\ 0 \end{bmatrix}$$

The coefficient matrix is a 4-by-2 matrix. The right-hand-side vector is of size 4. We can solve for the decision variables by considering any two rows of the coefficient matrix, and the corresponding elements in the right-hand-side vector.

However, not all extreme points need to be feasible. We still must do a feasibility check for potential solutions. The following code illustrates the approach. We define the coefficient matrix (c) and the right-hand-side vector (rhs). Two nested loops pick out rows of the coefficient matrix and place them in matrix B. Similarly, the corresponding elements of the right-hand-side vector are placed in vector R. The reduced system of linear equations thus has two equations in two unknowns.

```
clc
clear

c=[3, 5; 4, 3; 1, 0; 0,1];
rhs=[260, 200, 0, 0];

function result=feasible(a, b)
 result=0;
 if a < 0 then return; end
 if b < 0 then return; end
 if 3*a+5*b > 260 then return; end
 if 4*a+3*b > 200 then return; end
 result=1;
endfunction

function result=objective(a, b)
 result=80*a+72*b;
endfunction

N=size(rhs,'c');

best_X1=0;
best_X2=0;
best_Z=0;

for i=1:N-1
  for j=i+1:N
      B(1,:)=c(i,:);
      B(2,:)=c(j,:);
      R(1)=rhs(i);
      R(2)=rhs(j);
      // solve the set of linear equations
      X=(B^-1)*R;
      disp(X);
      if ~feasible(X(1),X(2) then continue; end
      Z=objective(X1,X2);
      if Z>best_Z then
          best_Z=Z;
          best_X1=X1;
          best_X2=X2;
          end
  end
end

printf("X1:%d X2:%d Z:%d\n",..
               best_X1, best_X2, best_Z);
```

Figure 21.4. Scanning the Extreme Points.

The solution is obtained by inverting the matrix B and multiplying with the (reduced) right-hand-side vector. This is not the best approach when the size of the matrix is large, but in this case with only two unknowns, it can be justified for its simplicity. Each solution is checked for feasibility, and the main loop keeps track of the best solution, as done in the previous approach.

The approach gives the same solution X1=20 X2=40 Z=4480.

4. Exercises

1. If we had one more kg of sugar, how much more profit could we have made? This is called the marginal cost of sugar. We would be willing to pay up to this much for a kg of sugar, since we would still be making a profit, but not more than that. Repeat the exercise for butter.
2. How much must the price of a batch of Cookie 1 be changed for the the solution (20 batches of Cookie 1 and 40 batches of Cookie 2) not to be the optimum? Give a range for the price of a batch of Cookie 1 and another range for Cookie 2. Such studies are called sensitivity analysis. It reveals how sensitive the solution is to the model parameters.
3. The code given for the second approach works with two nested loops, one for each row of the coefficient matrix of the reduced set of equations. How would you generalize the code for the cases where the number of decision variables may vary?

XXII. CONSTRAINED NONLINEAR PROGRAMMING

Nonlinear programming deals with finding the values of the decision variables at which a nonlinear objective function attains its maximum or minimum. Sometimes there are constraints which limit the feasible region of the space spanned by the decision variables. We have seen such a case while discussing locating an airport among a set of cities so as to minimize the total length of roadways to be constructed. Here, we add a constraint to the location problem.

Consider four cities with coordinates (x_i, y_i) for i=1, 2, 3, 4 located within a square region. Let the city coordinates x_i, y_i be in the range [1, 100]. Let there be a river that diagonally passes through the region. The river is on the line y=100-x. Similar to the airport location problem, we want to build a port on the river while minimizing the total length of the roadways to each of the cities.

1. The Code

The code consists of a function subroutine and a loop. The function is the same as the one we used for the airport location problem. It simply computes the total distance from a given point to all four cities.

The main loop iterates through the X coordinate, from 1 to 100. At each iteration, we find the Y coordinate, and then call the function to compute the total road length needed if the port is to be laced at this location. The loop also keeps track of the best total road length so far. When the loop terminates, we have the coordinates of the best location.

```
clc
clear

// Four planar points, coordinates in the
// range of [1, 100]
city_x=[10, 80, 35, 36];
city_y=[42,  9, 91, 19];
// numerically find a point to place a new
// port on the river (y=100-x), such that
// the total distance from the cities to
// the port is minimized.

// f(a,b) gives the total distance from
// points (city_x, city_y) to (a,b)
function [d]=f(a, b)
 d=0;
 for i=1:4
  d=d+sqrt((a-city_x(i))^2+(b-city_y(i))^2);
 end
endfunction

bestZ=400;
bestX=0;
bestY=0;

for x=1:100
 y=100-x;
 Z(x)=f(x,y);
 if Z(x)<bestZ then
   bestX=x;
   bestY=y;
   bestZ=Z(x);
   end
end

plot(Z);
xtitle('X coordinate');
xlabel('km');
ylabel('total road length (km)');

printf("best lcation: (%d,%d)\n", bestX, 100-bestX);
printf("min length  :  %d\n", bestZ);
```

Figure 22.1. Seeking the Best Port Location.

The code prints out the best location for the port and the corresponding total road length.

```
best location: (48,52)
min length   :  169
```

Figure 22.2. Code Output.

The code plots t

he total road length as a function of the X coordinate. It is observed that the objective function value is somewhat sensitive to the location. Thus, searching for the best location seems to be worth the effort.

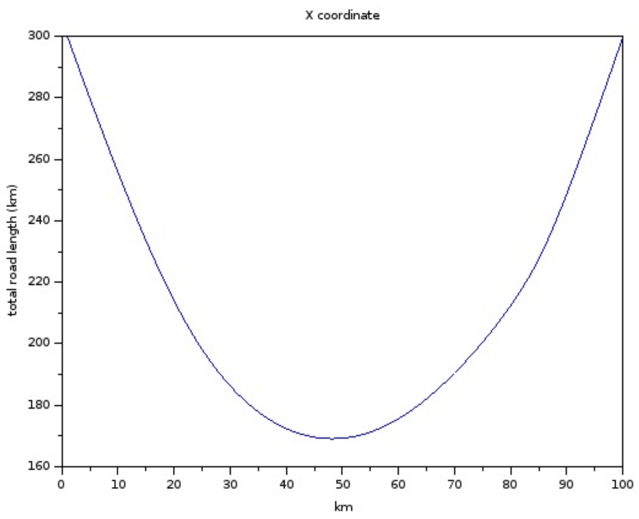


Figure 22.3. Total Road Length (objective function value) as a Function of the X Coordinate of the Port.

2. Exercises

1. Note that the city coordinates we used are the same as the ones we used for the airport location problem. The airport could be built anywhere in the region, while the river port must be on the river. That is, the airport optimization was unconstrained, while the river port location is constrained to be on the river. Compare the two solutions.
2. Add a few more lines to the code to plot the river and the cities. Also show where the location of the port which minimizes the total road length, as shown below.

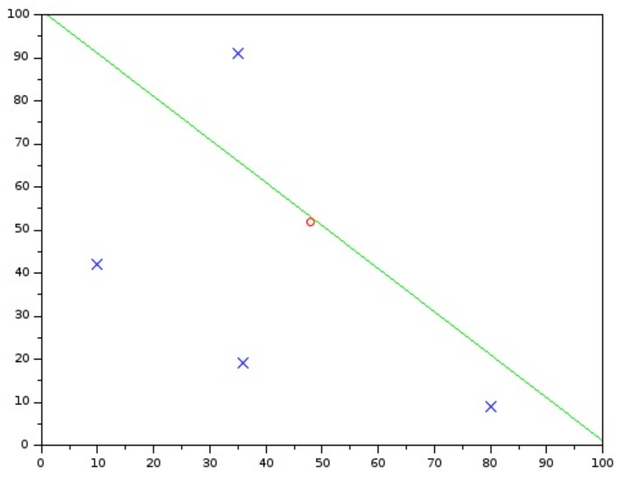


Figure 22.4. Cities (blue crosses), the River (green line), and the Port Location (red circle).

3. Suppose we have certain regions of the river which are deemed inconvenient for the port, say due to soil conditions or river width. These will introduce further constrains on the problem. How would you handle such additional constrains?
4. The constraint we used is a linear constrain. That is, the port location being limited to coordinates where y=100-x is a linear one. How would you handle nonlinear constrains? For example, consider the case $x^2+y^2<80^2$?

XXIII. MONTE CARLO ANALYSIS

Sometimes it becomes difficult to compute a given phenomenon, for the lack of a conceptual attack on the problem. It is often the case that we try to build as much insight as possible in search of a greater understanding. These initial trials are therefore quite important. A good engineer should possess the skills to quickly generate computational cases and glean insights from them.

In certain fortunate cases, an initial solution may present itself through analysis involving randomness. A classical textbook example is in computing the numerical value of π. It is known that the area of a disk with a radius of r is πr^2. Consider a unit square encapsulating a quarter of a disc as shown below.

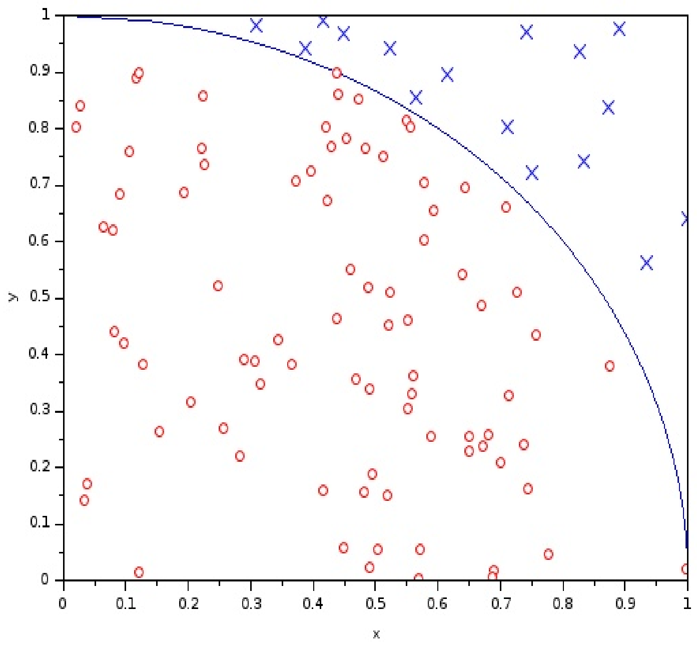


Figure 23.1. Computing the Value of π.

Here, we have a quarter disc inscribed within a unit square. If we generate random points in the unit square, it is rather straightforward to test if the random point is also within the disk. Let the point (x,y) be randomly generated. That is, we generate a random x in the interval [0, 1], and similarly a y in the same range. We make sure that x and y are statistically independent. Then, the point (x,y) will be inside the disc if

$$x^2+y^2\leq 1 \quad .$$

The fraction of the points within the disc approximates the fraction of the area of the unit square covered by the disk. Then, 4 times the fraction will approximate the area of the disc. Since the radius of the disc is 1, then, 4 time the fraction will also serve as an approximation to π.

1. The Code

The code given below is rather uncomplicated. In a sense, this is the beauty of the Monte Carlo method. It provides an uncluttered attack on the problem at hand.

```
clc
clear

RunLength=100000;
Count=0;

for i=1:RunLength
 x=rand();
 y=rand();
 if(x^2+y^2<=1) then Count=Count+1; end
end

disp(4*Count/RunLength);
```

Figure 23.2. The Code.

As expected, the output gives values close to π. One may increase the run length to seek better approximations.

2. Exercises

1. Change the run length and observe its effect on the quality of the approximation. Try run lengths of 100, 1,000, 10,000, 100,000, and 1,000,000. For each case, make 10 runs and compute the average. Plot these results to show the effect of the run length on the accuracy of the approximation.
2. Use the Monte Carlo method to compute the integral of Sin(x) for the interval [0, π]. That is, generate random points in the rectangle (0,0) to (π, 1). Then find the fraction of points under the curve Sin(x).

3. What is the expected distance between two random points in a unit square? What about in a unit equilateral triangle ot a unit circle?

XXIV. STATISTICAL FORECASTING

Statistical forecasting tries to estimate future values of a time series from information extracted from the past values. It differs from other forecasting methods, such as the so-called Delphi Method, as it relies on a statistical evaluation and modeling of the time series.

As an example, consider the rather fortunate company, the demand for whose product seems to display a constant upward trend. The monthly demand in tons is plotted over the past 30 months.

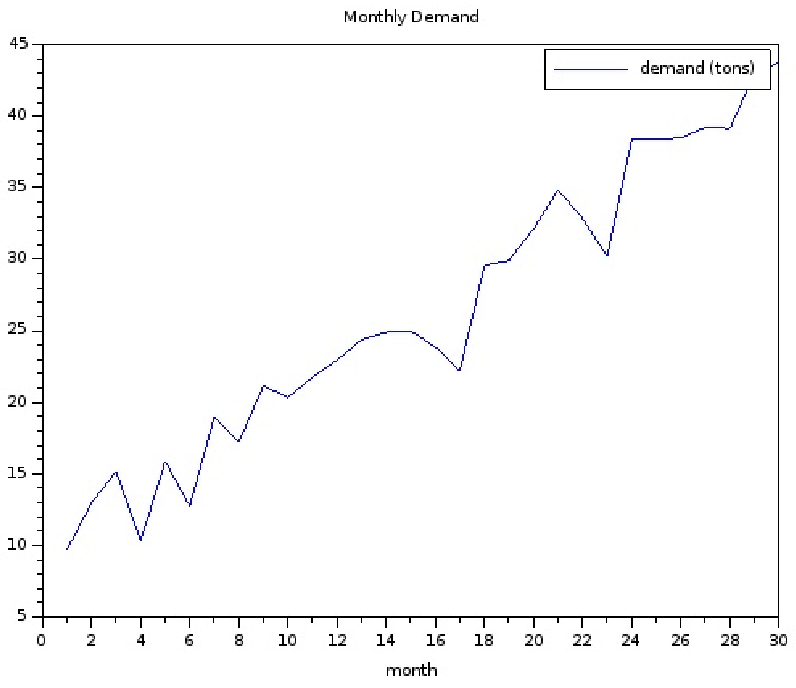


Figure 24.1. Monthly Demand in Tons.

The question on hand is “What will demand be for the next month?” We present two rather elementary techniques often used in industrial engineering.

1. Moving Averages

The Moving Averages (MA) method simply takes the average of a window of past values. If the window width is 10, for example, we average the most recent 10 values and use this average as the estimate for the next value in the time series. The wider the window, the less sensitive it is to noise. This is because, any noise superimposed on the time series will tend to cancel out in the long run. However, as the window widens, so does the estimate become less agile. That is, it takes a longer time to catch up with any sudden changes in the time series.

```
clc
clear
d=[ 9.68 12.99 15.14 10.32 15.86..
   12.75 18.98 17.26 21.15 20.32..
   21.76 22.94 24.38 24.88 24.99..
   23.86 22.22 29.56 29.90 32.11..
   34.84 32.85 30.18 38.36 38.37..
   38.48 39.20 39.13 42.95 43.75 ];

w=3;

function e=MovingAverage(s, n, k)
    count=min(n, k);
    e=0;
    for i=1:count
     e=e+s(n-i);
    end
    e=e/count;
endfunction

// moving average with window w
my=[];
mx=[];

for j=w+1:size(d,2)
    mx($+1)=j;
    my($+1)=MovingAverage(d, j, w);
end

plot(d')
xtitle('Monthly Demand');
xlabel('month');
ylabel('tons');

plot(mx, my, 'r')
legend(['demand (tons)'..
           'moving average (3) forecast']);
```

Figure 24.2. The Code for Moving Averages.

The code centers around the function MovingAverage(s, n, k), which computes the moving average forecast from the series s, using up to its first n terms. The last parameter k is the width of the moving average. The number of trailing elements to average is named “count” which is the minimum of k and n. The function simply accumulates the series elements in e, and then divides the sum in e by count.

The main loop computes the moving average estimates, from w+1 to the end of the data series, where w is the width parameter of the moving average. The estimates and the corresponding month values (indices) are stored in matrices my and mx, respectively. This way, as we plot (mx,my), the graph starts from the first value in mx, not from the origin. The code plots (mx,my) in red, also marking the data points with small circles (the letter 'o'). The resultant output plot is given below.

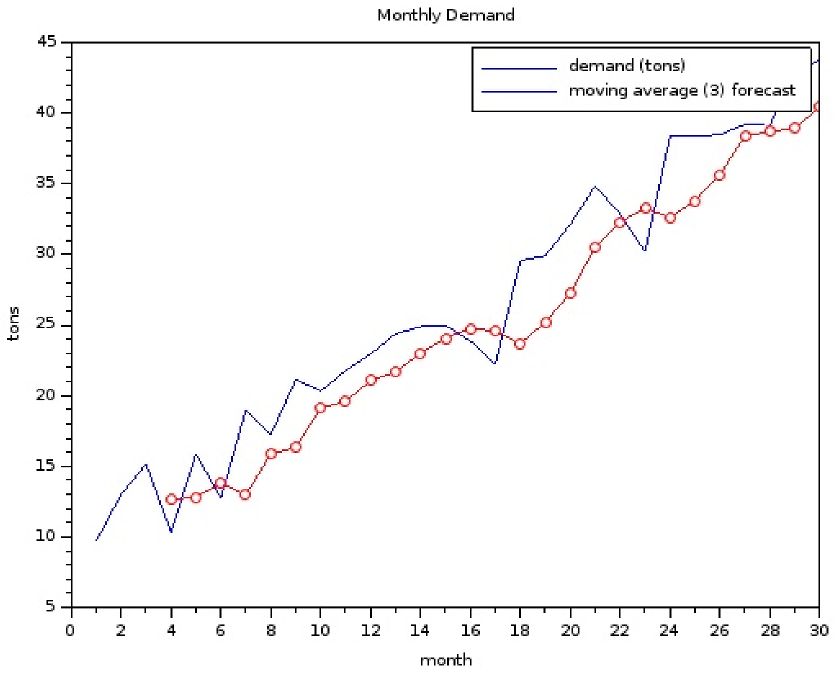


Figure 24.3. Moving Average Estimates (Window w=3).

Note that the estimate usually lags the data, since past data points are averaged. Since there is a general increasing trend, past points usually have a lower value than the next data point.

2. Exponential Smoothing

Another quite popular forecasting method is the Exponential Smoothing (ES) method. Here, the forecast for the next data point is estimated as a weighted sum of the last data point and the last estimate.

$$\hat{d}_{n+1} = \alpha d_n + (1-\alpha)\hat{d}_n$$

The circumflexes (the hats above the d's) indicate that they are estimates. The series $\{d_n\}$ is observed up to period n. The next estimate blends the last observation and the most recent estimate with parameter α, chosen between 0 and 1. The larger the α, the more emphasis on the most recent data point. Likewise, the smaller the α, the more emphasis on the most recent estimate. Thus, as α is increased, the estimate becomes more sensitive to the data. It also becomes more affected by any noise in the data. Contrarily, as α is decreased, the estimate becomes less sensitive to noise.

We next present the code that implements ES on the same data set.

```
clc
clear

d=[ 9.68 12.99 15.14 10.32 15.86..
   12.75 18.98 17.26 21.15 20.32..
   21.76 22.94 24.38 24.88 24.99..
   23.86 22.22 29.56 29.90 32.11..
   34.84 32.85 30.18 38.36 38.37..
   38.48 39.20 39.13 42.95 43.75 ];

alpha=0.5;

function e=ExponentialSmoothing(s, n, a)
 e=s(1);
    for i=2:n
     e=a*s(i)+(1-a)*e;
    end
endfunction

// exponential smoothing
ey=[];
ex=[];
for j=2:size(d,2)
    ex($+1)=j;
    ey($+1)=ExponentialSmoothing(d, j, alpha);
end

plot(d')
xtitle('Monthly Demand');
xlabel('month');
ylabel('tons');
legend(['demand (tons)']);
```

```
plot(ex, ey, 'ro-')
legend(['demand (tons)' 'exponential smoothing(0.5)
forecast']);
```

Figure 24.4. The Code for Exponential Smoothing.

The code is almost the same as the one given for MA. Once again, we use a function to compute the next estimate. Here ExponentialSmoothing(s, n, a) computes the next estimate given the series s, up to its n-th element. The estimate is thus for the (n+1)-st element of the series s. The last parameter a is the weight (alpha). The output plot shows the original time series and the ES estimates, plotted in red with small circles indicating the data points.

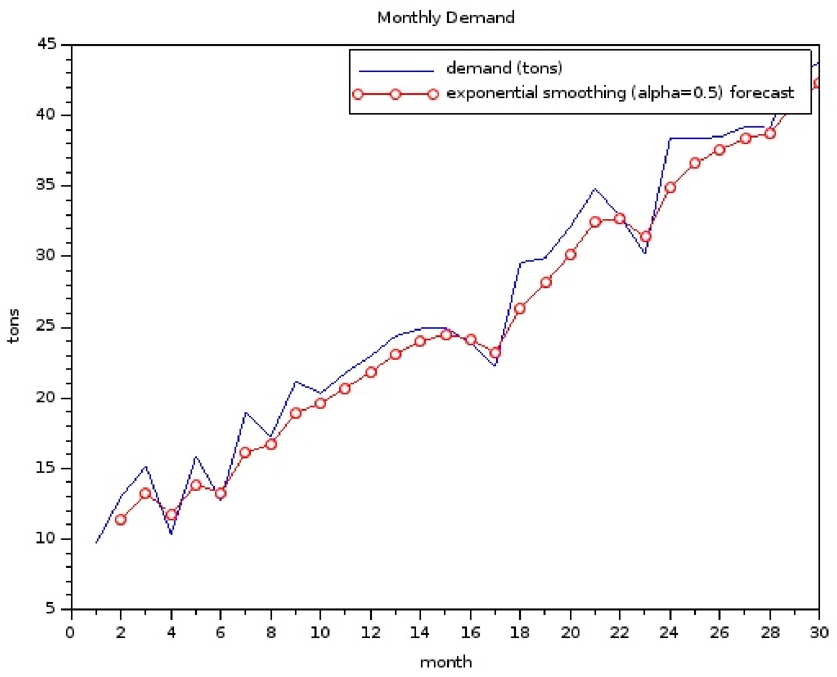


Figure 24.5. Exponential Smoothing Estimates (weight α=0.5).

3. Exercises

1. Find suitable data and run both forecasting methods. For example, you may use daily average temperatures or the travel time to go to school as your data set.
2. Change the MA and ES parameters in the given programs and observe their effects.

3. Both MA and ES are popular forecasting methods. What are the contrary advantages and disadvantages of the two forecasting methods? What are common advantages and the common disadvantages?
4. Try both methods on data that show seasonality. For example, beach attendance or ice cream sales are higher in the summer months. Is either method better in detecting seasonality?

XXV. WORKING WITH NOISY DATA

Measurement is an essential part of engineering. Accordingly, almost all engineering tasks require some form of data collection and processing. Measurements more often than not come with embedded noise. Data from human systems usually come with an added level of uncertainty. Suppose, for example, we observe successive arrivals to a supermarket checkout counter. The typical way to model the arrival stream is by treating the inter-arrival times as random variables. If we further assume that inter-arrival times are independently and identically distributed, then we need specify only one distribution function. Suppose we measure the inter-arrival times of 1000 successive customers and obtained the following histogram.

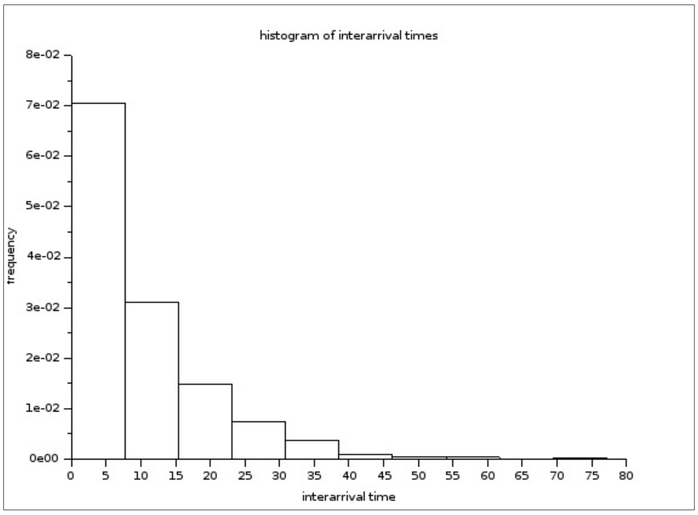


Figure 25.1. Histogram of Inter-arrival Times.

Moreover, let the average of the 1000 inter-arrival times be 10.03, and standard deviation 98.50.

1. Representing Data

As depicted above, drawing a histogram is a good initial step towards representing and understanding the properties of the random variable. A histogram may suggest the a suitable distribution to be used to model the random variable. Further, if we compute a few statistics, such as the average and the standard deviation, we may also start assigning distribution parameters.

For the data shown in the histogram in Figure 25.1, we compute a coefficient of variation=0.99. From the average and coefficient of variation, along with the general shape of the histogram, we may justifiably assume that the inter-arrival times are exponentially distributed with rate 1/10, that is, the reciprocal of the average.

2. Drawing Histograms

If need be, you may write a short function to draw the histogram. However, since histograms are frequently used, most engineering computation software has built-in functions to draw histograms. In this example, we will use the Scilab function histplot(). In its most elementary use,the function takes two parameters, the number of classes (n) and a vector of values (y), as histplot(n, y). The number of classes determines how many intervals are to be used. The documentation reveals that the default behavior of histplot() normalizes the frequencies (number of occurrences) of each interval. If you want to display the count, that is, the number of occurrences within that interval, then a third parameter is needed, as histplot(n, y, normalization=%f).

The following example illustrates the use of the function.

```
clc
clear
// RNG exponential distribution

fLambda=0.10;
nTrials=1000;
nClasses=10;

for n=1:nTrials
 y(n) = grand(1, 1, "exp", 1.0/fLambda);
end

 clf(1);
 scf(1);
 histplot(nClasses, y);
 xtitle('histogram of inter-arrival times');
 xlabel('inter-arrival time');
 ylabel('frequency');
```

Figure 25.2. Drawing Histograms.

The code uses the Scilab function grand() to generate exponentially distributed random variables and save them in the vector y. The function histplot() is called to draw the histogram.

The data and the histogram given in the previous section was actually generated by a similar program.

3. Exercises

1. Modify the code to generate uniformly distributed random variables and draw their histogram.
2. Modify the code to generate normally distributed random variables and draw their histogram.
3. Generate random variables with positive and negative skewness and observe differences in their histograms.

XXVI. THE LAW OF LARGE NUMBERS

In layman's terms, the law of large numbers simply states that the computed average of a sample drawn from a given distribution will converge to the distribution mean as the population size increases. This is an important theoretical point. It guarantees that the sample average is an unbiased estimate for the distribution mean.

We now attempt to check this assessment by computing sample averages and observing if they indeed converge to the distribution mean. Interestingly, here we use computation in a qualitatively different way. We use computation to verify a theoretical point. Many times, when understanding an engineering phenomena, before we make claims and try to justify them, we rely on numerical cases to develop our understanding, to gain insights, and to hone our instincts.

1. The Code

Consider the following code that uses the Scilab function grand() to generate exponentially distributed random variables. The distribution parameter fLambda is set to 0.50, which means that the expected value of the generated random variables will be 1/0.50=2.0.

The code generates nTrials number of random variables. The sum of the successive random variables are stored in the vector fSums. Similarly, the sample averages are stored in fAverages. The code plots fAverages, which, if the law of large numbers holds, should converge to the mean (2.00).

```
clc
clear
// law of large numbers

fLambda=0.50;
nTrials=10000;
fSums(1) = grand(1, 1, "exp", fLambda);
fAverages(1)=fSums(1);

for n=2:nTrials
 fExpRN = grand(1, 1, "exp", 1.0/fLambda);
 fSums(n) = fSums(n-1)+fExpRN;
 fAverages(n)=fSums(n)/n;
end

 scf(1);
```

```
plot(fAverages);
xtitle('Averages (Exp RNG using grand())');
xlabel('trials');
ylabel('sample average');
```

Figure 26.1. The Code.

The code output is a graph showing how sample averages change as a new random variable is added to the sample.

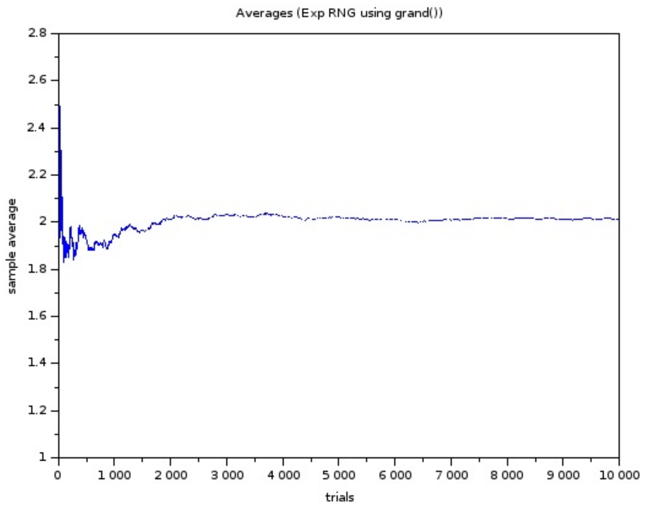


Figure 26.2. The Output.

As expected, the sample average converges toward the mean (2.00). Interestingly, the convergence rate is somewhat slow. Although the sample average is practically within 10 percent, it does not quite settle until the sample size if a few thousand. Such observations provide important engineering insights.

2. Exercises

1. Modify the code to generate normally distributed random variables with mean 10. Try coefficients of variation of 0.5, 1.0, and 2.0. Observe the convergence rate in each case. How does the coefficient of variation affect the rate at which the sample average converges to the mean?
2. Repeat the experiments with uniformly distributed random variables.

XXVII. THE CENTRAL LIMIT THEOREM

The central limit theorem is perhaps the cornerstone of statistics. It goes a long way towards explaining why the normal distribution plays such a central role in statistics. In a nutshell, the central limit theorem asserts that the distributions of sample averages converge towards the normal distribution. That is, consider a sample of random variables drawn from any distribution. Suppose we compute the sample average. Then, suppose we repeat this process of drawing a sample and computing its average. The series of averages, themselves, may be regarded as random variables. Moreover, if the sample sizes are the same, these averages will have the same distribution. The central limit theory says that the distribution of these averages will approach the normal distribution.

1. The Code

Once again, we use computation to test the validity of the central limit theorem. This test, of course, is not meant to be proof. Rather, the test is to provide numerical insights and hone our engineering instincts.

```
clc
clear

// central limit theorem

fLambda=0.50;
nSampleSize=100;
nTrials=1000;
nClasses=30;

for n=1:nTrials
    fSum=0;
    for k=1:nSampleSize
      fExpRN = grand(1, 1, "exp", 1.0/fLambda);
      fSum = fSum+fExpRN;
    end

 fAverages(n)=fSum/nSampleSize;
end

 scf(1);
 histplot(nClasses, fAverages);
```

```
xtitle('central limit theorem');
xlabel('averages');
ylabel('frequency');
```

Figure 27.1. The Code.

As seen, the code generates samples of size 'nSampleSize' from an exponentially distributed random variable with rate 'fLambda'. The sample averages are computed and store in the vector 'fAverages'. The experiment is repeated for 'nTrials' number of samples. These parameters are defined in the beginning, so that the code may be easily modified. The code then draws a histogram of the averages. We observe if the histogram resembles one that came from a normally distributed random variable.

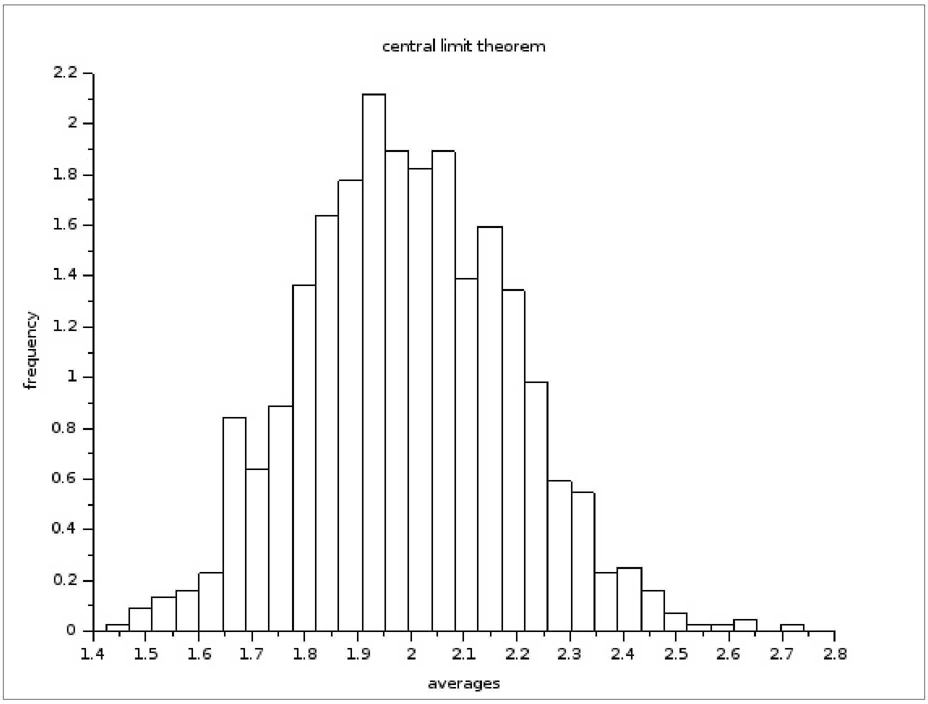


Figure 27.2. The Histogram of Averages of Exponentially Distributed Random Variables.

The histogram uses 30 classes (intervals). The exponential distribution has a mean of 2.0, since its rate os 0.50. Note that the averages are clustered around the mean (2.0). The histogram looks like a normal distribution.

2. Exercises

1. Predict the effects of changing the code parameters. Namely, what is the expected effect of changing the distribution rate, the sample size, or the number of trials?
2. Modify the code by altering the code parameters, as investigated in Exercise 1. How well did the output match your predictions of Exercise 1?
3. Modify the code to generate samples of uniformly distributed random variables. Repeat Exercise 1 and 2 with uniformly distributed random variables.
4. (Difficult)

   Part 1.

   In this chapter, we generated samples of exponentially distributed random variables, say X. Let the rate of the exponential distribution be 'r', and the sample size be 'N'. Let the sample averages be denoted as the random variable A. What are the mean and variance of the distribution of the averages, A? Note that the mean of A is a function of the rate r, irrespective of the sample size N. However, the variance of A is also a function of N.

   Part 2.

   Generate normally distributed random variables with the mean and variance equal to those of A as obtained in Part 1. Plot the histogram of these normally distributed random variables. How much does this histogram resemble the one in Figure 27.2? What are your observations and conclusions?

XXVIII. CURVE FITTING (SIMPLE AND MULTIPLE REGRESSION)

We all have seen a line fit through noisy data, such as depicted below. Here, the class attendance of students and their final class grade are plotted.

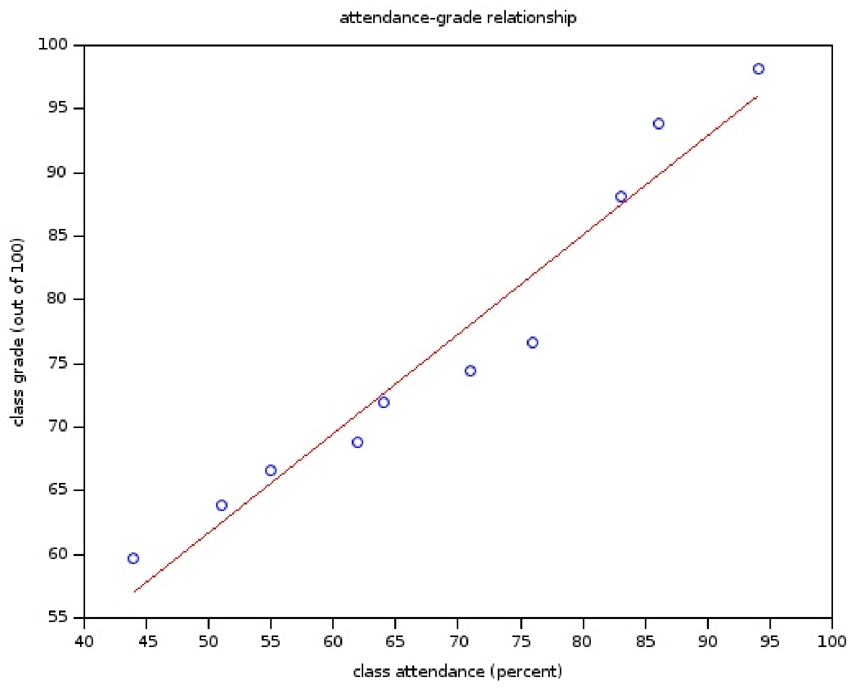


Figure 28.1. Relationship Between Class Attendance and Grade.

The plot offers evidence that the more you come to class, the better your grade will be. There is an implied model of a cause-effect relationship. The grade is the outcome, or the effect. The attendance rate is the cause that determines the effect. Regarding this cause-effect model, the attendance is the independent variable, that is, the variable one may vary. The grade is the dependent variable, that is, the effect of our attendance rate. In other words, once the attendance is set, the outcome (grade) follows.

The line that is fit through the data points is thus considered a model of this cause-effect relationship. The model is a quantitative one – and engineers like quantitative, or numerical, models. The model suggests that there is a linear relationship between the cause and the effect. In fact, one may write this as a algebraic function. Let attendance be denoted by X and the grade by Y. The model may then be expressed as

$$Y=b_0 + b_1*X$$

where b_0 is the intercept of the line, and b_1 is the slope. We call this a linear regression model, since the independent variables are combined in a linear fashion (multiplied by constants and added). Moreover, the model above is a so-called “simple” linear regression, since it has only one independent variable. It is also possible to have multiple independent variables, with a corresponding model, such as

$$Y=b_0*X_0 + b_1*X_1 + b_2*X_2 + b_3*X_3 + b_4*X_4 \ldots + b_k*X_k.$$

The above model is still linear in the independent variables (the X_i) but there are more than one independent variable. Such models are called multiple linear regression models.

You may have noticed that the above model has no explicit constant term. The constant term may simply be implemented by selecting all X_0 terms as 1.

1. The Least-Squares Method

Once the model parameters, that is, the coefficients bi, are determined, the linear regression model gives a concise formula that numerically relates the size of the effect to the size of the causes. The “least squares method” provides an efficient technique to determine the model parameters. Moreover, the method may be summarize in matrix form rather elegantly. We place the dependent variables in a column vector Y, as given below.

$$Y=\begin{bmatrix} 59.66 \\ 63.90 \\ 66.58 \\ 68.81 \\ 71.97 \\ 74.48 \\ 76.65 \\ 88.08 \\ 93.83 \\ 98.18 \end{bmatrix}$$

The corresponding independent variables are placed in a matrix as,

$$X = \begin{bmatrix} 1 & 44 \\ 1 & 51 \\ 1 & 55 \\ 1 & 62 \\ 1 & 64 \\ 1 & 71 \\ 1 & 76 \\ 1 & 83 \\ 1 & 86 \\ 1 & 94 \end{bmatrix}$$

Note that the first column of the matrix X is all 1s. This follows the modeling convention

$$Y=b_0*X_0 + b_1*X_1$$

so that b_0 is the constant term of the model. The least squares method gives the parameters b_i as the vector

B=((X'*X)^-1)*X'*Y;

$$B = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{bmatrix} = \left(X^T X\right)^{-1} X^T Y$$

The method requires the inversion of a k-by-k matrix, where k is the number of model parameters. In case of simple linear regression, this is a two-by-two matrix.

2. The Code

The following code computes the model parameters for the attendance-grade relationship discussed in the beginning of the chapter. The data points and the fit is then plotted.

```
clc
clear

X=[
1    44;
1    51;
1    55;
1    62;
1    64;
1    71;
1    76;
1    83;
1    86;
1    94;
];

Y=[
59.66
63.90
66.58
68.81
71.97
74.48
76.65
88.08
93.83
98.18
];

B=((X'*X)^-1)*X'*Y;

disp(B)

FIT=[];
for i=1:size(Y,1)
 FIT(i)=B(1)+X(i,2)*B(2);
end

scf(0) // create figure 0
clf(0) // clear figure 0
plot(X(:,2),Y,'bo')
plot(X(:,2),FIT,'r')
xtitle('attendance-grade relationship');
xlabel('class attendance (percent)');
ylabel('class grade (out of 100)');
```

Figure 28.2. The Code.

The code outputs the model parameters (i.e., the coefficients) as,

```
22.675913
0.7804386
```

Thus, we may now declare the quantitative cause-effect relationship of attendance and grades as

$$grade = 0.78*attendance + 23$$

You may noticed that we truncated the coefficients to two significant digits. Given the sample size and various external factors, we are cognizant that our model is an approximation at best. Thus, two significant digits are probably sufficient. It would be meaningless to give these coefficients in 7 or 8 significant digits. We may even simplify our findings as

$$grade = 0.8*attendance + 20$$

which would most probably be sufficient as well as easy to remember.

Also note that the model is a descriptive one. It allows one to predict the outcome given the inputs based on a statistical study of similar cases. It does not provide an explanation for the cause-effect relationship. In this sense, the regression model is by no means an end product, but a component of a general view of a cause-effect relationship. After all, you may collect irrelevant data and still fit a curve through the data points.

As a final note, let us be reminded that not all fits are equally accurate. How well the fit describes the data is the topic of regression analysis. Here, by eyeballing the graph, we may be convinced that the fit is good enough to describe the data, since almost all points, although not exactly on the line, are nonetheless quite close to the fitted line.

3. Exercises

1. Collect data from your classmates on the distance and travel time from home to school. Fit a line through the data that gives the travel time based on the distance. Plot the data points and the fit. Discuss how well the model explains the data.
2. Consider throwing a ball into the air and measuring how high it went, along with how long it took to land back on the ground. Let H and T be the height and the time. You will notice that a relationship between H and T is best given by a quadratic model, since the ball decelerates on its way up, and then accelerates falling back down. Consider the modeling

   $$H=b_0*T_0 + b_1*T_1 + b_2*T_2$$

   where T_0 terms are 1s, T_1 terms are the elapsed time, and the T_2 terms are the squared times (squares of the T_1 terms). Collect data (or use the data given below) and estimate the model parameters. How are the model parameters related to the gravitational acceleration?

Trial	Time	Height
1	1.05	2.78
2	1.58	6.23
3	2.09	10.92
4	2.57	16.55
5	3.14	24.67

XXIX. SIMULATING A SIMPLE QUEUE

The simplest queueing system is typified by customers randomly arriving at a server. If the times between successive arrivals and the service times are exponentially distributed, this system is referred to as an M/M/1 queue. Here the letter 'M' refers to the memoryless property of the exponential distribution. The two Ms indicate that the arrival and the service processes are exponential, while the trailing '1' indicates that the system contains only one server.

Imagine a single check out line at a small store with a cashier. Customers come and the cashier checks out the customers. The times between customers joining the line is assumed to be exponentially distributed. So is the service times, that is, the time it takes the cashier to check out each customer, exponentially distributed. Of course, these two distributions may have different rates.

The memoryless property of the exponential distribution is quite interesting. When the inter-arrival times are exponentially distributed, this simply means that the expected remaining time to an arrival is independent of the time we have already waited for an arrival. Thus, the process has no memory of how long we have waited so far for the next arrival. The discrete counterpart is the geometric distribution. Say, you toss a die until you get a '1'. The expected number of tosses until the '1' arrives is 6, since the probability of tossing the '1' is 1/6. Now, suppose that you have already tossed the die 3 times and that the '1' has not “arrived”. The expected number of times you need to toss the die until the '1' occurs is still 6, independent of the 3 tosses so far. Thus, the process has no memory of the 3 tosses already completed. The process is memoryless.

There is a close relationship between the exponential distribution and the geometric distribution. The latter may be considered as the limiting continuous case of the former. Specifically, the probability of an arrival in the next Δt time units is $\lambda \Delta t$ where λ is the rate of the exponential distribution. This property may be viewed as the discretization of the continuous distribution. We will use this property to simulate the M/M/1 queue.

The preceding sentence used the word “simulation”. A simulation is like a video game. You model a dynamical system by a set of states and actions, and then let the computer numerically go through the steps that correspond to the evolution of the system over time. Meanwhile, the code keeps track of performance measures. Here we will record how much the server is idle and the number of customers in the system as indicators of the system performance.

1. A Specific Queue

Consider an M/M/1 queue with an arrival rate of rArrive=10 customers per hour and a service rate of rService=20 customers per hour. Let us select our Δt to be hDeltaTime=0.001 hour. Then, the probability of an arrival in the next Δt is

pArrival=hDeltaTime*rArrive.

Similarly we define,

pService=hDeltaTime*rService.

Defining the parameters explicitly in the beginning of the code is good software practice. This way, you can change the system parameters easily and repeat the runs. If the code is well organized, you can change the parameter settings at one well-defined line in the code.

The run time and the number of periods (of duration Δt) are likewise defined. The code runs the clock from 0 to the specified run time in Δt time steps, called periods. This approach amounts to a continuous-time simulation of the system. As the simulation progresses, the code keeps track of the number of customers in the system and the number of periods the server is idle.

```
clc
clear

rArrive =10;     // rate: arrival per hour
rService=20;     // rate: service completion
hDeltaTime=0.001; // hour
pArrive=hDeltaTime*rArrive;   // prob arrival
pService=hDeltaTime*rService; // prob service

nRunTime=100;    // hours
nNumPeriods=nRunTime/hDeltaTime;

nCustomers=0;    // number of customers in the system
nEmptyPeriods=0;
nCustomerPeriods=0;

for time=1:nNumPeriods
  if((nCustomers>0) & (rand()<pService))
    then
```

```
     nCustomers=nCustomers-1;
    end

  if(rand()<pArrive)
    then
     nCustomers=nCustomers+1;
    end

  if(nCustomers<1)
   then
    nEmptyPeriods=nEmptyPeriods+1;
   else
nCustomerPeriods=nCustomerPeriods+nCustomers;
   end
   end

 fAveCustomers=nCustomerPeriods/nNumPeriods;
 fUtilization=1.0-..
    (nEmptyPeriods/nNumPeriods);

printf( simulated performance measures\n");
printf("Utilization: %8.2f\n", fUtilization);
printf("Average number of customers:..
                    %8.2f\n", fAveCustomers);
```

Figure 29.1. The Code.

Upon completion of all simulation iterations, the code computes and prints the utilization, that is, the percent of time the server was busy, as well as the average number of customers in the system.

The code output is shown below.

```
simulated performance measures
Utilization:                     0.50
Average number of customers:     1.00
```

Figure 29.2. The Output.

As seen, the server of our M/M/1 queue is busy 50% of the time. This mean that it is also idle half the time. This makes sense. Suppose we have the expected 10 arrivals per hour. The server serves 20 customers per hour. This means it takes the server an expected 1/20 hours to serve each customer. Given the 10 arrivals, it would take a total of 10/20, or half an hour to serve all arrivals in an hour. This makes the utilization ½ or 50%. Of course, the number of arrivals in an hour will vary from over time, but the argument is intuitive.

2. Exercises

1. The exponentially distributed inter-arrival times and service times are obtained by considering a small Δt of 0.001 hours. We referred to this as a discretization. Just as an image is made up of discrete pixels, the smaller the Δt the finer the image. In our case, we want Δt to be small enough to give good results. That is faithfully represent the continuous distributions. However, a small Δt also means more iterations and thus, longer processing times. Experiment with different Δt values to see when it becomes so large that it fails to provide an acceptable representation of the continuous distribution.
2. Change the arrival and service rates and run the simulation again. Observe what happens when the arrival rate is greater than the service rate. Discuss your findings.
3. The ratio of the arrival rate to the service rate is called the traffic intensity. It is known that the utilization of an M/M/1 queue is the same as the traffic intensity. Modify the code to compute the traffic intensity and see how it varies from the simulated utilization factor.
4. Theoretical work shows that the number of customers in an M/M/1 queue is $\frac{\rho}{1-\rho}$ where ρ is the traffic intensity. Modify the code to print out the theoretical value of the expected number of customers in the system. Compare the theoretical values to the simulation results.
5. We discretize a continuous time process by considering small time steps of duration Δt . We understand that the smaller the time step, the more accurate our simulation will be. However, the smaller the time step, the more the computations. In fact during many time steps no system event occurs. That is, many time steps are void of either an arrival or a departure. Modify the program to display how many time steps witness a system event. Let the number of time steps during the run in which there is a system event be K. Let there be a total of N time steps during the simulation run.
    1. Consider plotting the faction K/N as a function of the time step size h. Guess the shape of this function.

2. Plot the fraction K/N as a function of the time step size Δt.
   Did the graph turn out to be as you conjectured?

XXX. DISCRETE-EVENT SIMULATION

The preceding chapter developed a continuous-time simulation for the M/M/1 queue. Time was discretized into small time steps (Δt). We understand that the smaller the Δt , the more accurate our simulation will be. However, the smaller the Δt , the more computations are needed. Realizing that only a few of the time steps actually witness a system event, we can improve the efficiency of the simulation by simply considering the system at times when an event occurs, and disregarding the rest of the time steps. This approach is called “discrete-event simulation”.

It should be noted that simulation and related software are rather well developed subjects in industrial engineering. There are many good simulation tools available to the industrial engineer. Simulating a system using a general-purpose engineering computation tool such as Octave or Scilab is not the best idea. We undertake such a task here to illustrate the fundamental concepts in simulation, rather than offer a practical approach.

The queueing system discussed in this chapter is a rather simple construct, whose performance measures are readily obtained in closed form. Almost all realistic industrial engineering systems are so complicated that closed-form expressions are not available. One may resort to approximations or simplifications to obtain analytical expressions. Alternatively, one may also simulate the system. Simulation allows one to keep system idiosyncrasies that hinder analytical solutions.

1. The Code

The following code illustrates a discrete-time simulation of the M/M/1 queue. We define and use a function named “GetNextEvent” which takes the number of customers in the system as an argument. It returns two quantities: the next event type and the time until the next event. You may think of the return value of the function as a vector of size 2. The code defines the event types as “eventArrival” and “eventService”. These are the only two things that can happen. Either a new arrival or the completion of service. We define these event types as global variables and assign them values 1 and 2, respectively. Such definitions are referred to as “enumerate types” in software languages. The C language, for example, has built-in features to facilitate enumerated types. Enumerated types help code readability. Who wants to refer to the card suits as {1,2,3,4} when it is possible to define {spades,clubs,hearts,diamonds}?

Since we will use the enumerated types, as well as the arrival and service rates in the main body of the code as well in the function, we defined these as “global variables”.

```
global rArrive;
global rService;
rArrive =10; // rate: arrival per hour
rService=20; // rate: service completion per hour
global eventArrival;
global eventService;
eventArrival=1;
eventService=2;
```

Figure 30.1. Global Variables and Enumerated Types.

The function GetNextEvent() is given below. It generates two exponentially distributed random variables. One is the time to next arrival, the other, the time to service completion. The function takes as an argument, the number of customers in the system. If the number of customers in the system is zero, then only an arrival is possible. In this case, the function returns eventArrival as the event type, and the time to the next arrival. If there is one or more customer in the system, then a comparison is in order. The next event depends on which random variable is smaller. If it is the time to the next arrival, then the function returns eventArrival and the time to the next arrival. Otherwise it returns eventService and the time to service completion.

```
function [nextEvent, elapsedTime]=..
GetNextEvent(nCustomers)
 tArrival=grand(1, 1, "exp", 1.0/rArrive);
 tService=grand(1, 1, "exp", 1.0/rService);
 if ((nCustomers>0) & (tService<tArrival)) then
  nextEvent=eventService;
  elapsedTime=tService;
 else
  nextEvent=eventArrival;
  elapsedTime=tArrival;
 end
endfunction
```

Figure 30.2. The Function GetNextEvent().

Once again, we use the built-in Scilab function grand() to generate the exponentially distributed random variables.

The main portion of the code is rather straightforward. We initialize a few variables and run the simulation until we reach the end of the run time.

```
tRunTime=100; // hours
nCustomers=0; // number of customers in the system
tEmpty=0;
tCustomerTime=0;

time=0;

while (time<tRunTime),
  [nextEvent, elapsedTime]=GetNextEvent(nCustomers);
  if(nCustomers==0) then tEmpty=tEmpty+elapsedTime;
end;
  tCustomerTime=tCustomerTime+nCustomers*elapsedTime;
  time=time+elapsedTime;
  if(nextEvent==eventArrival) then
     nCustomers=nCustomers+1;
    end;
  if(nextEvent==eventService) then
     nCustomers=nCustomers-1;
   end;
end;
```

Figure 30.3. The Main Loop.

The main loop keeps track of time and runs until time exceed the specified tRunTime. The function GetNextEvent() is called. The function returns the elapsed time until the next event as well as the type of the next event. If the next event is an arrival, then the number of customers in the system is incremented. If it is a service completion, then the number of customers is decremented. The loop also keeps track of two performance measures. The first is the total empty time, kept as the variable tEmpty. The elapsed time is added to tEmpty if there were no customers in the system. The other performance measure, associated with the variable tCustomerTime keeps track of the number of customers in the system multiplied by the elapsed time. This will be used to compute the average number of customers in the system.

```
  fAveCustomers=tCustomerTime/tRunTime;
 fUtilization=tEmpty/tRunTime;

 printf("--- simulated performance measures ---\n");
```

```
printf("Utilization:..
                           %8.2f\n", fUtilization);
printf("Average number of customers:..
                          %8.2f\n", fAveCustomers);

traffic_intensity=rArrive/rService;
ave_num_in_system=..
        traffic_intensity/(1.0-traffic_intensity);

printf("--- computed performance measures ---\n");
printf("traffic_intensity:%8.2f\n",traffic_intensity);
printf("ave_num_in_system:%8.2f\n",ave_num_in_system);
```

Figure 30.4. Displaying the Performance Measures.

The final few lines of the code is given in Figure 30.4. The performance measures are computed and displayed. Values obtained from the simulation as well as theoretical values are displayed.

2. Exercises

1. How quickly does the performance measures converge to the theoretical values? Experiment with different rates and run times. How would you quantify the convergence rate describing how quickly the simulation results approach the theoretical limits?
2. Plot the convergence rate defined in the previous exercise as a function of the traffic intensity.
3. Compare the efficiency of the two simulation programs given in this and the previous chapter. What are the advantages and the disadvantages of continuous-time simulation versus discrete-event simulation?

XXXI. SIMULATING THE G/G/1 QUEUE

The G/G/1 queue is like the M/M/1 queue, except that the inter-arrival and service time distributions are allowed to be general gistribution. The letter 'G' denotes a general distribution.

It must be noted that the code given in the previous chapter works only because the exponential distribution is a memoryless distribution. That is, at and given time, the remaining time to the next arrival is still exponentially distributed with the same rate, irrespective of how long we have waited for the arrival. The same goes for service completion. Thus, our function GetNextEvent() generates two random variables, and uses only one of them, disposing of the other. The service completion time is obtained the same way, no mater how long the current customer has been in the system. If the service time distribution is not memoryless, then we must obtain a service completion time and associate it with the customer. A good way to do this is by placing all the arrival times and service times in a vector. Similarly, one could then compute another vector of service completion times. Finally, a run through the arrival times and service completion times will allow us to collect the performance measure statistics. The following code implements such a strategy.

1. The Code

As usual, the code starts with generating a vector of inter-arrival times and service times. Again, we make use of the built-in Scilab function grand() to generate the inter-arrival and service times. In the code given, we use the uniform distribution. However, the code is general, and would work with any distribution.

```
nTotalArrivals=100;  // determines run time
// arrival times and service (processing) times
arrivalTimes=grand(nTotalArrivals, 1, "unf", 0.0, 4.0);
serviceTimes=grand(nTotalArrivals, 1, "unf", 0.5, 1.5);

for n=2:nTotalArrivals
 arrivalTimes(n)=arrivalTimes(n)+arrivalTimes(n-1);
end
```

Figure 31.1. Generating the Inter-arrival and the Service Times.

Rather than the time between arrivals, we cumulatively add the inter-arrival times so that the k-th element of the vector arrivalTimes holds the time at which the k-th customer arrives. That is, the elements of the vector become timestamps, rather than inter-arrival times.

Similarly, the service completion times are computed and placed in the vector departureTimes. Again, the k-th element of this vector shows the time at which the k-th customer leaves the system.

```
// generate service completion times
waitTime=0;
lastCompletionTime=0;
departureTimes=[];
for n=1:nTotalArrivals
 waitTime=max(0, lastCompletionTime-arrivalTimes(n));
 departureTimes(n)=arrivalTimes(n)
+waitTime+serviceTimes(n);
 lastCompletionTime=departureTimes(n);
end
tRunTime=departureTimes(nTotalArrivals);
```

Figure 31.2. Generating the Departure Times.

In computing the departure times, we must add to the arrival time not only the service time, but also the time that the customer spends waiting in the queue. We compute the latter as the variable waitTime. We also keep track of the time the last customer is served, as variable lastCompletionTime. The simulation run time (tRunTime) is simply the time the last customer leaves the system. Alternatively, you could cut the simulation when the last customer arrives. As long as the number of customers is high enough, these two approaches would give similar results.

The rest of the code simply goes through the arrival times and the departure times to collect the statistics. Again, we keep track of the time the system is empty and the product of the number of customers and the elapse time. The loop makes use of two indices. The variable indexArrival denotes the index of the most recent customer who arrived. Similarly, the variable indexDepart holds the index of the customer who most recently left the system.

```
// run simulation collect data
nCustomers=0;    // number of customers in the system
tEmpty=0;
tCustomerTime=0;
indexArrive=1;
indexDepart=1;
time=0;
```

```
while (indexArrive<nTotalArrivals)

if
(arrivalTimes(indexArrive)<departureTimes(indexDepart))
then
     // next event is an arrival
 elapsedTime=arrivalTimes(indexArrive)-time;
  if(nCustomers==0) then tEmpty=tEmpty+elapsedTime;
  else
   tCustomerTime=tCustomerTime+elapsedTime*nCustomers;
   end
     nCustomers=nCustomers+1;
     time=arrivalTimes(indexArrive);
     indexArrive=indexArrive+1;
   else
     // next event is a departure
     elapsedTime=departureTimes(indexDepart)-time;
     tCustomerTime=tCustomerTime+..
                           elapsedTime*nCustomers;
     nCustomers=nCustomers-1;
     time=departureTimes(indexDepart);
     indexDepart=indexDepart+1;
  end
end
```

Figure 31.3. The Main Loop.

The remainder of the code is similar to the one given in the previous chapter. It computes and displays the performance measures.

```
 fUtilization=1.0-(tEmpty/tRunTime);
 fAveCustomers=tCustomerTime/tRunTime;

 printf("--- simulated performance measures ---\n");
 printf("Utilization:                    %8.2f\n",..
                                         fUtilization);
 printf("Average number of customers: %8.2f\n",..
                                        fAveCustomers);
```

Figure 31.4. Displaying the Performance Measures.

Although the code is a bit longer than the one given in the previous chapter, we note that it is written not for efficiency or code size but for readability. It is written to be logically organized. You may feel that the three separate loops could indeed be combined to improve efficiency. This, however, would make the code less tractable. Once again, our objective is to illustrate the fundamental principles. You are encouraged to experiment with the code and try to merge some of the steps to improve efficiency.

2. Exercises

1. The code given above uses the uniform distribution to generate inter-arrival and service times. Experiment with the distribution parameters and observe their effect on the utilization factor and the average number of customers in the system.
2. It was mentioned that the utilization factor of the M/M/1 queue is the traffic intensity. Check if the same true for the uniform distributions. Consider the traffic intensity to be the ratio of the mean service time to the mean inter-arrival time.
3. Modify the code and try several other distributions such as the Erlang or beta distributions. Please note that the inter-arrival times and service times must be non-negative. Compare the results to the M/M/1 queue with the same mean inter-arrival time and the mean service time. Which distribution leads to a higher utilization rate? Why?

Alphabetical Index

REFERENCE TEXTBOOKS

Avraham Shtub, A. and Y. Cohen , Introduction to Industrial Engineering, 2nd Edition, CRC Press, Boca Raton, FL, 2015, ISBN 9781498706018.

Hildebrand, F. B., Introduction to Numerical Analysis, 2nd Edition , Dover Books on Mathematics, Mineola, NY, ISBN-13: 080-0759653638 / ISBN-10: 0486653633 (reprint of the McGraw-Hill Book Co., NY, 1956 edition).

Hillier F.S., and G.J. Lieberman, Introduction to Operations Research, 7th Edition, McGraw Hill, Boston, MA, 2001 , ISBN 10: 0072321695 / ISBN 13: 9780072321692.

SOFTWARE SOURCES

Example code used in this book, books.yeralan.org, 2016.

Scilab, www.scilab.org, 2016.

Octave, www.gnu.org/software/octave/, 2016.