CHAPTER 1: *ERRORS*

Introduction

Computational physics relies greatly on numerical methods, which are the subject of study in the field of mathematics known as *numerical analysis*. Instead of solving mathematical problems analytically (or by pen and paper), numerical analysis strives to solve problems iteratively (or through repetition). The solution can then be coded and given for computers to solve.

As a result, however, particular errors arise during the solution of the problem. Since the goal of numerical analysis is simply to approximate the solution and not to solve it exactly, discrepancies can arise between the value the computer arrives at and the analytical solution to the problem. Thus, a budding physicist must:

(1) know where these errors come from,

(2) know how to quantify them and how the error propagates in a calculation, and

(3) minimize these errors to keep the approximate as close to the original as possible.

This chapter intends to discuss errors in computational physics in terms of (1), (2) and (3). We begin with the sources of these errors.

Sources of error

There are five main sources of error in numerical analysis, which are as follows:

(1) Error in the mathematical modeling of the problem

In physics, a *mathematical model* refers to a physicist's attempt to explain physical phenomena by relating two or more physical variables of interest using mathematical relationships. Physical phenomena being as complicated as they are, physicists often resort to simplifying assumptions and in doing so, they create more tractable and less accurate models, resulting in error

Example. The force acting on a projectile of mass m is usually described using the differential equation

$$m\frac{d^2y}{dt^2} = -mgy \tag{1.1}$$

which can easily be solved analytically. Normally, however, as the projectile falls to the Earth, the air exerts a drag force on it, which can be assumed to be proportional to velocity. Thus, improving on model (1),

$$m\frac{d^2y}{dt^2} = -mgy + b\frac{dy}{dt} \tag{1.2}$$

where b is greater than zero and is the drag coefficient. Thus, there is some error in using model (1.1), as it neglects to account for the frictional force between the air and the projectile.

(2) Blunders

These simply refer to programming errors which the physicist can make while attempting to code the solution numerically.

(3) Uncertainty in the physical data

There is uncertainty or error inherent in data gathered from physical experiments, which in turn affect the accuracy of the solution arrived at numerically.

Example. Using a meterstick to measure the length of a wooden plank results in some error, as the meterstick can only measure up to a tenth of a centimeter (or a millimeter). Thus, in measuring the plank in Figure 1.1, we would have to estimate since the end of the plank is in between two of the millimeter marks, resulting in a measurement with an uncertainty of up to a millimeter.

Figure 1.1. Using a meterstick to measure a wooden plank
Retrieved from: http://slideplayer.com/slide/7080663/

If this data is then fed to a program which interpolates a function based on the given lengths, for example, the resulting function is going to carry with it some degree of uncertainty.

(4) Machine errors

Machine errors refer to errors which are inherent in the use of floating point numbers in machines. In particular, this can include underflow/overflow errors and rounding/chopping errors. Both will be discussed in detail in the following chapter.

To give the reader a brief overview on the latter, however, the computer can only allocate a part of its memory to storing certain numbers. Because there are numbers which could go on forever when represented using decimals (i.e., the fraction $\frac{1}{3}$ or the number π), they end up being chopped or rounded off by the machine.

Example. Using $\frac{1}{3}$ as an example, $\frac{1}{3} \approx 0.33\overline{3}$. Say the computer can only store up to three decimal places, so that to the computer, $\frac{1}{3} = 0.333$. Thus, there is an error of about $0.000333\overline{3}$.

(5) Mathematical truncation error

Truncation error refers to the error associated with truncating (or cutting short) a mathematical procedure and is often the subject of numerical analysis. Certain operations (such as the infinite series) could theoretically go on forever as infinite processes but in the machine, they are often approximated as finite processes.

Example. Using the Taylor series as an example, the Taylor series of e^x is given by

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{1.3}$$

Therefore, using (1.3), $e^{0.1}$is

$$e^{0.1} = 1 + (0.1) + \frac{(0.1)^2}{2!} + \frac{(0.1)^3}{3!} + \cdots = \mathbf{1.10517091808} \tag{1.4}$$

However, the computer can't compute up to infinity. Hence, it ends up truncating the series. Say it computes the series up to the third term.

Hence,

$$e^{0.1} \approx 1 + (0.1) + \frac{(0.1)^2}{2!} \approx \mathbf{1.105} \tag{1.5}$$

which results in an error of about 0.00017091808.

After having discussed the sources of errors, we can now look into how we can quantify them.

Measuring errors

There are primarily two particular types of error: **absolute** and **relative.**

In particular, the type of error will vary depending on whether or not we know the true value of the solution.

(a) when the true value is known

(Absolute) true error is the difference between the true value and the approximate value. **Relative true error**, on the other hand, is the ratio of the true error to the true value.

Mathematically,

$$true\ error\ (TE) = true\ value - approximate\ value \tag{1.6}$$

$$relative\ true\ error\ (RTE) = \frac{true\ error}{true\ value} = \frac{true\ value - approximate\ value}{true\ value} \tag{1.7}$$

Example.

Using (1.4) and (1.5), we know that

$$e^{0.1} = 1.10517091808$$

and that by using the Taylor series up to the third term as an approximation, $e^{0.1}$ is approximately

$$e^{0.1} \approx 1.105$$

Thus, using (1.6) and (1.7),

$$\begin{aligned} TE &= true\ value - approximate\ value \\ &= 1.10517091808 - 1.105 \\ &= \mathbf{0.00017091808} \end{aligned}$$

as computed earlier

$$\begin{aligned} RTE &= \frac{true\ error}{true\ value} \\ &= \frac{0.00017091808}{1.10517091808} \\ &= 0.0001546530652 \\ &= \mathbf{0.01546530652}\% \end{aligned}$$

Sometimes, it is advisable to take the absolute value of the relative true error. Therefore,

$$|RTE| = \mathbf{0.01546530652}\%$$

(b) when the true value is unknown

(Absolute) approximate error is the difference between the present approximation and the previous approximation. **Relative approximate error**, on the other hand, is the ratio of the approximate error to the present approximation.

Mathematically,

$$approximate\ error\ (AE) = present\ - previous \tag{6}$$

$$relative\ approximate\ error\ (RAE) = \frac{approximate\ error}{present} = \frac{present\ - previous}{present} \tag{7}$$

These measures are useful if the true value cannot be determined analytically or through a calculator. Oftentimes, the absolute and relative approximate errors are useful because in most numerical methods, the approximations are expected to converge to a certain value and hence the absolute and relative approximate errors are also expected to approach zero as this happens. Care should be exercised, however, since the algorithm may converge to the wrong value (which will be discussed in succeeding chapters)

Example.

Returning to the Taylor series example, we know from (3) that

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Using this, we will try to find an approximation for $e^{0.1}$ by successively adding terms in the Taylor series. The first term will be treated as the first iteration, the sum of the first and second terms as the second iteration and so on.

Thus, for the first iteration

$$e^{0.1} = 1$$

and for the second iteration,

$$e^{0.1} = 1 + 0.1 = 1.1$$

Therefore,

$$\begin{aligned} AE &= present\ - previous \\ &= 1.1 - 1 \\ &= \mathbf{0.1} \end{aligned}$$

$$\begin{aligned} RAE &= \frac{approximate\ error}{present} \\ &= \frac{0.1}{1.1} \\ &= 0.0909091 \\ &= \mathbf{9.09091}\% \end{aligned}$$

Again, it is sometimes advisable to take the absolute value of the relative approximate error. Therefore,

$$|RAE| = \mathbf{9.09091}\%$$

Knowing now how to compute for error measures, we can now discuss criteria that physicists use to ascertain that the error is within a tolerable level.

Minimizing errors

There are two main methods by which one can assure that a numerical method obtains an answer whose error is acceptable: (1) by using a tolerance or (2) pre-specifying the number of significant digits.

METHOD 1 **Using a tolerance**

The user can pre-specify a tolerance level ϵ. The absolute value of the relative approximate error should be less than this tolerance level ϵ (In symbols, $|RAE| \leq \epsilon$) for the approximation to be deemed acceptable.

Example.

Using the Taylor series example and setting $\epsilon = 0.0001\%$, we can compute the relative approximate error for the succeeding iterations, using the earlier computations as a guide. In doing so, we find that:

no. of terms	f(x)	|RAE| (in %)
1	1.0000000000	-
2	1.1000000015	9.09091
3	1.1050000018	0.45249
4	1.1051666685	0.01508
5	1.1051708352	0.00038
6	1.1051790185	0.00001

Table 1.1 Using $\epsilon = 0.0001\%$

Note that the above table was obtained from automating the computations using Fortran, so there may be floating point errors in the computation for f(x) and the relative approximate error.

Table 1.1 shows that when we set $\epsilon = 0.0001\%$, we need to use at least 6 terms of the Taylor series for $|RAE| \leq \epsilon$. Thus,

$$e^{0.1} \approx \mathbf{1.1051790185}$$

METHOD 2 **Pre-specifying the number of significant digits**

The user can also pre-specify the number of significant digits needed. The absolute value of the relative approximate error should be less than 0.5 multiplied by 10^{2-m}, where m is the number of significant digits pre-specified (In symbols, $|RAE| \leq 0.5 \times 10^{2-m}$) for the approximation to be deemed acceptable.

Example.

Using the Taylor series example and setting $m = 9$, we can compute the relative approximate error for the succeeding iterations, using the earlier computations as a guide. In doing so, we find that:

no. of terms	f(x)	|RAE| (in %)	no. of significant digits
1	1.0000000000	-	0
2	1.1000000015	9.09091	1
3	1.1050000018	0.45249	3
4	1.1051666685	0.01508	4
5	1.1051708352	0.00038	6
6	1.1051790185	0.00001	7
7	1.1051709199	0.00000	9

Table 2.2 Using m = 9

Note that the above table was obtained from automating the computations using Fortran, so there may be floating point errors in the computation for f(x) and the relative approximate error.

Table 1.2 shows that when we set $m = 9$, we need to use at least 7 terms of the Taylor series for $|RAE| \leq 0.5 \times 10^{2-m}$. Thus,

$$e^{0.1} \approx \mathbf{1.1051709199}$$

After discussing the three main topics, we can now proceed to implementing the procedures discussed in this chapter in code using Fortran.

Minimizing error (Fortran example)

To avoid doing the tedious calculations discussed above over and over, we will use Fortran to minimize the relative approximate error through a pre-set tolerance. Minimizing the RAE through a pre-specified number of significant digits using Fortran is relatively straightforward and closely parallels the procedures outlined here. For this code, we will approximate $e^{0.1}$ using the Taylor series expansion of e^x as discussed in previous examples.

Figure 1.2 shows a flowchart detailing the basic steps of the algorithm.

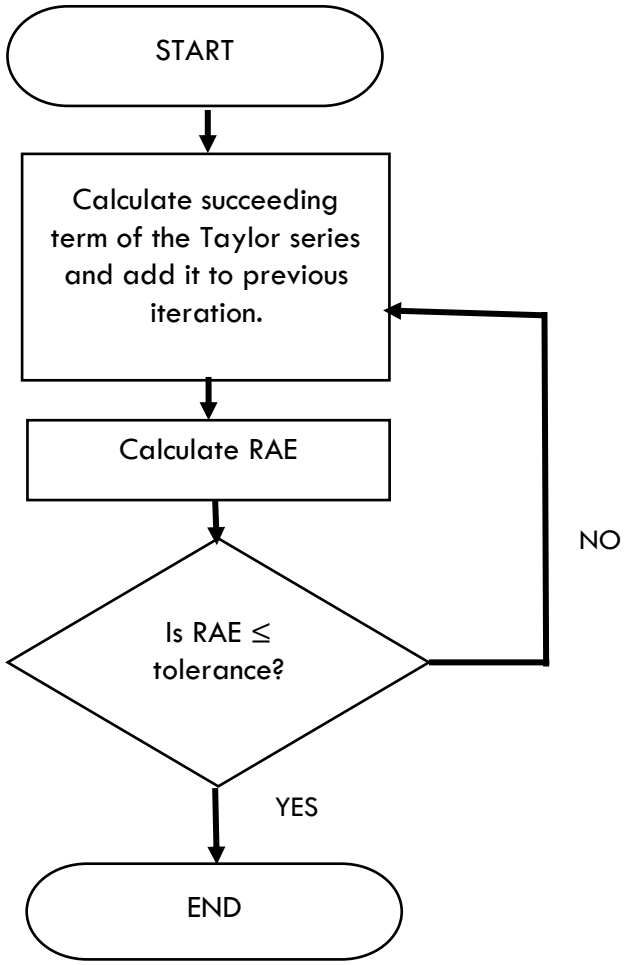

Figure 1.2. *Flowchart for error minimization using pre-set tolerance*

```
!=========================================================================================
! Code_1_tolerance.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the number of iterations needed given a pre-set tolerance.
!=========================================================================================

program tolerance

	implicit none
	!-----------------------------------------------------------------------------------
	! Declare variables
	real*16 :: error = 1.0, tol = 0.0001
	integer :: i
	character(len = 1) :: start_error = "-"
	!-----------------------------------------------------------------------------------

	!-----------------------------------------------------------------------------------
	! Print heading
	write(*,"(a12, 3x, a4, 13x, a4)") "no. of terms", "f(x)", "%RAE"
	write(*,*)
	!-----------------------------------------------------------------------------------
```

The program starts by declaring the necessary variables and printing the heading of the output. `tol` is set to 0.0001, but the program may be edited to set the tolerance to the desired level.

```
do i = 1, 30
	error = abs((taylorexp(i) - taylorexp(i - 1)) / taylorexp(i)) * 100

	if(i == 1) then
		write(*,"(i2, 13x, f12.10, 5x, a1, 10x, i1)") i, taylorexp(i), start_error
	else
		write(*,"(i2, 13x, f12.10, 3x, f10.5, 3x, i1)") i, taylorexp(i), error
	end if

	if(error <= tol) then
		exit
	endif
end do
```

A do loop that runs from i = 1 to 30 (any arbitrary number will do here) is then used to perform the calculations specified in the flowchart. The program prints out the number of terms/iterations (`i`), the Taylor series approximation (`taylorexp(i)`) and the relative approximate error per iteration (`error`). For the first term, there is no relative approximate error so we print `start_error` instead, which is simply the character "–".

`error` follows the computation of the relative approximate error, as specified in (1.7). The computation of the Taylor series approximation is relegated to the function `taylorexp`, which will be discussed shortly afterwards. Once the condition `error <= tol` is met, the program exits.

```
function taylorexp(i) result(taylorexpeval)
       real*16 :: taylorexpeval
       integer :: i, count

       count = 0
       taylorexpeval = 0

       do while(count < i)
              taylorexpeval = taylorexpeval + 0.1 ** count / factorial(count)
               count = count + 1
       end do
end function taylorexp
```

The function taylorexp computes the Taylor series approximation for every iteration. It takes i as input and outputs taylorexpeval. Using the counter variable count, we add the Taylor series terms 0.1 ** count/factorial(count) to the variable taylorexpeval while count < i (we use this condition since we initialized count to be 0. If count = 1 initially, for example, we would use the condition count <= i instead).

Note that the Taylor series terms use the function factorial, which will be discussed after this.

```
function factorial(n) result(factorialeval)
       integer :: n, factorialeval, i

       factorialeval = 1
       do i = 1 , n
              factorialeval = factorialeval * i
       end do

       if(n == 0) then
              factorialeval = 1
       end if
end function factorial
```

The function factorial computes for the factorial of a given number and is used in the taylorexp function discussed earlier. It takes n as input and outputs factorialeval. Using a do loop running from i = 1 to n, we multiply variable i to the variable factorialeval. If n = 0, then we let factorialeval = 1.

The full program is as follows:

```
!=============================================================================================
! Code_1_tolerance.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the number of iterations needed given a pre-set tolerance.
!=============================================================================================

program tolerance

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	real*16 :: error = 1.0, tol = 0.0001
	integer :: i
	character(len = 1) :: start_error = "-"
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Print heading
	write(*,"(a12, 3x, a4, 13x, a4)") "no. of terms", "f(x)", "%RAE"
	write(*,*)
	!---------------------------------------------------------------------------------------


	!---------------------------------------------------------------------------------------
	! This implements the algorithm.
	! This also prints out the no. of terms, Taylor series approx. and the RAE per
	iteration.
	!---------------------------------------------------------------------------------------


	do i = 1, 30
		error = abs((taylorexp(i) - taylorexp(i - 1)) / taylorexp(i)) * 100

		if(i == 1) then
			write(*,"(i2, 13x, f12.10, 5x, a1, 10x, i1)") i, taylorexp(i),
			start_error
		else
			write(*,"(i2, 13x, f12.10, 3x, f10.5, 3x, i1)") i, taylorexp(i), error
		end if

		if(error <= tol) then
			exit
		endif
	end do
```

```
        write(*,*) ""
        write(*,"(a35, f12.10, 1x, a1)") "exp(0.1) is approximately equal to ",
        taylorexp(i),  " ."

        contains
        !-------------------------------------------------------------------------------
        ! This function calculates the factorial of a number
        function factorial(n) result(factorialeval)
                integer :: n, factorialeval, i
                factorialeval = 1
                do i = 1 , n
                        factorialeval = factorialeval * i
                end do

                if(n == 0) then
                        factorialeval = 1
                end if
        end function factorial
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! This function calculates the Taylor expansion of exp(0.1) per iteration. Edit if
        ! needed
        function taylorexp(i) result(taylorexpeval)
                real*16 :: taylorexpeval
                integer :: i, count

                count = 0
                taylorexpeval = 0

                do while(count < i)
                        taylorexpeval = taylorexpeval + 0.1 ** count / factorial(count)
                        count = count + 1
            end do

         end function taylorexp
        !-------------------------------------------------------------------------------

end program tolerance
```

Executing the program,

```
no. of terms   f(x)            %RAE

 1             1.0000000000    -
 2             1.1000000015     9.09091
 3             1.1050000018     0.45249
 4             1.1051666685     0.01508
 5             1.1051708352     0.00038
 6             1.1051709185     0.00001

exp(0.1) is approximately equal to 1.1051709185

Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.
```

Figure 1.2. *Output of program* *Code_1_tolerance.f90*

We find out that $e^{0.1} \approx \mathbf{1.1051709185.}$

Propagation of errors

In this section, we consider how error propagates in a calculation where the numbers involved have some uncertainty associated with them. First, we consider the propagation of errors in basic operations, then we move on to how error propagates in a function.

CASE 1 Basic operations

(a) Multiplication and Division

Say we have two numbers x_t and y_t. Let x_a and y_a be the actual numbers used in calculation and let α and β be the errors associated with x_t and y_t, respectively, so that:

$$x_t = x_a + \alpha \tag{1.8}$$
$$y_t = y_a + \beta \tag{1.9}$$

For the error associated with multiplying x and y, we have

$$\begin{aligned} x_t y_t &= (x_a + \alpha)(y_a + \beta) \\ &= x_a y_a + \alpha y_a + \beta x_a + \alpha\beta \end{aligned}$$

Therefore,

$$\begin{aligned} TE(x_a y_a) &= \alpha y_a + \beta x_a + \alpha\beta \\ &= \alpha(y_t - \beta) + \beta(x_t - \alpha) + \alpha\beta \\ &= \alpha y_t + \beta x_t - \alpha\beta \end{aligned}$$

$$RTE(x_a y_a) = \frac{\alpha y_t + \beta x_t - \alpha\beta}{x_t y_t}$$
$$= \frac{\alpha}{x_t} + \frac{\beta}{y_t} - \left(\frac{\alpha}{x_t}\right)\left(\frac{\beta}{y_t}\right)$$
$$= RTE(x_a) + RTE(y_a) - RTE(x_a)RTE(y_a)$$

If $|RTE(x_a)|$ and $|RTE(y_a)| \ll 1$ (as they normally are), then,

$$RTE(x_a y_a) = RTE(x_a) + RTE(y_a) \tag{1.10}$$

and the relative true errors are additive under multiplication.

Using a similar argument for division,

$$\frac{x_t}{y_t} = \frac{x_a + \alpha}{y_a + \beta}$$

Therefore,

$$\begin{aligned} TE\left(\frac{x_a}{y_a}\right) &= \frac{x_a + \alpha}{y_a + \beta} - \frac{x_a}{y_a} \\ &= \frac{x_t}{y_t} - \frac{x_t - \alpha}{y_t - \beta} \\ &= \frac{(x_t)(y_t - \beta) - (x_t - \alpha)(y_t)}{y_t(y_t - \beta)} \end{aligned}$$

$$= \frac{\alpha y_t - \beta x_t}{y_t(y_t - \beta)}$$

$$RTE\left(\frac{x_a}{y_a}\right) = \frac{\alpha y_t - \beta x_t}{y_t(y_t - \beta)} \cdot \frac{1}{\frac{x_t}{y_t}}$$

$$= \left(\frac{\alpha}{x_t}\right)\left(\frac{y_t}{y_t - \beta}\right) - \left(\frac{\beta}{y_t}\right)\left(\frac{y_t}{y_t - \beta}\right)$$

$$= \frac{RTE(x_a) - RTE(y_a)}{1 - RTE(y_a)}$$

If $|RTE(x_a)|$ and $|RTE(y_a)| \ll 1$ (as they normally are), then,

$$RTE\left(\frac{x_a}{y_a}\right) = RTE(x_a) - RTE(y_a) \tag{1.11}$$

and the relative true errors are subtractive under division.

Thus, *relative* errors do not propagate rapidly under multiplication and division.

It can also be shown that relative errors are additive/subtractive in the multiplication/division of powers. That is,

$$RTE({x_a}^m {y_a}^n) = mRTE(x_a) + nRTE(y_a) \tag{1.12}$$

$$RTE\left(\frac{{x_a}^m}{{y_a}^n}\right) = mRTE(x_a) - nRTE(y_a) \tag{1.13}$$

(b) Addition and Subtraction

For the error associated with adding or subtracting x and y, we have

$$x_t \pm y_t = (x_a + \alpha) \pm (y_a + \beta)$$

Therefore,

$$\begin{aligned} TE(x_a \pm y_a) &= (x_a + \alpha) \pm (y_a + \beta) - (x_a \pm y_a) \\ &= \alpha \pm \beta \\ &= TE(x_a) \pm TE(y_a) \end{aligned} \tag{1.14}$$

and the absolute true errors are subtractive under division.

Thus, *absolute* errors do not propagate rapidly under addition and subtraction.

However, relative errors may propagate quickly under addition and subtraction. This is important to note, especially since we use relative and not absolute errors to minimize the uncertainty in our calculations. Thus, when using relative error as the criterion to minimize uncertainty, it is preferable to resort to multiplication and division as opposed to addition and subtraction to minimize uncertainty error.

Example. *adapted from Atkinson (2008)*

The roots of a quadratic equation are given by the quadratic formula

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.15}$$

Say we have the equation $x^2 - 26x + 1 = 0$. Using (1.15),

$$r_1 = 13 + \sqrt{168}$$
$$r_2 = 13 - \sqrt{168}$$

If the machine can only store up to five digits, then $\sqrt{168} = 12.961$ and

$$r_{1a} = 13 + 12.961 = 25.961$$
$$r_{2a} = 13 - 12.961 = 0.039$$

However, there would be a rounding error now associated with both roots. Thus, we can calculate the relative error for both

$$|RTE(r_{1a})| = 1.85 \times 10^{-5}$$
$$|RTE(r_{2a})| = 0.01250$$

Say instead of directly adding or subtracting $\sqrt{168}$, we do this instead:

$$r_2 = 13 - \sqrt{168} \cdot \frac{13 + \sqrt{168}}{13 + \sqrt{168}}$$
$$= \frac{1}{13 + \sqrt{168}}$$

Therefore,

$$r_{2a} = \frac{1}{13 + 12.961} = 0.0385$$

There are two errors associated with this new method: the error associated with $\sqrt{168}$ and the error propagated through division. However, since division results in minimal error propagation, we expect the error to be slightly smaller or larger compared to the original relative error.

Computing for the relative error,

$$|RTE(r_{2a})| = 4.82 \times 10^{-4}$$

Thus, the new method actually *improves* upon the previous one in terms of relative error for the second root. One can check and see that the new method is actually worse than the first one for the first root in terms of relative error, but this is simply a case of error trade-off between the two roots.

CASE 2 Functions

In general, given $v = f(w, x, y, \dots)$ where $w, x, y, \dots$ are *independent* variables, we can obtain the uncertainty in x by getting its total differential; that is,

$$dv = \left(\frac{\partial z}{\partial w}\right) dw + \left(\frac{\partial z}{\partial x}\right) dx + \left(\frac{\partial z}{\partial y}\right) dy + \cdots \tag{1.16}$$

Thus,

$$\Delta v = \left|\frac{\partial z}{\partial w}\right| \Delta w + \left|\frac{\partial z}{\partial x}\right| \Delta x + \left|\frac{\partial z}{\partial y}\right| \Delta y + \cdots \tag{1.17}$$

One can check to see that (1.17) applies to all the basic operations outlined in case 1.

Example. *adapted with changes from Kaw (2011)*

The strain ϵ in an axial member is given by:

$$\epsilon = \frac{F}{h^2 E} \tag{1.18}$$

where F = axial force (in N)
h = length or width of the cross section (in m)
E = Young's modulus (in Pa)

Say $F = 25 + 0.1\ N$
$h = 7 + 0.001\ mm$
$E = 30 + 4\ GPa$

Then applying (1.18),

$$\epsilon = \frac{25\ N}{(7 \times 10^{-3} m)^2 (30 \times 10^9 Pa)} = \mathbf{17.0068 \times 10^{-6}}$$

To get the error associated with ϵ, we use (1.16).

$$\begin{aligned}
\Delta\epsilon &= \left|\frac{\partial \epsilon}{\partial F}\right| \Delta F + \left|\frac{\partial \epsilon}{\partial h}\right| \Delta h + \left|\frac{\partial \epsilon}{\partial E}\right| \Delta E \\
&= \left(\frac{1}{h^2 E}\right) \Delta F + \left(\frac{2F}{h^3 E}\right) \Delta h + \left(\frac{F}{h^2 E^2}\right) \Delta E \\
&= \left[\frac{1}{(7 \times 10^{-3} m)^2 (30 \times 10^9 Pa)}\right] 0.1N + \left[\frac{2(25\ N)}{(7 \times 10^{-3} m)^3 (30 \times 10^9 Pa)}\right] [0.001 \times 10^{-3} m] \\
&\qquad + \left[\frac{25\ N}{(7 \times 10^{-3} m)^2 (30 \times 10^9 Pa)^2}\right] [4 \times 10^9\ Pa] \\
&= \mathbf{2.3405 \times 10^{-6}}
\end{aligned}$$

Thus,

$$\epsilon = \mathbf{17.0068 \times 10^{-6} \pm 2.3405 \times 10^{-6}}$$

Application – Election disaster caused by rounding error

Retrieved from: *Rounding error changes parliament makeup*
Deborah Weber-Wulff
http://mate.uprh.edu/~pnegron/notas4061/parliament.htm

Figure 1.3. *The state of Schleswig-Holstein, Germany*

In 1992, a rounding error had thrown off elections in the German state of Schleswig-Holstein. Federal election laws at the time stated that no party with less than 5% of the vote may hold a seat in the parliament. Parliament seats are then distributed in two ways: by direct vote and by list. Persons who win precinct votes are first seated, after which the remaining seats are allocated by list to parties with 5% of the vote or more through a complicated system.

During that time, the Green Party had registered 5% of the total vote, which, due to the complicated system at the time, meant that the competing Social Democrats Party (SPD) could no longer get some of their listed members a seat in the parliament. However, it was soon discovered that the Green Party had only fielded 4.97% of the vote, and that the program they used only outputted percentages up to one decimal place. Hence, the SPD *did* get some of their members seated after all while the Green Party ended up not winning seats at all.

CHAPTER 2: *MACHINE REPRESENTATION OF NUMBERS*

Introduction

We commonly use numbers in the *decimal (or base-10) system*. What this means is that any number can be expressed as a linear combination of powers of 10. Take the number 1025.03125, for instance. We can write this number as

$$1025.03125 = (1 \times 10^3) + (0 \times 10^2) + (2 \times 10^1) + (5 \times 10^0) + (0 \times 10^{-1}) + (3 \times 10^{-2}) + (1 \times 10^{-3}) + (2 \times 10^{-4}) + (5 \times 10^{-5})$$

However, the machine does not normally store numbers in its natural base-10 representation. Instead, it can only store numbers in its *binary (or base-2) form*. A number like 1025.03125, for example, would be stored as 10000000001.00001 or $1.000000000100001 \times 2^{-10}$ in a machine, which would make sense, since

$$1025.03125 = (1 \times 2^{10}) + (0 \times 2^9) + (0 \times 2^8) + (0 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (0 \times 2^{-4}) + (1 \times 2^{-5})$$

In addition, , machines often use what is known as *floating point representation*, where the number is converted to scientific notation and then bits are allocated with regards to the sign of the number, the sign of the exponent, the base (or *mantissa*) and the exponent (or *ficand*).

Someone trained in numerical analysis must know how numbers are represented in a machine, as discrepancies between a number and its machine representation can oftentimes lead to errors. Thus, he/she must:

(1) know how to convert a number from base-10 to base-2 and vice versa,

(2) know how to convert a number from its decimal representation to its binary floating-point representation and vice versa,

(3) know the errors associated with the floating-point representation of numbers and calculate the associated machine epsilon.

This chapter intends to discuss the machine representation of errors, particularly the topics outlined above. We begin with the binary representation of numbers.

Binary representation

Having discussed the decimal and binary representations in the introduction, we now proceed to converting numbers from one system to another.

CASE 1 decimal to binary

(a) integers

To convert integers from decimal to binary, one simply needs to follow the following algorithm, summarized in Figure 2.1.

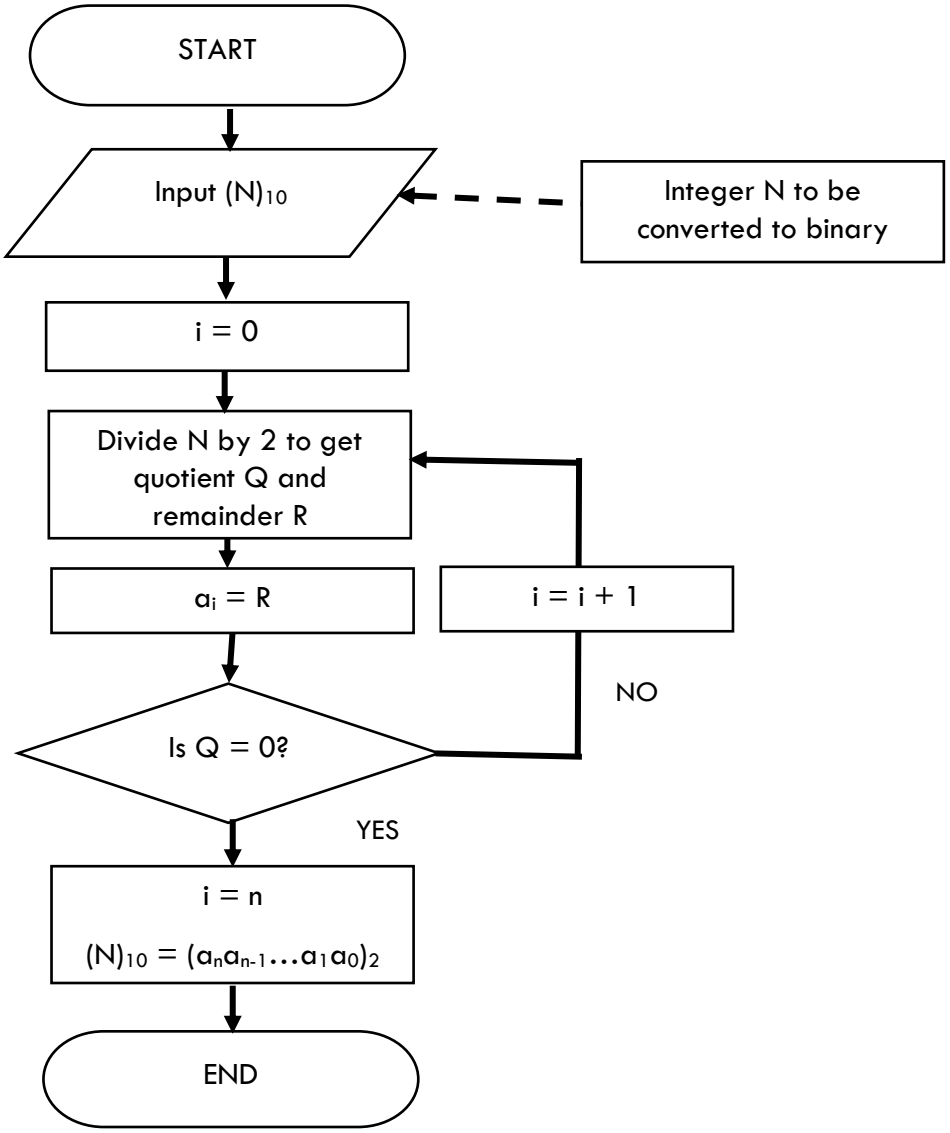

Figure 2.1. *Flowchart for conversion of integers from decimal to binary*

Adapted from Kaw (2011)

Example 1 Using the example from earlier: $(1025)_{10}$,

i	OPERATION	=	QUOTIENT	REMAINDER
0	1025/2	=	512	$1 = a_0$
1	512/2	=	256	$0 = a_1$
2	256/2	=	128	$0 = a_2$
3	128/2	=	64	$0 = a_3$
4	64/2	=	32	$0 = a_4$
5	32/2	=	16	$0 = a_5$
6	16/2	=	8	$0 = a_6$
7	8/2	=	4	$0 = a_7$
8	4/2	=	2	$0 = a_8$
9	2/2	=	1	$0 = a_9$
10	1/2	=	0	$1 = a_{10}$
			Q = 0 END	

***Table 2.1**. Conversion of $(1025)_{10}$ from decimal to binary*

Hence,

$$(1025)_{10} = (a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0)_2$$
$$= (\mathbf{10000000001})_\mathbf{2}$$

(b) fractions

To convert integers from decimal to binary, one simply needs to follow the following algorithm, summarized in Figure 2.2 on the next page.

Example 2 Using the example from earlier: $(0.03125)_{10}$,

i	OPERATION	=	FRACTIONAL	WHOLE
-1	0.03125 x 2	=	0.0625	$0 = a_{-1}$
-2	0.0625 x 2	=	0.125	$0 = a_{-2}$
-3	0.125 x 2	=	0.25	$0 = a_{-3}$
-4	0.25 x 2	=	0.5	$0 = a_{-4}$
-5	0.5 x 2	=	0	$1 = a_{-5}$
			F = 0 END	

***Table 2.2**. Conversion of $(0.03125)_{10}$ from decimal to binary*

Hence,

$$(0.01325)_{10} = (a_{-1}a_{-2}a_{-3}a_{-4}a_{-5})_2$$
$$= (\mathbf{0.00001})_\mathbf{2}$$

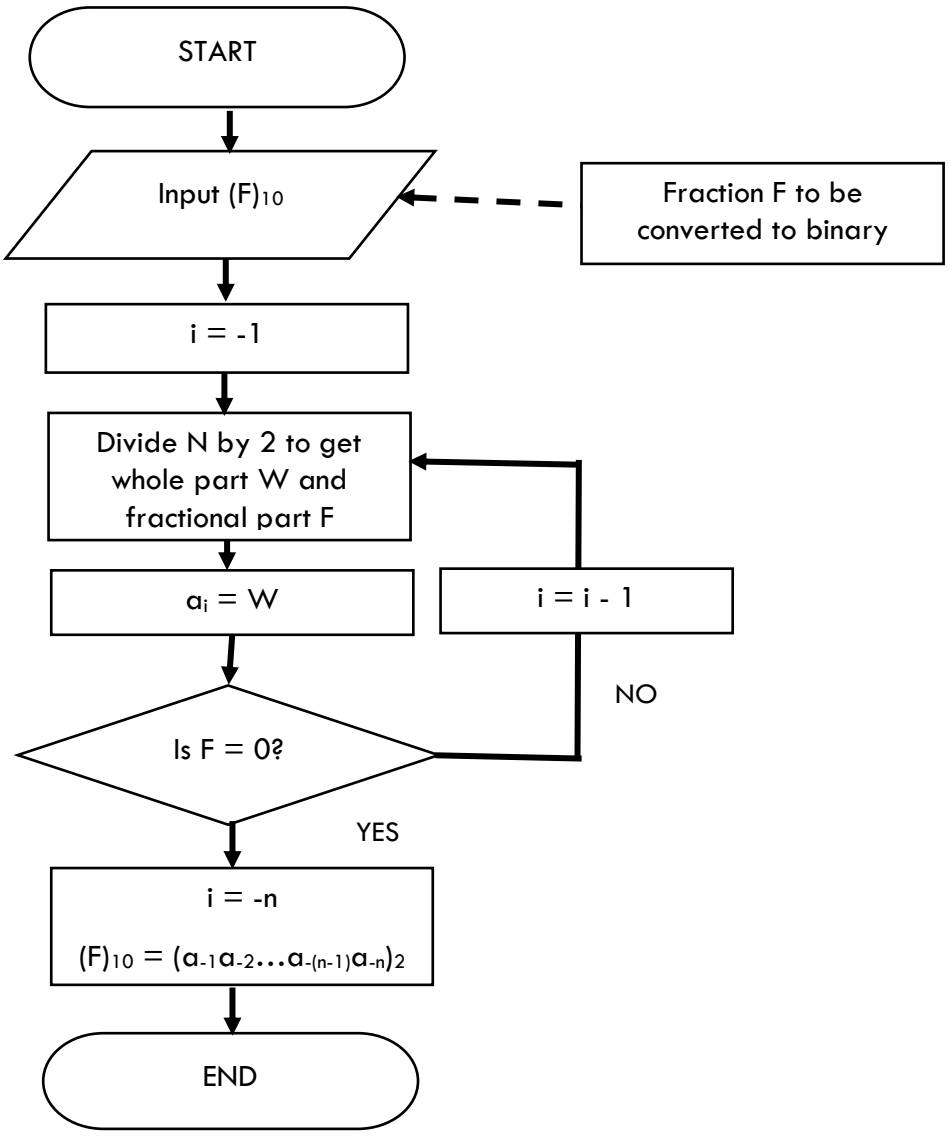

Figure 2.2. *Flowchart for conversion of fractions from decimal to binary*

Adapted from Kaw (2011)

CASE 2 binary to decimal

(a) integers

To convert integers from binary to decimal, one simply needs to follow the following algorithm, summarized in Figure 2.3 on the next page.

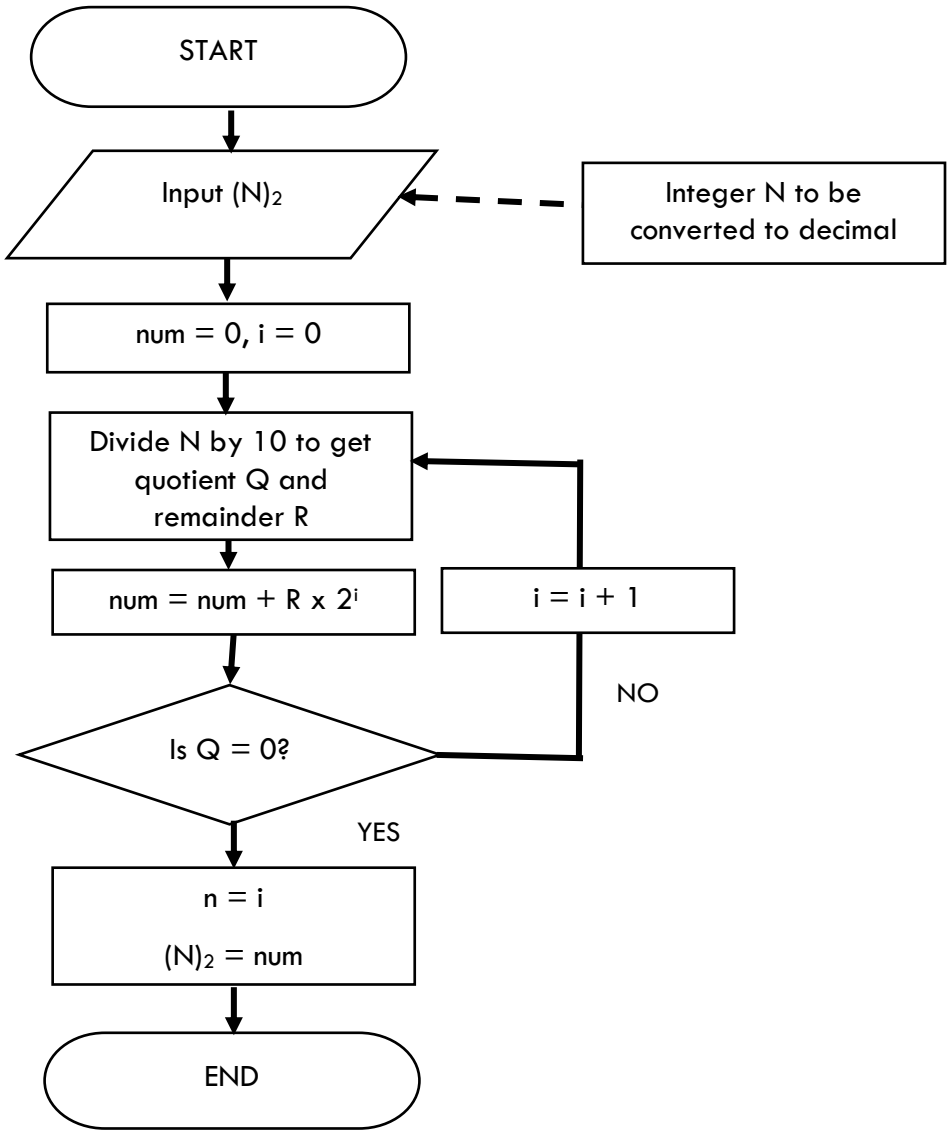

Figure 2.3. *Flowchart for conversion of integers from binary to decimal*

Example 3 Using the example from earlier: $(10000000001)_2$,

i	OPERATION	=	QUOTIENT	REMAINDER	X 2^i + NUM	=	NUM
0	10000000001/10	=	1000000000	1	$2^0/1$	=	1
1	1000000000/10	=	100000000	0	$2^1/2$	=	1
2	100000000/10	=	10000000	0	$2^2/4$	=	1
3	10000000/10	=	1000000	0	$2^3/8$	=	1
4	1000000/10	=	100000	0	$2^4/16$	=	1
5	100000/10	=	10000	0	$2^5/32$	=	1
6	10000/10	=	1000	0	$2^6/64$	=	1
7	1000/10	=	100	0	$2^7/128$	=	1
8	100/10	=	10	0	$2^8/256$	=	1
9	10/10	=	1	0	$2^9/512$	=	1
10	1/10	=	0	1	$2^{10}/1024$	=	**1025**
			Q = 0 END				

***Table 2.3**. Conversion of $(10000000001)_2$ from decimal to binary*

Hence,

$$(10000000001)_2 = (\mathbf{1025})_{\mathbf{10}}$$

(b) fractions

To convert fractions from decimal to binary, one simply needs to follow the following algorithm, summarized in Figure 2.4 on the next page.

Example 4 Using the example from earlier: $(0.00001)_2$,

i	OPERATION	=	FRACTIONAL	WHOLE	X 2^{-i} + NUM	=	NUM
1	0.00001 x 10	=	0.0001	0	$2^{-1}/0.5$	=	0
2	0.0001 x 10	=	0.001	0	$2^{-2}/0.25$	=	0
3	0.001 x 10	=	0.01	0	$2^{-3}/0.125$	=	0
4	0.01 x 10	=	0.1	0	$2^{-4}/0.0625$	=	0
5	0.1 x 10	=	0	1	$2^{-5}/0.03125$	=	**0.03125**
			F = 0 END				

***Table 2.4**. Conversion of $(0.00001)_2$ from decimal to binary*

Hence,

$$(0.00001)_2 = (\mathbf{0.03125})_{\mathbf{10}}$$

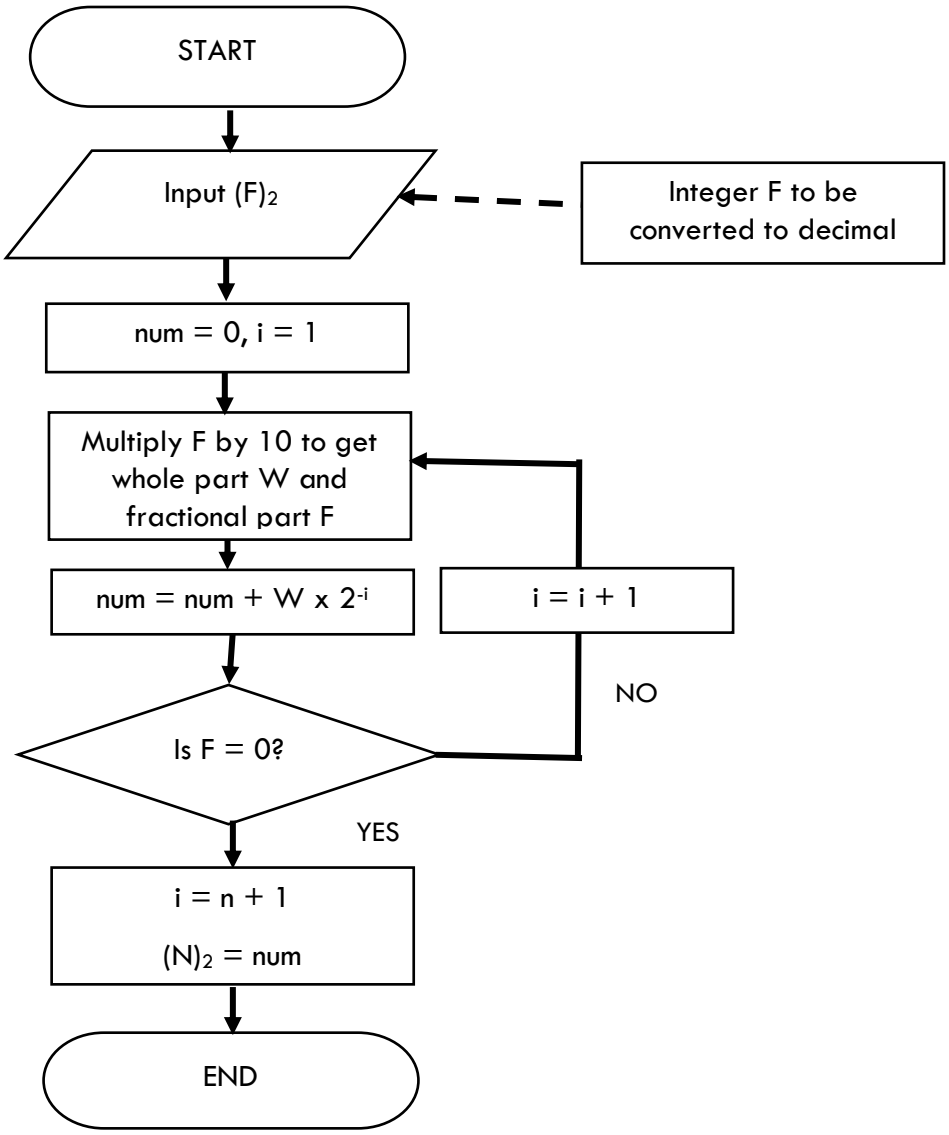

Figure 2.4. *Flowchart for conversion of fractions from binary to decimal*

Binary representation (Fortran example)

We can use Fortran to convert a number from base-10 to base-2 and vice-versa using the algorithms outlined in the flowcharts. For this code, the user will specify the number to be converted.

CASE 1 decimal to binary

```
!=============================================================================================
! Code_2_decimal_to_binary.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program converts a number from decimal to binary.
!=============================================================================================

program decimal_to_binary

implicit none
       !-------------------------------------------------------------------------------------
       ! Declare variables
       real*16 :: num, frac, frac_frac
       integer :: rem, whole, frac_whole, i = 1, j = 1, k = 1, l = 1
       integer, dimension(100) :: whole_digits, frac_digits
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! User input, Print output
       write(*,*) "Please enter a number in base 10."
       read(*,*) num
       write(*,*) ""
       write(*,"(a10, 2x, f20.10, 2x, a13)") "The number", num, "in binary is: "
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! Separate number into fractional and non-fractional parts
       whole = int(num)
       frac = num - whole
       !-------------------------------------------------------------------------------------
```

The program starts by declaring the necessary variables.

Since the procedures for converting integers and fractions to binary are distinct, we separate the number into its whole and fractional parts, storing the whole part in *whole* and the fractional part in *frac*. The variables relating to converting the whole part to binary (*rem*, counter *i*) and the fractional part to binary (*frac_whole*, *frac_digits*, counter *k*) are also declared.

Note also that the arrays *whole_digits* and *frac_digits* have also been declared. In the case of converting the whole part to binary, we will run through the remainders per iteration in reverse. On the other hand, in converting the fractional part to binary, we will run through the whole parts per iteration. Using arrays and counters (*j* and *l*, in this case) makes both processes easier.

```
!-------------------------------------------------------------------------------------------
! This converts the whole part to binary.
do while(whole /= 0)
       rem = mod(whole, 2)
       whole = whole / 2
       whole_digits(i) = rem
       i = i + 1
end do
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! This prints the binary representation of the whole part.
do while(j < i)
       write(*, "(i1)", advance="no") whole_digits(i - j)
       j = j + 1
end do
!-------------------------------------------------------------------------------------------
```

A do-while loop is then used to perform the calculations specified in Figure 2.1. Because the remainders per iteration constitute the digits of the number's binary representation, we store `rem` in the array `whole_digits`

Note from the earlier code snippet that unlike in Figure 2.1, `i` is initially set to 1. This is because in Fortran, array indexing starts at 1. If we wanted to follow the flowchart and start with `i = 0`, `whole_digits(i)` should become `whole_digits(i + 1)`. Once the program finishes running through the loop, `i = n + 1`.

Another do-while loop is used to print the whole number's binary representation. Because we need to run through the remainders per iteration *backwards*, we should print from the n^{th} element up to the 1^{st} element; ergo, we should take the $(i - j)^{th}$ element of whole_digits, where `j` runs from 1 to `n-1`. Thus, the program runs until `j = n - 1` or as long as `j < i`.

```
!-------------------------------------------------------------------------------------------
! This converts the fractional part to binary.
do while(frac_frac /= 0)
       frac = frac * 2
       frac_whole = int(frac)
       frac_frac = frac - frac_whole
       frac = frac_frac
       frac_digits(k) = frac_whole
       k = k + 1
end do
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! This prints out the binary representation of the fractional part.
write(*, "(a1)", advance="no") "."
```

```
do while(l < k)
       write(*, "(i1)", advance="no") frac_digits(l)
       l = l + 1
end do
!-------------------------------------------------------------------------------------------
```

A do-while loop is also used to perform the calculations specified in Figure 2.2. Because the whole parts per iteration constitute the digits of the number's binary representation, we store `frac_whole` in the array `frac_digits`

Note from the earlier code snippet that unlike in Figure 2.2, `k` is initially set to 1. This is because in Fortran, an array index cannot be negative. If we wanted to follow the flowchart and start with `k = -1`, `frac_digits(i)` should become `frac_digits(-k)` and we should subtract 1 from the counter per iteration (`k = k - 1`). Once the program finishes running through the loop, `k = n + 1`.

We also use a do-while loop to print the binary representation of the fractional part, using the counter `l` running from 1 to `n-1` or as long as `l < k`.

The full program is as follows:

```
!===========================================================================================
! Code_2_decimal_to_binary.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program converts a number from decimal to binary.
!===========================================================================================

program decimal_to_binary

implicit none
       !-------------------------------------------------------------------------------------
       ! Declare variables
       real*16 :: num, frac, frac_frac
       integer :: rem, whole, frac_whole, i = 1, j = 1, k = 1, l = 1
       integer, dimension(100) :: whole_digits, frac_digits
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! User input, Print output
       write(*,*) "Please enter a number in base 10."
       read(*,*) num
       write(*,*) ""
       write(*,"(a10, 2x, f20.10, 2x, a13)") "The number", num, "in binary is: "
       !-------------------------------------------------------------------------------------
```

```
        !-------------------------------------------------------------------------------
        ! Separate number into fractional and non-fractional parts
        whole = int(num)
        frac = num - whole
        !-------------------------------------------------------------------------------


        !-------------------------------------------------------------------------------
        ! This converts the whole part to binary.
        do while(whole /= 0)
                rem = mod(whole, 2)
                whole = whole / 2
                whole_digits(i) = rem
                i = i + 1
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! This prints the binary representation of the whole part.
        do while(j < i)
                write(*, "(i1)", advance="no") whole_digits(i - j)
                j = j + 1
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! This converts the fractional part to binary.
        do while(frac_frac /= 0)
                frac = frac * 2
                frac_whole = int(frac)
                frac_frac = frac - frac_whole
                frac = frac_frac
                frac_digits(k) = frac_whole
                k = k + 1
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! This prints out the binary representation of the fractional part.
        write(*, "(a1)", advance="no") "."
        do while(l < k)
                write(*, "(i1)", advance="no") frac_digits(l)
                l = l + 1
        end do
        !-------------------------------------------------------------------------------
end program decimal_to_binary
```

Executing the program,

```
 Please enter a number in base 10.
1025.03125

The number       1025.0312500000  in binary is:
10000000001.00001
Process returned 0 (0x0)   execution time : 4.525 s
Press any key to continue.
```

Figure 2.5. Output of program Code_2_decimal_to_binary.f90

We find out that $(1025.03125)_{10} = \mathbf{10000000001.00001}$

CASE 2 binary to decimal

```
!=============================================================================================
! Code_3_binary_to_decimal.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program converts a number from binary to decimal.
!=============================================================================================

program binary_to_decimal

    implicit none
    !---------------------------------------------------------------------------------------------
    ! Declare variables
    real*16 :: num, frac, dec_frac = 0
    integer :: whole, digit, frac_digit, dec_whole = 0, i = 0, j = 1
    !---------------------------------------------------------------------------------------------

    !---------------------------------------------------------------------------------------------
    ! User input, Print output
    write(*,*) "Please enter a number in base 2."
    read(*,*) num
    write(*,*) ""
    write(*,"(a10, 2x, f20.10, 2x, a13)") "The number", num, "in binary is: "
    !---------------------------------------------------------------------------------------------

    !---------------------------------------------------------------------------------------------
    ! Separate number into fractional and non-fractional parts
    whole = int(num)
    frac = num - whole
    !---------------------------------------------------------------------------------------------
```

The program starts by declaring the necessary variables.

Since the procedures for converting integers and fractions to binary are distinct, we separate the number into its whole and fractional parts, storing the whole part in *whole* and the fractional part

in *frac*. The variables relating to converting the whole part to binary (*digit*, *dec_whole*, counter *i*) and the fractional part to binary (*frac_digit*, *dec_frac*, counter *j*) are also declared.

```
!-------------------------------------------------------------------------------------------
! This converts the whole part to binary.
do while(whole /= 0)
       digit = mod(whole, 10)
       whole = whole / 10
       dec_whole = dec_whole + digit * 2 ** i
       i = i + 1
end do
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! This converts the fractional part to binary.
do while(frac /= 0)
       frac = frac * 10
       frac_digit = int(frac)
       frac = frac - frac_digit
       dec_frac = dec_frac + real(frac_digit) * 2.0 ** -j
       j = j + 1
end do
!-------------------------------------------------------------------------------------------

write(*, "(f20.10)", advance="no") dec_whole + dec_frac
```

A do-while loop is then used to perform the calculations specified in Figures 2.3 and 2.4. Both programs follow the flowcharts to the letter, so there is not much use discussing both in detail. The last command in the above code snippet then adds the binary representation for both whole and fractional parts, and then prints out the sum.

The full program is as follows:

```
!===========================================================================================
! Code_3_binary_to_decimal.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program converts a number from binary to decimal.
!===========================================================================================

program binary_to_decimal

    implicit none
    !-------------------------------------------------------------------------------------
    ! Declare variables
    real*16 :: num, frac, dec_frac = 0
    integer :: whole, digit, frac_digit, dec_whole = 0, i = 0, j = 1
    !-------------------------------------------------------------------------------------
```

```
    !-------------------------------------------------------------------------------
    ! User input, Print output
    write(*,*) "Please enter a number in base 2."
    read(*,*) num
    write(*,*) ""
    write(*,"(a10, 2x, f20.10, 2x, a13)") "The number", num, "in binary is: "
    !-------------------------------------------------------------------------------

    !-------------------------------------------------------------------------------
    ! Separate number into fractional and non-fractional parts
    whole = int(num)
    frac = num - whole
    !-------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------
       ! This converts the whole part to binary.
       do while(whole /= 0)
              digit = mod(whole, 10)
              whole = whole / 10
              dec_whole = dec_whole + digit * 2 ** i
              i = i + 1
       end do
       !-------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------
       ! This converts the fractional part to binary.
       do while(frac /= 0)
              frac = frac * 10
              frac_digit = int(frac)
              frac = frac - frac_digit
              dec_frac = dec_frac + real(frac_digit) * 2.0 ** -j
              j = j + 1
       end do
       !-------------------------------------------------------------------------------

       write(*, "(f20.10)", advance="no") dec_whole + dec_frac

end program binary_to_decimal
```

Executing the program,

```
 Please enter a number in base 2.
111111111.00001

The number  111111111.0000100000  in binary is:
              511.0312500277
Process returned 0 (0x0)   execution time : 6.205 s
Press any key to continue.
```

Figure 2.6. *Output of program* *Code_3_binary_to_decimal.f90*

We find out that $(111111111.00001)_2 = \mathbf{511.0312500277}$

Note that both programs are subject to floating-point errors, which will be discussed later in the chapter.

Floating point representation

Say a machine can only store up to five digits of a number. Therefore, the machine rounds off a number like 120.78496 and stores it as 120.78. This results in some relative error. In particular, for smaller numbers, the relative error will tend to be high whereas for larger ones, relative error tends to be small.

Example 5

The number 120.78496 is rounded off to 120.78.
Hence,

$$RTE = \frac{120.78496 - 120.78}{120.78496} = 0.0041065\%$$

The number 523410.00496 is rounded off to 523410.
Hence,

$$RTE = \frac{523410.00496 - 523410}{523410.00496} = 9.36 \times 10^{-7}\%$$

For the same amount of true error, the relative error is larger for smaller numbers because the denominator is smaller. Because we want to keep relative errors similar in magnitude across all numbers, we denote numbers using their *floating-point representation*. For any number y in base-10, its floating-point representation is given by

$$y = \sigma \times \sigma_e \times m \times 10^e \qquad (2.1)$$

where σ = sign of the number (- for negative)
σ_e = sign of the exponent (- for negative)
m = base/*mantissa* ($1 \leq m < 10$)
e = exponent/*ficand*

Example 6
Using the same numbers in Example 5,

120.78496 is now 1.207×10^2 or

1	2	0	7	2

in floating-point.

523410.00496 is now 5.234×10^5 or

5	2	3	4	5

in floating-point.

Hence,

$$RTE = \frac{120.78496 - 120.7}{120.78496} = 0.0703\%$$

$$RTE = \frac{523410.00496 - 523400}{523410.00496} = 0.001911\%$$

Thus, because of the floating-point representation, the errors are around the same order of magnitude for both small and large numbers.

For any number y in base-2, its floating-point representation is given by

$$y = \sigma \times \sigma_e \times m \times 2^e \qquad (2.2)$$

where σ = sign of the number (0 for positive, 1 for negative)
σ_e = sign of the exponent (- for negative)
m = base/*mantissa* ($(1)_2 \leq$ m $< (10)_2$)
e = exponent/*ficand*

Example 7

A machine stores floating-point numbers in 10 bits — the first bit used for the sign of the number, the second for the sign of the exponent, the next four for the exponent and the last four for the mantissa.

(a) How would you represent 524.0078125 in binary floating-point?
(b) Given the number

0	1	1	1	0	1	1	0	1	1

what number is it in base-10?

Answers

(a) We first convert 524.00078125 to binary using the algorithms discussed earlier.

i	OPERATION	=	QUOTIENT	REMAINDER
0	$524/2$	=	262	$0 = a_0$
1	$262/2$	=	131	$0 = a_1$
2	$131/2$	=	65	$1 = a_2$
3	$65/2$	=	32	$1 = a_3$
4	$32/2$	=	16	$0 = a_4$
5	$16/2$	=	8	$0 = a_5$
6	$8/2$	=	4	$0 = a_7$
7	$4/2$	=	2	$0 = a_8$
8	$2/2$	=	1	$0 = a_9$
9	$1/2$	=	0	$1 = a_{10}$
			Q = 0 END	

Hence,

$$(524)_{10} = (\mathbf{1000001100})_{\mathbf{2}}$$

i	OPERATION	=	FRACTIONAL	WHOLE
-1	0.0078125 x 2	=	0.015625	$0 = a_{-1}$
-2	0.015625 x 2	=	0.03125	$0 = a_{-2}$
-3	0.03125 x 2	=	0.0625	$0 = a_{-3}$
-4	0.0625 x 2	=	0.125	$0 = a_{-4}$
-5	0.125 x 2	=	0.25	$0 = a_{-5}$
-6	0.25 x 2	=	0.5	$0 = a_{-6}$
-7	0.5 x 2	=	0	$1 = a_{-7}$
			F = 0 END	

Hence,

$$(524.0078125)_{10} = (\mathbf{1000001100.0000001})_{\mathbf{2}}$$
$$= (\mathbf{1.0000011000000001})_{\mathbf{2}} \times \mathbf{2}^{(\mathbf{101})_{\mathbf{2}}}$$

Both the number and the exponent are positive. Thus, $\sigma = \sigma_e = 0$.

The mantissa $m = 1000$ and the exponent $e = 0101$.

Therefore, we end up with the floating-point representation

0	0	0	1	0	1	1	0	0	0

Often, the leading 1 in the mantissa is omitted, so that $m = 1000$. Thus, we end up with the following floating-point representation.

0	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**

Notice that if you convert this number to decimal, the result will be 512 and not 524.0078125. There is a true error of 12.0078125 — an example of a *floating-point error*. These types of errors will be discussed later on.

(b) From the given number,

a. $\sigma = 0$. Therefore, the number is positive.
b. $\sigma_e = 1$. Therefore, the exponent is negative
c. The mantissa $m = 11011$ and the exponent $e = 1101$

Thus, using the algorithms described earlier, the number is equal to

$$(1.1011)_2 \times 2^{(-1101)_2} = (1.1011)_2 \times 2^{-13}$$
$$= (0.00000000000011011)_2$$
$$\approx \mathbf{0.00020599365234375}$$

Actual computers often make use of single precision (16-bit) or double precision (32-bit) numbers, the allocation of which is governed by *IEEE-754* (IEEE stands for Institute of Electrical and Electronics Engineers.)

Floating point errors and the machine epsilon

We return to the previous example, where the binary floating-point representation of 524.0078125 does not exactly match up to the true value when converted back to decimal. In fact, when converted, the result will be 512 — a discrepancy of around 12.0078125. This is an example of a *rounding/chopping error*. This type of floating-point error occurs when the number of bits allocated to the mantissa/exponent is insufficient so that the machine is forced to chop or round off the number.

Underflow/overflow errors had also been previously mentioned in Chapter 1. These errors come about when the result of a calculation is a number which is smaller or larger than the machine can represent. For this, let us return to the previous example.

0	0	1	1	1	1	1	1	1	1

***Figure 2.7**. Example of a 10-bit binary floating-point number*

There are 2^{10} possible floating-point numbers which can be represented in a 10-bit machine (Note that each bit can hold up to 2 values — 0 and 1). The range of values that can be represented in such a system can be determined by setting both the mantissa and the exponent to their maximum possible values. When $m = 1111$ and $e = 1111$ as in Figure 2.7, then the maximum possible value that can be represented is given by

$$\begin{aligned} 1.1111 \times 2^{1111} &= 1.1111 \times 2^{15} \\ &= 1111100000000000 \\ &= \mathbf{63488} \end{aligned}$$

By the same argument, the minimum possible value is given by $\mathbf{-63488.}$

Thus, if a calculation or user input were to result in a number greater/less than ± 63488, this would result in an *overflow/underflow error*, respectively.

Because of these floating-point errors, there must be some means by which to determine the accuracy of a floating-point representation. Such a measure exists and is called the *machine epsilon* (ϵ_{mach}). The machine epsilon is calculated by finding the difference between 1 and the next number that can be represented. In the case of our 10-bit machine from earlier, 1 is represented as

0	0	0	0	0	0	0	0	0	0

While the next number that can be represented is

0	0	0	0	0	0	0	0	0	1

To find the machine epsilon,

$$\begin{aligned} \epsilon_{mach} &= (1.0001)_2 \times 2^{(0000)_2} - 1 \\ &= (1.0001)_2 \times 2^{(0000)_2} - 1 \\ &= (1.0001)_2 - 1 \end{aligned}$$

$$= 1.0625 - 1 = \mathbf{0.0625}$$

As a shortcut, ϵ_{mach} can also be calculated as 2 to the negative number of bits allocated to the mantissa. Alternatively,

$$\epsilon_{mach} = 2^{-4} = \mathbf{0.0625}$$

As a measure of accuracy, the machine epsilon often acts as an upper bound for the absolute value of the relative true error, as shown in the succeeding example.

Example 8

Using the numbers from Example 7(a),

$$|RTE| = \frac{12.0078125}{524.0078125} = \mathbf{0.02292}$$

which is less than the machine epsilon for a 10-bit machine that uses 4 bits for the mantissa, which is

$$\epsilon_{mach} = 2^{-4} = \mathbf{0.0625}$$

Application – Computational Physics as a Field

Physics majors are often used to the hustle and bustle of solving complicated problems analytically that many of them forget that many of today's problems are now being solved with the help of computers. Back then, many scientists relied on physicists and mathematicians to unravel seemingly intractable problems whose solution could hold the key to a variety of interesting applications. To make these intractable problems solvable, physicists of yesteryear had to employ many simplifying assumptions, leading to frictionless, linear and seemingly perfect models that would not make sense at all in the real world. Nevertheless, these perfect models – despite being poor reflections of the real world – helped build a solid body of knowledge from which we now draw to solve more daunting and complicated problems.

Ideally, the goal of physics is to explain real-world phenomena with mathematical models which can be exactly and elegantly solved. However, this is not always the case, as there are times when the equation is too convoluted to solve (or in some cases, the equation does not seem to be known at all!) This led to the development of numerical analysis as a field, and along with it emerged subfields like computational physics. Scientists back then had to develop ingenious methods to numerically or iteratively solve problems that could not be dealt with by hand. Indeed, the numerical methods and the approximations these scientists made persist today, but have ultimately been cast in the dark, what with the advent of computers and the emergence of several computational engines such as MATLAB, Mathematica, LabView and the like.

Still, however, the field of computational physics (and by extension, numerical analysis) remains vibrant as ever. With the development of computers, scientists have been able to translate their numerical methods into actual code, allowing the computer to automate the operation and allowing the scientists to run the operation with the push of a button. Similarly, visualizing physical phenomena through graphs or actual simulations has become easier. Before, a scientist would have had to go through lines and lines of tedious calculations to determine the effect of a small change in one of the variables to a certain model. Now, however, one can simply pull up MATLAB, Mathematica or even a spreadsheet program like Excel to find out what would happen.

Computational physics also continues to remain vibrant because of the growing interest in more complicated (and hence, more intractable) systems. Much of engineering is rooted in the fact that the physical system in question is linear. However, in truth, many natural phenomena are nonlinear in nature, and the math becomes much harder when we throw in nonlinearities. Solving a system of nonlinear equations or finding the solution to a simple nonlinear differential equation are things that are hardly ever taught in most math courses, and it is precisely because the nonlinearities make the process of solving much more complicated. With computational physics, we are able to explore nonlinear phenomena even further, allowing us to develop solutions for some engineering problems.

Moreover, there has also been interest in systems with many degrees of freedom. Many social and economic models, for example, have drawn from statistical mechanics, which itself involve systems with multiple degrees of freedom. Developments in computational physics allow us to refine our understanding of statistical mechanics and in turn, apply these models to other relevant fields such as sociology, economics and finance.

All in all, however, computational physics has stayed true to its roots and in spite of all the trappings, it is just another way of doing physics, often serving as an intermediary between laboratory experiments and theoretical calculations.

Gould, Tobochnik and Christian (2016) provide a great analogy between laboratory experiments and computer simulations. You start with a model, which you implement through a program. You test the program and once it works as intended, you make the necessary calculations, which serve as the foundation of your analysis. Lab work is no different, as you start with a sample and make use of physical apparatus to test your hypotheses. You calibrate the apparatus to make sure that they work as intended and after doing so, you make the necessary measurements which your analysis will revolve upon.

Laboratory Experiment	Computer Simulation
sample	model
physical apparatus	computer program
calibration	testing of program
measurement	computation
data analysis	data analysis

***Table 2.5**. Laboratory experiment vs computer simulations*

As developments in the realm of computer science continue to emerge, it is inevitable that computational physics may not only complement but also supersede traditional experimental physics. While we are on the edge of out seats waiting for these developments, we must take it upon ourselves to learn computational physics from the ground up.

CHAPTER 3: *SYSTEMS OF LINEAR EQUATIONS*

Introduction

Systems of linear equations are a mainstay in physics and are often used as a mathematical tool. From determining the currents in a closed circuit using Kirchhoff's laws to determining the eigenvectors associated with certain eigenvalues in quantum mechanics, linear equations have several known applications in the field of physics and as a student of computational physics, one must know how to solve these systems not only analytically but also numerically.

In this chapter, we will discuss the most primitive method of solving linear equations known as Gaussian elimination. We will first solve systems using an algorithm, but we will also be developing a formula which will come in handy once we try to implement the algorithm in Fortran.

Gaussian elimination

Given a general set of n equations in n unknowns,

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

we can usually represent the system as the matrix equation

$$\mathrm{Ax} = \mathrm{B}$$

where

$$\mathrm{A}_{n\times n} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

$$x_{n\times 1} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

and

$$B_{n\times 1} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Using linear algebra, the above system can be solved by finding the inverse of A and pre-multiplying it to B. Since the computation of the inverse is hard to implement numerically, however, we resort to a method known as Gaussian elimination. This method is often composed of two steps:

(1) *forward elimination*, where unknowns are systematically eliminated in each equation until only one unknown remains in the last equation, two unknowns remain in the second to the

last equation and so on. All unknowns must still be present in the first equation after this method is finished.

(2) *back substitution*, where starting from the last equation (which has only one unknown, as mentioned previously), the unknowns are solved for. The first unknown is solved for in the last equation, then substituted to the second to the last equation to look for the second unknown. This process continues until all unknowns have been solved for.

Forward elimination

This step starts by eliminating the first unknown x_1 from all rows except the first row. To do that, we divide the first row by a_{11} and multiply by $-a_{21}$, resulting in

$$-a_{21}x_1 - \frac{a_{21}}{a_{11}}a_{12}x_2 - \cdots - \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1$$

We then add the first equation to the second equation, eliminating x_1 and resulting in

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 - \left(a_{23} - \frac{a_{21}}{a_{11}}a_{13}\right)x_3 \ldots - \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n = b_2 - \frac{a_{21}}{a_{11}}b_1$$

We let

$$a_{22}^{(1)} = a_{22} - \frac{a_{21}}{a_{11}}a_{12}$$

$$\vdots$$

$$a_{2n}^{(1)} = a_{2n} - \frac{a_{21}}{a_{11}}a_{1n} \tag{3.1}$$

$$b_2^{(1)} = b_2 - \frac{a_{21}}{a_{11}}b_1 \tag{3.2}$$

so that the second equation becomes

$$a_{22}^{(1)}x_2 + \cdots + a_{2n}^{(1)}x_n = b_2^{(1)}$$

Repeating this for all $n-1$ equations, we get:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + \cdots + a_{3n}^{(1)}x_n &= b_3^{(1)} \\ &\vdots \\ a_{n2}^{(1)}x_2 + a_{n3}^{(1)}x_3 + \cdots + a_{nn}^{(1)}x_n &= b_n^{(1)} \end{aligned}$$

Now we wish to eliminate x_2 from all rows except the first and second. We repeat what we did earlier, dividing the second row by $a_{22}^{(1)}$ and multiply by $-a_{32}^{(1)}$, resulting in

$$-a_{32}^{(1)}x_2 - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}a_{23}^{(1)}x_3 \ldots - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}a_{2n}^{(1)}x_n = \frac{a_{32}^{(1)}}{a_{22}^{(1)}}b_2^{(1)}$$

We then add the second equation to the third equation, eliminating x_2 and resulting in

$$\left(a_{33}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}a_{23}^{(1)}\right)x_3 - \left(a_{34}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}a_{24}^{(1)}\right)x_4 - \cdots - \left(a_{3n}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}a_{23}^{(1)}\right)x_n = b_3^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}}b_2^{(1)}$$

We let

$$a_{33}^{(2)} = a_{33}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}} a_{23}^{(1)}$$

$$\vdots$$

$$a_{3n}^{(2)} = a_{3n}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}} a_{2n}^{(1)} \tag{3.3}$$

$$b_{3}^{(2)} = b_{3}^{(1)} - \frac{a_{32}^{(1)}}{a_{22}^{(1)}} b_{2}^{(1)} \tag{3.4}$$

so that the second equation becomes

$$a_{33}^{(2)} x_3 + \cdots + a_{3n}^{(2)} x_n = b_3^{(2)}$$

Repeating this for all $n-2$ equations, we get:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\ &\vdots \\ a_{n3}^{(2)}x_3 + \cdots + a_{nn}^{(2)}x_n &= b_n^{(2)} \end{aligned}$$

After $n-1$ steps of forward elimination, the system should look like the following:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\ &\vdots \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned} \tag{3.5}$$

To be able to implement forward elimination in Fortran (which will be discussed later), we would like to have a general formula for the coefficients which we can then run in a loop.

We can thus generalize (3.1) – (3.4) for any coefficient $a_{ij}^{(k)}$ and for any coefficient $b_i^{(k)}$.

Following (3.1) – (3.4),

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \left(\frac{a_{ik}}{a_{kk}} a_{kj} \right)^{(k-1)} \quad where \;\; 1 \le k \le n-1, k+1 \le i \le n, k \le j \le n \tag{3.6}$$

$$b_i^{(k)} = b_i^{(k-1)} - \left(\frac{a_{ik}}{a_{kk}} b_k \right)^{(k-1)} \quad where \;\; 1 \le k \le n-1, k+1 \le i \le n \tag{3.7}$$

Back substitution

After adding a few equations to (3.1) to make the discussion clearer, we have the following system:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\
&\vdots \\
a_{(n-2)(n-2)}^{(n-3)}x_{(n-2)} + a_{(n-2)(n-1)}^{(n-3)}x_{(n-1)} + a_{(n-2)n}^{(n-3)}x_n &= b_{(n-2)}^{(n-3)} \\
a_{(n-1)(n-1)}^{(n-2)}x_{(n-1)} + a_{(n-1)n}^{(n-2)}x_n &= b_{(n-1)}^{(n-2)} \\
a_{nn}^{(n-1)}x_n &= b_n^{(n-1)}
\end{aligned}$$

Notice that there is one unknown in the last equation. Hence, we can then solve for x_n.

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

We substitute this to the second to the last equation, then solve for $x_{(n-1)}$, yielding

$$x_{(n-1)} = \frac{b_{(n-1)}^{(n-2)} - a_{(n-1)n}^{(n-2)}x_n}{a_{(n-1)(n-1)}^{(n-2)}} \tag{3.8}$$

Repeating this process for the third to the last equation, we substitute, then solve for $x_{(n-2)}$, resulting in

$$x_{(n-2)} = \frac{b_{(n-2)}^{(n-3)} - a_{(n-2)n}^{(n-3)}x_n - a_{(n-2)(n-1)}^{(n-3)}x_{(n-1)}}{a_{(n-2)(n-2)}^{(n-3)}} \tag{3.9}$$

We can generalize this for any coefficient x_i . Following (3.8) and (3.9),

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)}x_j}{a_{ii}^{(i-1)}} \tag{3.10}$$

Gaussian elimination – example

NOTE: This problem is adapted from *Matrix Analysis and Applied Linear Algebra* by Meyer.

Suppose that 100 insects are distributed in an enclosure consisting of four chambers with passageways between them as shown in Figure 3.1.

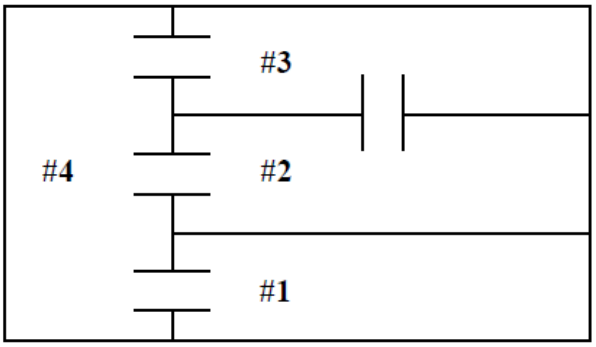

Figure 3.1. *Accompanying figure*

Adapted from Meyer (2000)

At the end of one minute, the insects have redistributed themselves. Assume that a minute is not enough time for an insect to visit more than one chamber and that at the end of a minute 40% of the insects in each chamber have not left the chamber they occupied at the beginning of the minute. The insects that leave a chamber disperse uniformly among the chambers that are directly accessible from the one they initially occupied—e.g., from #3, half move to #2 and half move to #4.

If at the end of one minute there are 12, 25, 26, and 37 insects in chambers #1, #2, #3, and #4, respectively, determine what the initial distribution had to be.

Given the specifics of the problem, the system can be represented as follows:

$$\begin{aligned} 0.4x_1 + 0x_2 + 0x_3 + 0.2x_4 &= 12 \\ 0x_1 + 0.4x_2 + 0.3x_3 + 0.2x_4 &= 25 \\ 0x_1 + 0.3x_2 + 0.4x_3 + 0.2x_4 &= 26 \\ 0.6x_1 + 0.3x_2 + 0.3x_3 + 0.4x_4 &= 37 \end{aligned} \tag{3.11}$$

To understand how the linear equations in (3.11) came about, let us examine the fourth equation in the system. The fourth equation represents the number of insects in chamber #4. An insect can go to chamber #4 coming from chambers #1, #2 and #3. Looking at chamber #2, for example, 40% of the insects stay so 60% of them leave. Since the insects disperse uniformly, half of the leaving insects (or 30%) go to chamber #3 and the remaining 30% go to chamber #4. Thus, 0.3 of the initial number of insects in chamber #2 (x_2) contribute to the 37 insects in chamber #4 after one minute. The same logic applies to all the other coefficients, as well as to all other equations in the system.

Forward elimination

First step

Using Gaussian elimination, we start by eliminating x_1 from all equations except the first. Dividing the first equation by 0.4, multiplying it by -0.6 and adding it to the fourth equation, we get

$$\begin{aligned} -0.6x_1 + 0x_2 + 0x_3 - 0.3x_4 &= -18 \\ \underline{0.6x_1 + 0.3x_2 + 0.3x_3 + 0.4x_4 = 37} \end{aligned}$$

$$0x_1 + 0.3x_2 + 0.3x_3 + 0.1x_4 = 19$$

so the system becomes

$$\begin{aligned} 0.4x_1 + 0x_2 + 0x_3 + 0.2x_4 &= 12 \\ 0x_1 + 0.4x_2 + 0.3x_3 + 0.2x_4 &= 25 \\ 0x_1 + 0.3x_2 + 0.4x_3 + 0.2x_4 &= 26 \\ 0x_1 + 0.3x_2 + 0.3x_3 + 0.1x_4 &= 19 \end{aligned}$$

Second step

We now eliminate x_2 in all equations except the first and second. Dividing the second equation by 0.4, multiplying it by -0.3 and adding it to the third and fourth equation, we get

$$\begin{aligned} 0x_1 - 0.3x_2 - 0.225x_3 - 0.15x_4 &= -18.75 \\ \underline{0x_1 + 0.3x_2 + 0.4x_3 + 0.2x_4 = 26} \\ 0x_1 + 0x_2 + 0.175x_3 + 0.05x_4 &= 7.25 \end{aligned}$$

$$\begin{aligned} 0x_1 - 0.3x_2 - 0.225x_3 - 0.15x_4 &= -18.75 \\ \underline{0x_1 + 0.3x_2 + 0.3x_3 + 0.1x_4 = 19} \\ 0x_1 + 0x_2 + 0.075x_3 - 0.05x_4 &= 0.25 \end{aligned}$$

so the system becomes

$$\begin{aligned} 0.4x_1 + 0x_2 + 0x_3 + 0.2x_4 &= 12 \\ 0x_1 + 0.4x_2 + 0.3x_3 + 0.2x_4 &= 25 \\ 0x_1 + 0x_2 + 0.175x_3 + 0.05x_4 &= 7.25 \\ 0x_1 + 0x_2 + 0.075x_3 - 0.05x_4 &= 0.25 \end{aligned}$$

Third step

We now eliminate x_3 in all equations except the first, second and third. In this case, however, by inspection, we see that it is easier to eliminate x_4 in the fourth equation, so we do this instead. Adding the third equation to the fourth equation, we get

$$\begin{aligned} 0x_1 + 0x_2 + 0.175x_3 + 0.05x_4 &= 7.25 \\ \underline{0x_1 + 0x_2 + 0.075x_3 - 0.05x_4 = 0.25} \\ 0x_1 + 0x_2 + 0.25x_3 + 0x_4 &= 7.5 \end{aligned}$$

so the system becomes

$$\begin{aligned} 0.4x_1 + 0x_2 + 0x_3 + 0.2x_4 &= 12 \\ 0x_1 + 0.4x_2 + 0.3x_3 + 0.2x_4 &= 25 \\ 0x_1 + 0x_2 + 0.175x_3 + 0.05x_4 &= 7.25 \\ 0x_1 + 0x_2 + 0.25x_3 + 0x_4 &= 7.5 \end{aligned}$$

Back substitution

From the fourth equation,

$$0.25x_3 = 7.5$$
$$x_3 = \frac{7.5}{0.25} = \mathbf{30}$$

Substituting x_3 to the third equation,

$$0.175(30) + 0.05x_4 = 7.25$$
$$0.05x_4 = 2$$
$$x_4 = \frac{2}{0.05} = \mathbf{40}$$

Substituting x_3 and x_4 to the second equation,

$$0.4x_2 + 0.3(30) + 0.2(40) = 25$$
$$0.4x_2 = 8$$
$$x_2 = \frac{8}{0.4} = \mathbf{20}$$

Substituting x_2, x_3 and x_4 to the first equation,

$$0.4x_1 + 0(20) + 0(30) + 0.2(40) = 12$$
$$0.4x_2 = 4$$
$$x_1 = \frac{4}{0.4} = \mathbf{10}$$

Hence, there were initially **10** insects in chamber #1, **20** insects in chamber #2, **30** insects in chamber #3 and **40** insects in chamber #4.

Gaussian elimination (Fortran example)

To implement Gaussian elimination in Fortran, we first need to see the algorithm in a flowchart. The algorithm is summarized in Figures 3.2 and 3.3

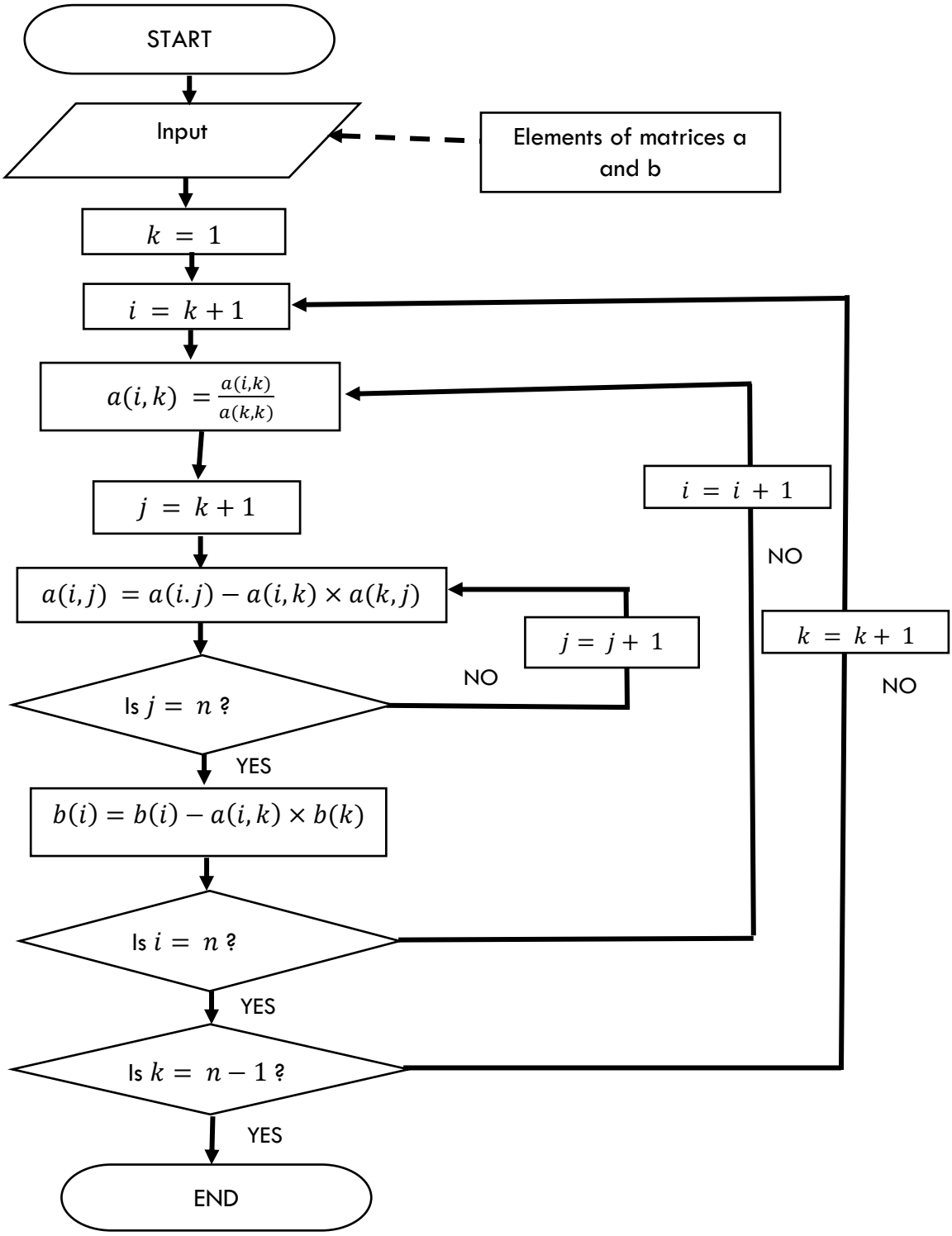

Figure 3.2. *Flowchart for forward elimination*

Algorithm adapted with changes from Cheney & Kincaid (2008)

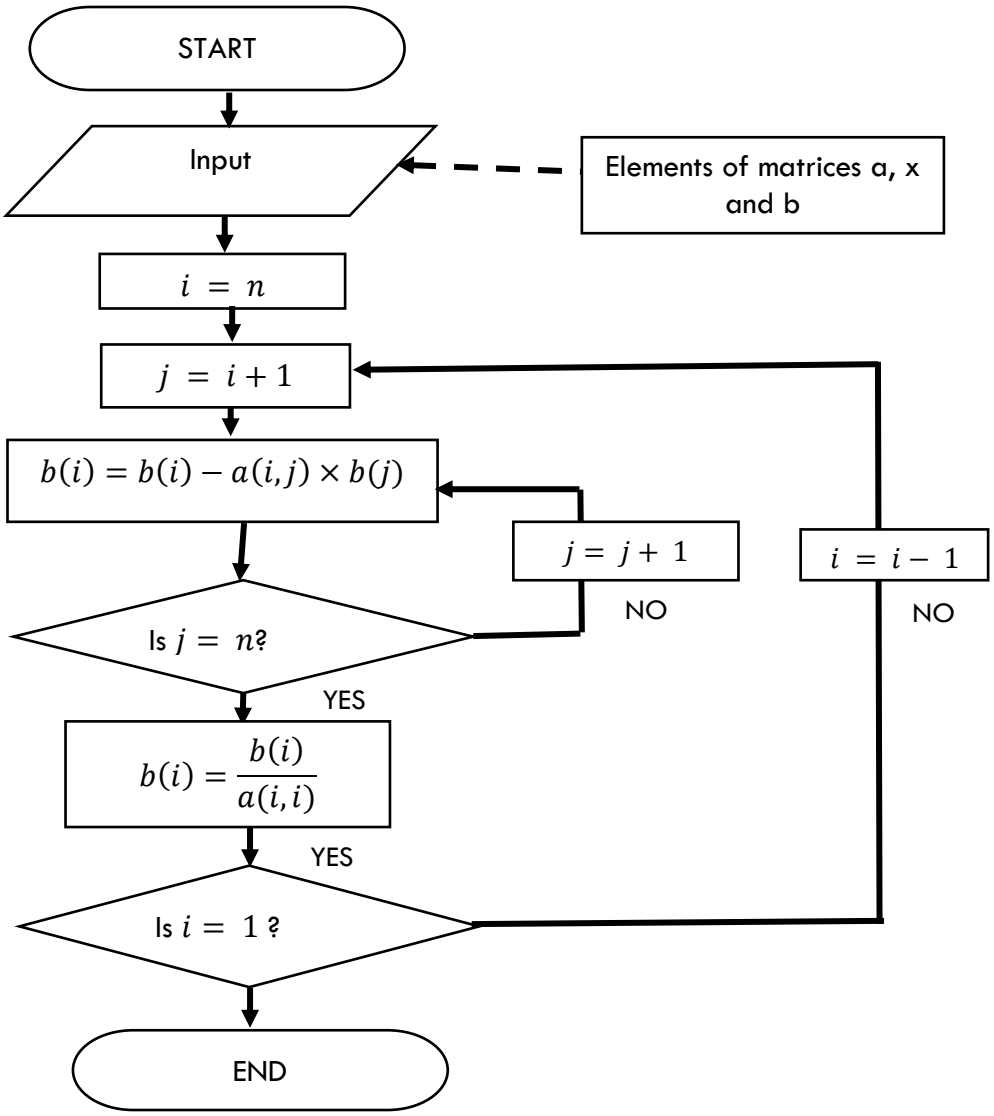

Figure 3.3. *Flowchart for back substitution*

Algorithm adapted with changes from Cheney & Kincaid (2008)

Notice that the algorithms in the flowcharts vary slightly from the algorithms used earlier in solving systems of linear equations by hand. We will discuss the differences and why they came into place as we go through the code.

```
!=============================================================================================
! Code_4_naive_gaussian.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program solves a system of linear equations using the naive Gaussian method.
!=============================================================================================

program naive_gaussian

implicit none
	!-----------------------------------------------------------------------------------------
	! Declare variables
	integer :: n, i, j, k
	real*16, dimension(:, :), allocatable :: a
	real*16, dimension(:), allocatable :: b
	character :: confirmed
	!-----------------------------------------------------------------------------------------

	!-----------------------------------------------------------------------------------------
	! Ask user to specify n of matrix
	write(*,"(a47)", advance = "no") "How many unknowns will there be in the system? "
	read(*,*) n
	!-----------------------------------------------------------------------------------------

	!-----------------------------------------------------------------------------------------
	! Allocate n
	allocate(a(n, n))
	allocate(b(n))
	!-----------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`n` stores the number of unknowns while `i`, `j` and `k` are counters. The dimensions of arrays `a` and `b` are made allocatable so that the user can specify them. The solutions are also stored in `b`. `confirmed` is use to track whether or not the user specified the values of the matrices to his/her taste.

The user is then prompted to specify the number of unknowns, after which the program allocates the dimensions of arrays `a` and `b` based on the user's specification.

```
!-----------------------------------------------------------------------------------------
! Ask user to specify values
do while(confirmed /= "Y")
	write(*,*)
	write(*,"(a33, 1x, i2, 1x, a1, i2, 1x, a7)") "Please specify the values of your", n,
```

```
        "x", n, "matrix."
        write(*,*) "Please specify the values from left to right then top to bottom"
        write(*,*) "Press Enter once you've specified a value."
        do i = 1, n
                do j = 1, n
                        read(*,*) a(i,j)
                end do
        end do

        write(*,*) "The matrix you created is:"
        do i = 1, n
                do j = 1, n
                        write(*,"(f10.5, 3x)", advance = "no") a(i,j)
                end do
        write(*,*)
        end do

        write(*,*)
        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
        write(*,*)
        write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
        write(*,*) "These are the constants in your system of linear equations."

        do i = 1, n
                read(*,*) b(i)
        end do

        write(*,*) "The matrix you created is:"
        do i = 1, n
                write(*,"(f10.5)") b(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do
!-------------------------------------------------------------------------------------
```

The user is then prompted to specify the values for each matrix. After specifying the values, the user is shown the values in matrix form and is asked to confirm if the values had been entered correctly. As long as the user presses "N" (or any other character other than "Y"), the program will continue to prompt the user to enter the values of the matrix. Asking the user to confirm his choice of values ensures that the user entered his/her values correctly and in line with the instructions (left to right and top to bottom, "Enter" after every entry).

```
! Implement Naive Gaussian algorithm
!-----------------------------------------------------------------------------------------------
! Forward elimination
do k = 1, n-1
       do i = k + 1, n
              a(i, k) = a(i, k) / a(k, k)

       do j = k + 1, n
              a(i, j) = a(i, j) - a(i, k) * a(k, j)
       end do

       b(i) = b(i) - a(i, k) * b(k)
        end do
end do
!-----------------------------------------------------------------------------------------------
```

In this code snippet, the program implements forward elimination, following the algorithm outlined in Figure 3.2 to the letter. However, this algorithm has some slight differences with the algorithm discussed earlier.

For the first step, we run through rows 1 to n – 1 and columns k + 1 to n as in (3.6). These comprise the lower triangular part of the matrix, which will eventually be zeroed out. We apply (3.6) and divide all the elements of the lower triangular part of the matrix by the pivot element/diagonal element `a(k, k)`.

After this, we run through columns k + 1 to n. We can choose to run from k to n but if we let `j = k` as in (3.6), we expect the value to be zero (Let `j = k` in (3.6) and see for yourself). Thus, there is no point in computing it and we proceed with the $(k+1)^{th}$ row instead. Once we finish with the forward elimination of the matrix `a`, we move on to matrix `b`. We simply apply (3.7), running through rows k + 1 to n.

```
!-----------------------------------------------------------------------------------------------
! Back substitution
do i = n, 1, -1
       do j = i + 1, n
              b(i) = b(i) - a(i, j) * b(j)
       end do

       b(i) = b(i) / a(i, i)
end do
!-----------------------------------------------------------------------------------------------
!-----------------------------------------------------------------------------------------------
! Print out result
write(*,*)
write(*,*) "The solution to the system you specified is: "
do i = 1, n
       write(*,"(f10.5)") b(i)
end do
!-----------------------------------------------------------------------------------------------
```

The preceding code snippet now implements back substitution. We run through rows n to 1 in reverse order and columns i + 1 to n in the matrix a. We apply (3.10) but instead of x_j, we store the solutions in the matrix b to save some space. We can do this because the solutions are stored in reverse, from b(n) to b(1).

Taking the first iteration as an example,

$$\begin{aligned} b(n) &= b(n) - a(n, n+1) * b(n+1) \\ &= b(n) \end{aligned}$$

which is divided by a(n, n) afterwards, giving us the first solution.

For the second iteration,

$$b(n-1) = b(n-1) - a(n-1, n) * b(n)$$

which is then divided by a(n-1, n-1), as in (3.8). Thus, we can safely store our solutions in the matrix b. The full program is as follows:

```
!=============================================================================================
! Code_4_naive_gaussian.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program solves a system of linear equations using the naive Gaussian method.
!=============================================================================================

program naive_gaussian

implicit none
       !-------------------------------------------------------------------------------------
       ! Declare variables
       integer :: n, i, j, k
       real*16, dimension(:, :), allocatable :: a
       real*16, dimension(:), allocatable :: b
       character :: confirmed
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! Ask user to specify n of matrix
       write(*,"(a47)", advance = "no") "How many unknowns will there be in the system? "
       read(*,*) n
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! Allocate n
       allocate(a(n, n))
       allocate(b(n))
       !-------------------------------------------------------------------------------------
```

```
        "x", n, "matrix."
        write(*,*) "Please specify the values from left to right then top to bottom"
        write(*,*) "Press Enter once you've specified a value."
        do i = 1, n
               do j = 1, n
                      read(*,*) a(i,j)
               end do
        end do

        write(*,*) "The matrix you created is:"
        do i = 1, n
               do j = 1, n
                      write(*,"(f10.5, 3x)", advance = "no") a(i,j)
               end do
        write(*,*)
        end do

        write(*,*)
        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
        write(*,*)
        write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
        write(*,*) "These are the constants in your system of linear equations."

        do i = 1, n
               read(*,*) b(i)
        end do

        write(*,*) "The matrix you created is:"
        do i = 1, n
               write(*,"(f10.5)") b(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do
!-------------------------------------------------------------------------------

! Implement Naive Gaussian algorithm

!-------------------------------------------------------------------------------
! Forward elimination
```

```
        do k = 1, n-1
                do i = k+1, n
                        a(i, k) = a(i, k) / a(k, k)

                do j = k+1, n
                        a(i, j) = a(i, j) - a(i, k) * a(k, j)
                end do

                b(i) = b(i) - a(i, k) * b(k)
                 end do
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! Back substitution
        do i = n, 1, -1
                do j = i + 1, n
                        b(i) = b(i) - a(i, j) * b(j)
                end do

                b(i) = b(i) / a(i, i)
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! Print out result
        write(*,*)
        write(*,*) "The solution to the system you specified is: "
        do i = 1, n
                write(*,"(f10.5)") b(i)
        end do
        !-------------------------------------------------------------------------------

end program naive_gaussian
```

Executing the program,

```
How many unknowns will there be in the system? 4

Please specify the values of your  4 x 4 matrix.
 These are the coefficients in your system of linear equations.
 Please specify the values from left to right then top to bottom
 Press Enter once you've specified a value.
```

Figure 3.4. *User specifying the number of unknowns*

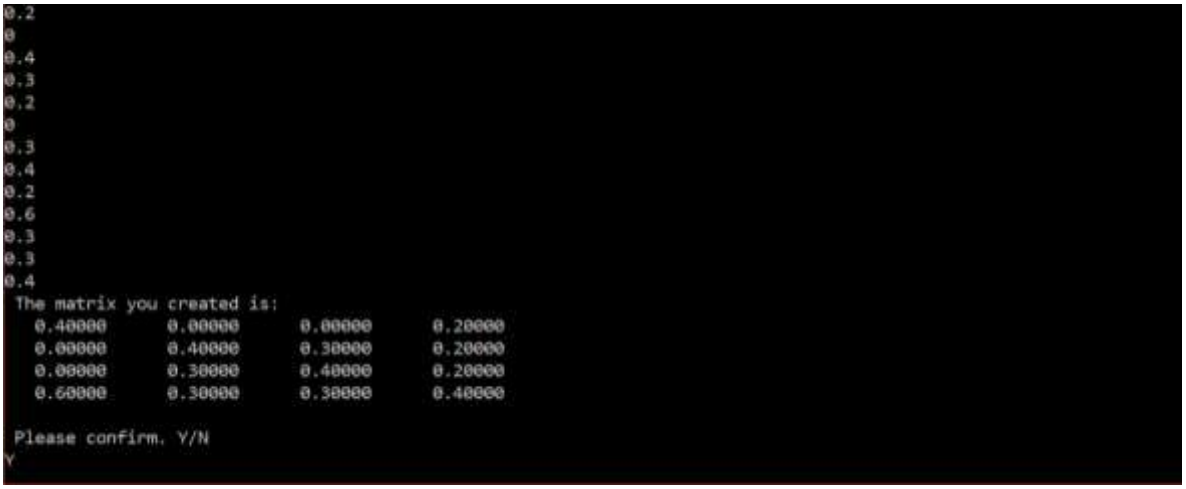

Figure 3.5. *User specifying the matrix of coefficients*

Notice how the values were entered. The user should press "Enter" after entering each value.

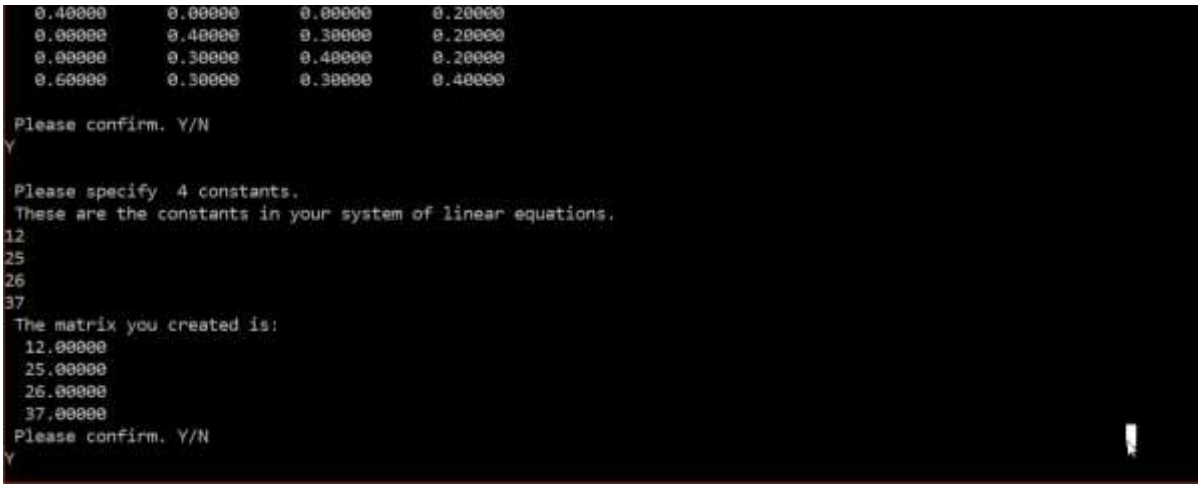

Figure 3.6. *User specifying the matrix of constants*

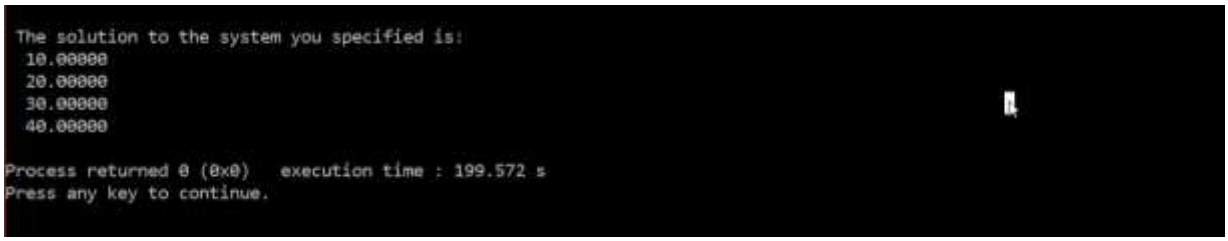

Figure 3.7. *Program output*

we find that the solution to (3.11) is given by

$$x_1 = \mathbf{10}, x_2 = \mathbf{20}, x_3 = \mathbf{30} \text{ and } x_4 = \mathbf{40}$$

Application – Kirchhoff's laws

Systems of linear equations are widely used in physics. One such application is in solving for the unknown quantities in an electric circuit using Kirchhoff's laws.

Kirchhoff's current rule (KCL) comes as a consequence of the Law of Conservation of Electrical Charge and tells us that the sum of the currents entering a node is equal to the sum of the currents leaving a node. In other words,

$$\sum I_{in} = \sum I_{out} \tag{3.12}$$

On the other hand, *Kirchhoff's voltage rule* (KVL) is a consequence of the electrostatic field being a conservative field. As such, the work done by an electrostatic field on a charge that moves around a closed loop is zero. Thus, the sum of all potential differences/voltages around a closed loop must be zero. Expressed mathematically,

$$\sum V = 0 \text{ (around a closed loop)} \tag{3.13}$$

Using Ohm's Law, (3.13) becomes

$$\sum \varepsilon + IR = 0 \text{ (around a closed loop)} \tag{3.14}$$

The number of KCL and KVL equations in our system depends on the number of nodes and loops. Conventionally there are $(\#\ of\ nodes - 1)$ KCL equations and $(\#\ of\ inner\ loops)$ KVL equations.

To demonstrate how we can apply the concepts in this chapter to solving the unknown quantities in an electrical circuit using Kirchhoff's laws, we use an example from Alexander & Sadiku (2009).

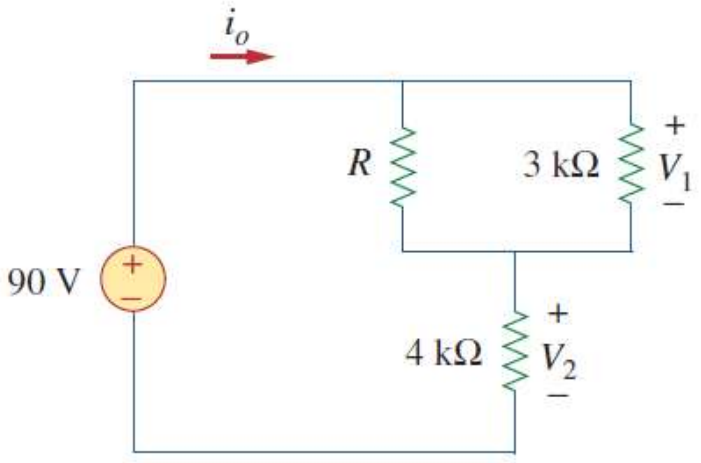

Figure 3.8. *An electrical circuit*

In Figure 3.8, we are asked to determine i_0, V_1 and V_2, knowing that $R = 6000\ \Omega$.

From earlier, we know that there are two nodes in the circuit, so there should be $2 - 1 = 1$ KCL equation. Furthermore, there are two loops, so there should be 2 KVL equations.

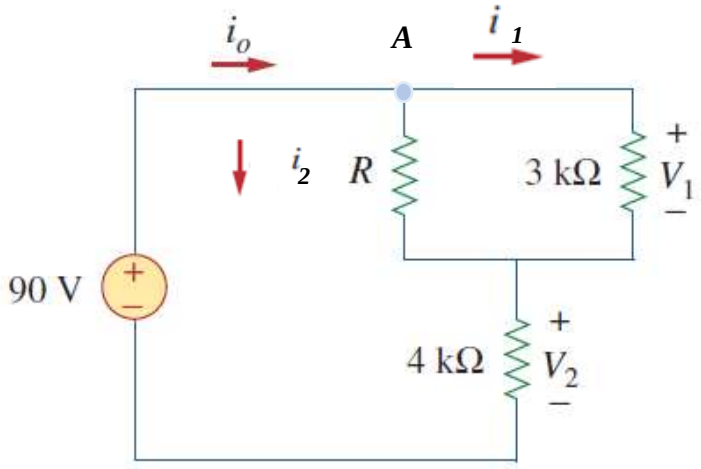

Figure 3.9. *An electrical circuit (labeled)*

Starting from node A (marked in Figure 3.9), we use (3.12) and get:

$$i_0 = i_1 + i_2 \tag{3.15}$$

Starting again from node A, we apply (3.14) and get:

$$i_1(3000\ \Omega) - i_2(6000\ \Omega) = 0 \tag{3.16}$$

Now we start from the voltage source and go around the loop, yielding:

$$90V - i_0(4000\Omega) - i_2(6000\ \Omega) = 0 \tag{3.17}$$

Thus, neglecting the units momentarily, we have the following system of linear equations from (3.15), (3.16) and (3.17):

$$\begin{aligned} i_0 - i_1 - i_2 &= 0 \\ 3000i_1 - 6000i_2 &= 0 \\ 4000i_0 + 6000i_2 &= 90 \end{aligned} \tag{3.18}$$

We can now use `Code_4_naive_gaussian.f90` to solve this system

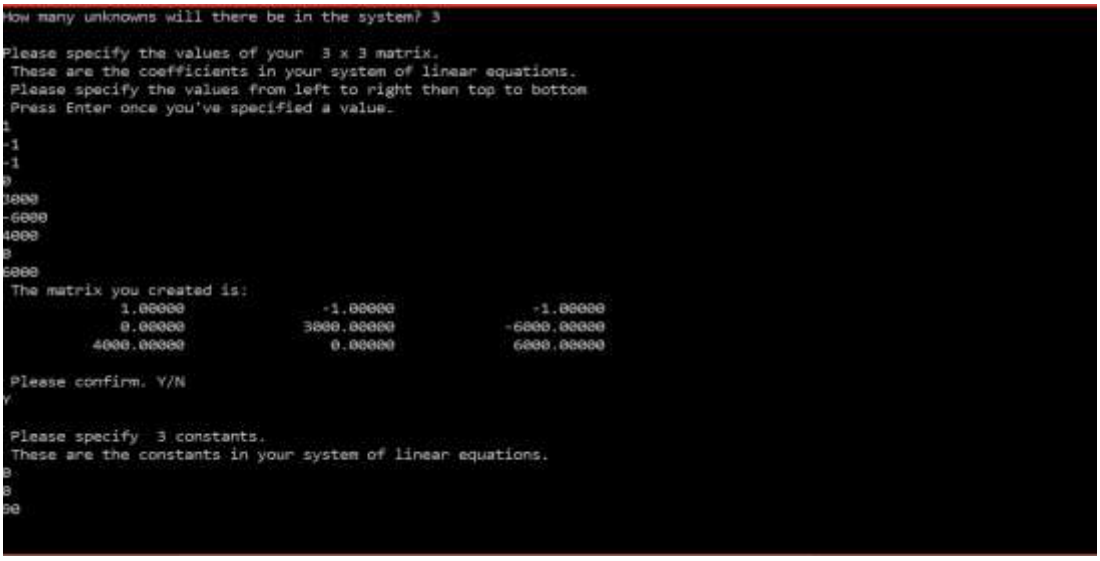

Figure 3.10. *User specifying the system of linear equations*

In doing so, we get

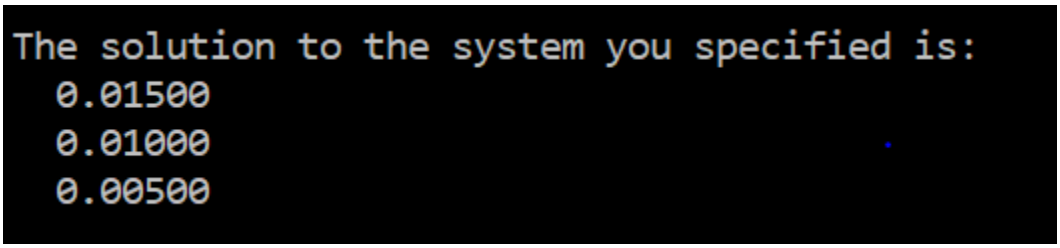

Figure 3.11. *Solutions to the system*

$i_0 = \mathbf{0.015\ A}$

$i_1 = 0.01\ A$

$i_2 = 0.005\ A$

Thus,

$V_1 = i_1(3000\ \Omega) = \mathbf{30\ V}$

$V_2 = i_0(4000\ \Omega) = \mathbf{60\ V}$

CHAPTER 4: *INTERPOLATING FUNCTIONS*

Introduction

In physics, values normally occur over a continuum. However, we can only obtain a finite number of data points experimentally. How then are we to relate two physical variables generally using only a limited set of data?

Given a set of points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, one can determine the function $f(x)$ passing through all $n + 1$ points using a process known as *interpolation*. Normally, $f(x)$ is often made to be a polynomial, as polynomials are easy to evaluate, differentiate and integrate.

In this chapter, we will discuss two simple methods of interpolating a function: (1) polynomial interpolation (otherwise known as the direct method) and (2) Lagrange interpolation.

Polynomial interpolation

Given a general set of $n + 1$ points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, the goal of polynomial interpolation (also known as the *direct method* or the *method of undetermined coefficients*) is to find a polynomial of degree n of the form

$$y = a_0 + a_1 x + a_2 x^2 \dots + a_n x^n \tag{4.1}$$

which will fit all the given values. A polynomial of degree less than n can be used, but will yield less accurate approximations.

Substituting all $n + 1$ points to (4.1) yields $n + 1$ equations, giving us

$$\begin{aligned} y_0 &= a_0 + a_1 x_0 + a_2 {x_0}^2 \dots + a_n {x_0}^n \\ y_1 &= a_0 + a_1 x_1 + a_2 {x_1}^2 \dots + a_n {x_1}^n \\ &\vdots \\ y_n &= a_0 + a_1 x_n + a_2 {x_n}^2 \dots + a_n {x_n}^n \end{aligned} \tag{4.2}$$

which we can then solve using the Gauss elimination method, discussed in the previous chapter.

Example 1

This example is adapted from Kaw (2011).

The velocity of a rocket is a function of time, with the values shown in Table 1.

t (s)	**v(t) (m/s)**
10	227.04
15	362.78
20	517.35
22.5	602.97

Table 4.3. *Velocity as a function of time*

a) Find the third-order polynomial corresponding to these values.
b) Determine the value of the velocity at t = 16 using the direct method.

c) Find the absolute relative approximation error for the third-order polynomial.

Solutions

a) From (4.1), the interpolating polynomial will be of the form

$$v(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{4.3}$$

Substituting all 4 points to (4.3) yields

$$\begin{aligned} a_0 + a_1(10) + a_2(10)^2 + a_3(10)^3 &= 227.04 \\ a_0 + a_1(15) + a_2(15)^2 + a_3(15)^3 &= 362.78 \\ a_0 + a_1(20) + a_2(20)^2 + a_3(20)^3 &= 517.35 \\ a_0 + a_1(22.5) + a_2(22.5)^2 + a_3(22.5)^3 &= 602.97 \end{aligned} \tag{4.4}$$

Writing (4.4) in matrix form, we get

$$\begin{pmatrix} 1 & 10 & 100 & 1000 \\ 1 & 15 & 225 & 3375 \\ 1 & 20 & 400 & 8000 \\ 1 & 22.5 & 506.25 & 11391 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 362.78 \\ 517.35 \\ 602.97 \end{pmatrix} \tag{4.5}$$

In setting up the coefficient matrix, notice that the first column will always be a column of 1's. The second column contains the values of the independent variable x while the third and fourth columns contain the square and the cube of these values, respectively.

To solve (4.5), we must perform Gaussian elimination on the coefficient and constant matrices. We first apply forward elimination. We first multiply the first row by -1 and add it to the remaining rows. This yields

$$\begin{pmatrix} 1 & 10 & 100 & 1000 \\ 0 & 5 & 125 & 2375 \\ 0 & 10 & 300 & 7000 \\ 0 & 12.5 & 406.25 & 10391 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 135.74 \\ 290.31 \\ 375.93 \end{pmatrix}$$

We multiply the second row by -2 and add it to the second remaining rows. We also multiply the second row by -2.5 and add it to the fourth row. Doing both operations, you will get

$$\begin{pmatrix} 1 & 10 & 100 & 1000 \\ 0 & 5 & 125 & 2375 \\ 0 & 0 & 50 & 2250 \\ 0 & 0 & 93.75 & 4453.50 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 135.74 \\ 18.83 \\ 36.58 \end{pmatrix}$$

Continuing the process, we multiply the third row by -1.875 and add it to the fourth row, resulting in

$$\begin{pmatrix} 1 & 10 & 100 & 1000 \\ 0 & 5 & 125 & 2375 \\ 0 & 0 & 50 & 2250 \\ 0 & 0 & 0 & 234.75 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 135.74 \\ 18.83 \\ 1.27375 \end{pmatrix}$$

We now proceed with back substitution. In doing so, we find that

$$a_0 = -4.2540$$
$$a_1 = 21.2166$$
$$a_2 = 0.1324$$
$$a_3 = 0.0054347$$

Thus, substituting these in (4.3) gives the interpolating polynomial, which is

$$v(t) = \mathbf{-4.2540 + 21.2166t + 0.1324t^2 + 0.0054347t^3} \tag{4.6}$$

b) Given (4.6), we can simply substitute $t = 16$ to $v(t)$ to get the velocity. In doing so, we get

$$\begin{aligned} v(16) &= -4.2540 + 21.2166(16) + 0.1324(16)^2 + 0.0054347(16)^3 \\ &= \mathbf{392.06\ m/s} \end{aligned}$$

c) We need to find the relative error with respect to the second-order polynomial approximation. The second-order polynomial approximation will vary based on the choice of data points to include. For our example, we include the first three data points in our approximation. Repeating what we did in the first part of (a), we should get a system of the following form

$$\begin{pmatrix} 1 & 10 & 100 \\ 1 & 15 & 225 \\ 1 & 20 & 400 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 362.78 \\ 517.35 \end{pmatrix} \tag{4.7}$$

Solving (4.7) using Gaussian elimination will yield the following interpolating polynomial

$$v(t) = \mathbf{12.05 + 3.733t + 0.3766t^2} \tag{4.8}$$

Substituting $t = 16$ to (4.8) then gives us

$$\begin{aligned} v(16) &= 12.05 + 3.733(16) + 0.3766(16)^2 \\ &= \mathbf{392.19\ m/s} \end{aligned} \tag{4.9}$$

Thus, we can now compute for the relative approximate error associated with the third-order polynomial approximation, given by

$$|RAE| = \left|\frac{392.06 - 392.19}{392.06}\right| \times 100 = \mathbf{0.033269\%}$$

For comparison, we can compute for the velocity using a first-order polynomial approximation and compare how the relative approximate error changes as we add more and more terms to the approximation. Using $t = 15$ and $t = 20$ as our two data points, we get an interpolating polynomial of

$$v(t) = \mathbf{-100.93 + 30.914t}$$

This leads to an estimate to $v(16)$ equal to

$$v(t) = -100.93 + 30.914(16)$$

$$= \mathbf{393.7}\ \boldsymbol{m/s}$$

Computing for the relative approximate error associated with the second-order polynomial approximation,

$$|RAE| = \left|\frac{392.19 - 393.7}{392.19}\right| \times 100 = \mathbf{0.38410}\%$$

Thus, we can see that the relative approximate error actually decreases as we add more and more terms. This supports (4.1), which states that for n points, you will need n equations to get the most accurate polynomial approximation possible. Therefore, in the succeeding programs/examples, we will always strive to obtain the most accurate polynomial approximation possible.

Polynomial approximation – Fortran example

Implementing polynomial approximation in Fortran is simply a matter of re-using the Gaussian elimination code previously discussed in Chapter 3.

```
!=============================================================================================
! Code_5_polynomial.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program interpolates the polynomial for a given set of points using the direct
! method.
!=============================================================================================

program polynomial

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	integer :: n, i, j
	real*16 :: point, total = 0.0
	real*16, dimension(:,:), allocatable :: a
	real*16, dimension(:), allocatable :: x, y, coeff
	character :: confirmed
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Ask user to specify the number of data points
	write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
	read(*,*) n
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Allocate n
	allocate(x(n))
	allocate(y(n))
	allocate(coeff(n))
	allocate(a(n,n))
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`n` stores the number of unknowns while `i` and `j` are counters. The dimensions of arrays `a`, `x`, `y` and `coeff` are made allocatable so that the user can specify them. `x` and `y` represent the arrays for the independent and dependent variable, respectively. `a` represents the coefficient matrix, while `y` represent the constant matrix. `coeff` stores the results. `confirmed` is use to track whether or not the user specified the values of the matrices to his/her taste. The user is then prompted to specify the

number of data points to be used, after which the program allocates the dimensions of the arrays based on the user's specification.

```
!-------------------------------------------------------------------------------------------
! Ask user to specify values
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your INDEPENDENT variables."

       do i = 1, n
              read(*,*) x(i)
       end do

       write(*,*) "Your list of independent variable values is:"
       do i = 1, n
              write(*,"(f10.5)") x(i)
       end do

       write(*,*) "Please confirm. Y/N"
       read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your DEPENDENT variables."

       do i = 1, n
              read(*,*) y(i)
       end do

       write(*,*) "Your list of dependent variable values is:"
       write(*,"(a10, 2x, a10)") "x", "y"
       do i = 1, n
              write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
       end do

       write(*,*) "Please confirm. Y/N"
       read(*,*) confirmed
end do

write(*,"(a44, 1x)", advance = "no") "At what point would you like to interpolate?"
read(*,*) point
!-------------------------------------------------------------------------------------------
```

The user is then prompted to specify the data points. The user is asked to confirm his selection so that he/she may see whether the values had been inputted correctly to the program. The user is also asked to specify the point to be evaluated using the interpolating polynomial.

```
!-----------------------------------------------------------------------------------------
! Fill matrix a
do i = 1, n
       do j = 1, n
              a(i, 1) = 1
              a(i, j) = x(i) ** (j - 1)
        end do
end do
!-----------------------------------------------------------------------------------------
```

As stated earlier, for the coefficient matrix `a`, the first column is simply a column of 1's. The second column contains the values of the independent variable x while the succeeding columns contain the $(i - 1)^{th}$ power of these values. Thus, we fill up the matrix `a` in accordance with these observations.

```
coeff = gaussian(n, a, y)

!-----------------------------------------------------------------------------------------
! Obtain polynomial and the value of point at polynomial.
write(*,*)
do i = 1, n
       total = total + coeff(i) * point ** (i - 1)
end do
write(*,"(a67, 1x, f10.5)") "The value of your chosen point at the interpolating polynomial
is: ", total
!-----------------------------------------------------------------------------------------
```

Having set up all the necessary matrices, we perform Gaussian elimination, which we execute via the function `gaussian`. This function has three parameters: `n`, the number of equations; `a`, the coefficient matrix and `y`, the constant matrix. After performing Gaussian elimination, we can now compute for the interpolating polynomial by substituting the coefficients to (4.1). We can then substitute the point of interest specified earlier by the user to the interpolating polynomial, yielding the code snippet above.

```
contains
!=========================================================================================
! This function implements the Naive Gaussian elimination algorithm.
!=========================================================================================
       function gaussian(n, a, b) result(x)
       !-----------------------------------------------------------------------------------
       ! Declare variables
       integer :: n, i, j, k
       real*16 :: total
       real*16, dimension(n, n) :: a
       real*16, dimension(n) :: b, x
```

```
        ! Implement Naive Gaussian algorithm

        !-------------------------------------------------------------------------------
        ! Forward elimination
        do k = 1, n - 1
               do i = k + 1, n
                      a(i, k) = a(i, k) / a(k, k)
                      do j = k + 1, n
                             a(i, j) = a(i, j) - a(i, k) * a(k, j)
                      end do
               b(i) = b(i) - a(i, k) * b(k)
               end do
        end do
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! Back substitution
        x(n) = b(n) / a(n, n)

        do i = n - 1, 1, -1
               total = b(i)
               do j = i + 1, n
                      total = total - a(i, j) * x(j)
               end do

               x(i) = total / a(i, i)
        end do
        !-------------------------------------------------------------------------------
end function gaussian
```

The function `gaussian` takes `n`, `a` and `b` as input and outputs `x`, the solution. This is similar to the program `Code_4_naive_gaussian.f90` discussed in Chapter 3. In this case, however, we cannot store the solution in the constant matrix `b`, as we have already taken that as an input. We therefore modify the program slightly so that we store the solutions in a separate matrix `x`.

The full program is as follows:

```
!===============================================================================
! Code_5_polynomial.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program interpolates the polynomial for a given set of points using the direct
! method.
!===============================================================================

program polynomial
        implicit none
```

```
!-------------------------------------------------------------------------------
! Declare variables
integer :: n, i, j
real*16 :: point, total = 0.0
real*16, dimension(:,:), allocatable :: a
real*16, dimension(:), allocatable :: x, y, coeff
character :: confirmed
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Ask user to specify the number of data points
write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
read(*,*) n
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Allocate n
allocate(x(n))
allocate(y(n))
allocate(coeff(n))
allocate(a(n,n))
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Ask user to specify values
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your INDEPENDENT variables."

       do i = 1, n
              read(*,*) x(i)
       end do

       write(*,*) "Your list of independent variable values is:"
       do i = 1, n
              write(*,"(f10.5)") x(i)
       end do

write(*,*) "Please confirm. Y/N"
read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
```

```
        write(*,*) "These are the values of your DEPENDENT variables."

        do i = 1, n
               read(*,*) y(i)
        end do

        write(*,*) "Your list of dependent variable values is:"
        write(*,"(a10, 2x, a10)") "x", "y"
        do i = 1, n
               write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

write(*,"(a44, 1x)", advance = "no") "At what point would you like to interpolate?"
read(*,*) point
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Fill matrix a
do i = 1, n
        do j = 1, n
               a(i, 1) = 1
               a(i, j) = x(i) ** (j - 1)
         end do
end do
!-------------------------------------------------------------------------------

coeff = gaussian(n, a, y)

!-------------------------------------------------------------------------------
! Obtain polynomial and the value of point at polynomial.
write(*,*)
do i = 1, n
        total = total + coeff(i) * point ** (i - 1)
end do
write(*,"(a67, 1x, f10.5)") "The value of your chosen point at the interpolating
polynomial is: ", total
!-------------------------------------------------------------------------------

contains
!===============================================================================
! This function implements the Naive Gaussian elimination algorithm.
!===============================================================================
        function gaussian(n, a, b) result(x)
```

```
            !-------------------------------------------------------------------------------
            ! Declare variables
            integer :: n, i, j, k
            real*16 :: total
            real*16, dimension(n, n) :: a
            real*16, dimension(n) :: b, x
            !-------------------------------------------------------------------------------

            ! Implement Naive Gaussian algorithm

            !-------------------------------------------------------------------------------
            ! Forward elimination
            do k = 1, n - 1
                   do i = k + 1, n
                          a(i, k) = a(i, k) / a(k, k)
                          do j = k + 1, n
                                 a(i, j) = a(i, j) - a(i, k) * a(k, j)
                          end do
                   b(i) = b(i) - a(i, k) * b(k)
                   end do
            end do
            !-------------------------------------------------------------------------------

            !-------------------------------------------------------------------------------
            ! Back substitution
            x(n) = b(n) / a(n, n)

            do i = n - 1, 1, -1
                   total = b(i)
                   do j = i + 1, n
                          total = total - a(i, j) * x(j)
                   end do

                   x(i) = total / a(i, i)
            end do
            !-------------------------------------------------------------------------------
      end function gaussian
end program polynomial
```

Executing the program,

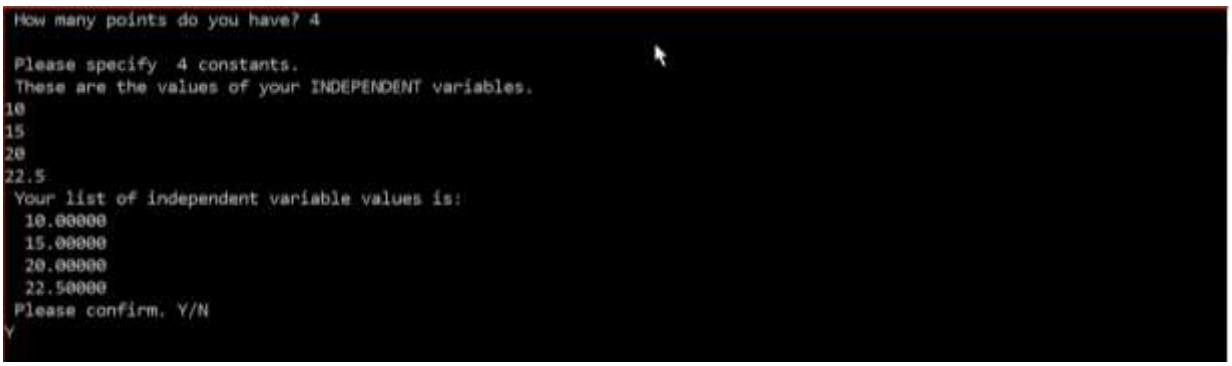

Figure 4.4. *User specifying the number of points and the x-values*

Figure 4.5. *User specifying the y-values*

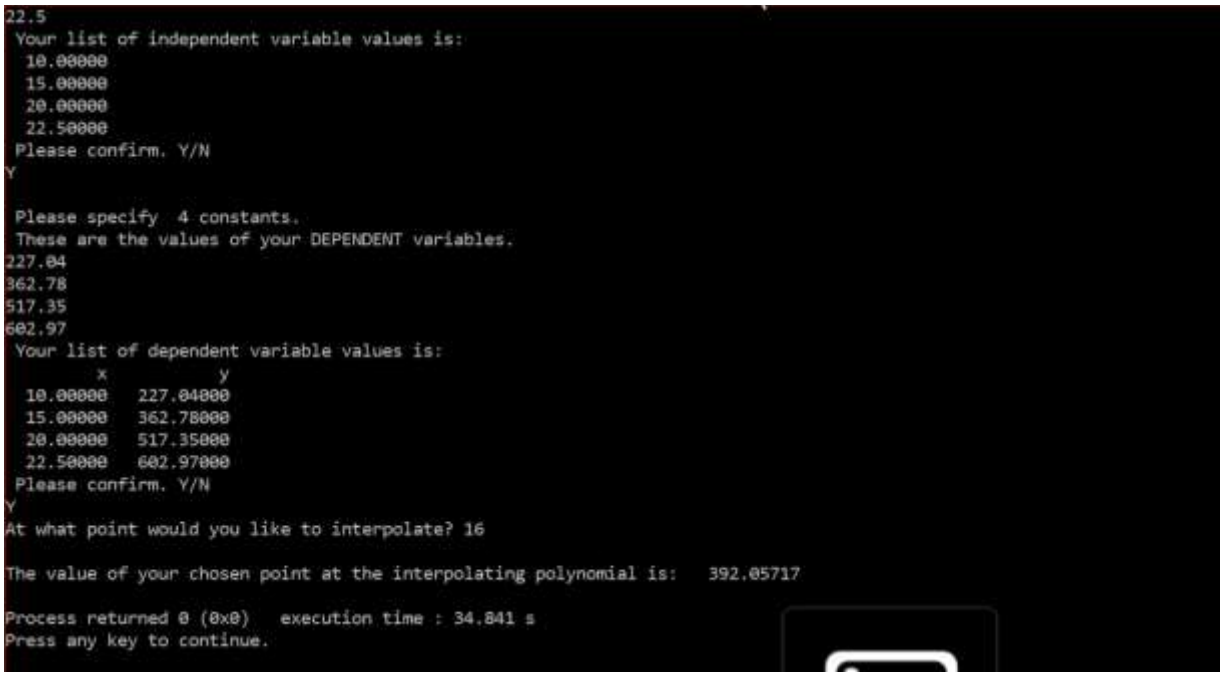

Figure 4.6. *User specifying the interpolation point, output*

we find that the solution to (4.9) is given by

$$v(16) = \mathbf{392.05717}\ \boldsymbol{m/s}$$

Lagrange interpolation

Given a general set of $n + 1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, another method known as *Lagrange interpolation* can be used to obtain the interpolating polynomial. However, as opposed to the polynomial approximation method, the Lagrange interpolating polynomial is given by

$$f_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i) \tag{4.10}$$

where $f_n(x)$ denotes that (4.10) is the nth order Lagrange polynomial for the given set of points and

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{4.11}$$

Example 2

This example is adapted from Kaw (2011).

Using the same example as earlier,

The velocity of a rocket is a function of time, with the values shown in Table 1.

t (s)	v(t) (m/s)
10	227.04
15	362.78
20	517.35
22.5	602.97

Table 4.4. *Velocity as a function of time*

a) Find the third-order polynomial corresponding to these values.
b) Determine the value of the velocity at t = 16 using Lagrange interpolation.
c) Find the absolute relative approximation error for the third-order polynomial.

Solutions

a) From (4.10), the interpolating polynomial will be of the form'

$$\begin{aligned} v(t) &= \sum_{i=0}^{3} L_i(t) v(t_i) \\ &= L_0(t)v(t_0) + L_1(t)v(t_1) + L_2(t)v(t_2) + L_3(t)v(t_3) \end{aligned} \tag{4.12}$$

If we let $t_0 = 10, t_1 = 15$ and so on, we get:

$$L_0(t) = \prod_{\substack{j=0 \\ j \neq 0}}^{n} \frac{t - t_j}{t_0 - t_j}$$

$$= \left(\frac{t-t_1}{t_0-t_1}\right)\left(\frac{t-t_2}{t_0-t_2}\right)\left(\frac{t-t_3}{t_0-t_3}\right)$$

$$L_1(t) = \prod_{\substack{j=0 \\ j\neq 1}}^{n} \frac{t-t_j}{t_1-t_j}$$

$$= \left(\frac{t-t_0}{t_1-t_0}\right)\left(\frac{t-t_2}{t_1-t_2}\right)\left(\frac{t-t_3}{t_1-t_3}\right)$$

$$L_2(t) = \prod_{\substack{j=0 \\ j\neq 2}}^{n} \frac{t-t_j}{t_2-t_j}$$

$$= \left(\frac{t-t_0}{t_2-t_0}\right)\left(\frac{t-t_1}{t_2-t_1}\right)\left(\frac{t-t_3}{t_2-t_3}\right)$$

$$L_3(t) = \prod_{\substack{j=0 \\ j\neq 3}}^{n} \frac{t-t_j}{t_3-t_j}$$

$$= \left(\frac{t-t_0}{t_3-t_0}\right)\left(\frac{t-t_1}{t_3-t_1}\right)\left(\frac{t-t_2}{t_3-t_2}\right)$$

Substituting these to (4.12) yields

$$\begin{aligned} v(t) &= L_0(t)v(t_0) + L_1(t)v(t_1) + L_2(t)v(t_2) + L_3(t)v(t_3) \\ &= \left(\frac{t-t_1}{t_0-t_1}\right)\left(\frac{t-t_2}{t_0-t_2}\right)\left(\frac{t-t_3}{t_0-t_3}\right)v(t_0) + \left(\frac{t-t_0}{t_1-t_0}\right)\left(\frac{t-t_2}{t_1-t_2}\right)\left(\frac{t-t_3}{t_1-t_3}\right)v(t_1) \\ &\quad + \left(\frac{t-t_0}{t_2-t_0}\right)\left(\frac{t-t_1}{t_2-t_1}\right)\left(\frac{t-t_3}{t_2-t_3}\right)v(t_2) + \left(\frac{t-t_0}{t_3-t_0}\right)\left(\frac{t-t_1}{t_3-t_1}\right)\left(\frac{t-t_2}{t_3-t_2}\right)v(t_3) \\ &= \left(\frac{\boldsymbol{t-15}}{\boldsymbol{10-15}}\right)\left(\frac{\boldsymbol{t-20}}{\boldsymbol{10-20}}\right)\left(\frac{\boldsymbol{t-22.5}}{\boldsymbol{10-22.5}}\right)\boldsymbol{(227.04)} \\ &\quad + \left(\frac{\boldsymbol{t-10}}{\boldsymbol{15-10}}\right)\left(\frac{\boldsymbol{t-20}}{\boldsymbol{15-20}}\right)\left(\frac{\boldsymbol{t-22.5}}{\boldsymbol{15-22.5}}\right)\boldsymbol{(362.78)} \\ &\quad + \left(\frac{\boldsymbol{t-10}}{\boldsymbol{20-10}}\right)\left(\frac{\boldsymbol{t-15}}{\boldsymbol{20-15}}\right)\left(\frac{\boldsymbol{t-22.5}}{\boldsymbol{20-22.5}}\right)\boldsymbol{(517.35)} \\ &\quad + \left(\frac{\boldsymbol{t-10}}{\boldsymbol{22.5-10}}\right)\left(\frac{\boldsymbol{t-15}}{\boldsymbol{22.5-15}}\right)\left(\frac{\boldsymbol{t-20}}{\boldsymbol{22.5-20}}\right)\boldsymbol{(602.97)} \end{aligned} \tag{4.13}$$

b) Given (4.13), we can simply substitute $t = 16$ to $v(t)$ to get the velocity. In doing so, we get

$$v(16) = \left(\frac{16-15}{10-15}\right)\left(\frac{16-20}{10-20}\right)\left(\frac{16-22.5}{10-22.5}\right)(227.04)$$
$$+\left(\frac{16-10}{15-10}\right)\left(\frac{16-20}{15-20}\right)\left(\frac{16-22.5}{15-22.5}\right)(362.78)$$
$$+\left(\frac{16-10}{20-10}\right)\left(\frac{16-15}{20-15}\right)\left(\frac{16-22.5}{20-22.5}\right)(517.35)$$
$$+\left(\frac{16-10}{22.5-10}\right)\left(\frac{16-15}{22.5-15}\right)\left(\frac{16-20}{22.5-20}\right)(602.97)$$
$$= \boldsymbol{392.06\ m/s}$$

c) We need to find the relative error with respect to the second-order Lagrange approximation. The second-order Lagrange approximation will vary based on the choice of data points to include. For our example, we include the first three data points in our approximation. Repeating what we did in the first part of (a), we should get a system of the following form

$$v(t) = \sum_{i=0}^{2} L_i(t)v(t_i)$$
$$= L_0(t)v(t_0) + L_1(t)v(t_1) + L_2(t)v(t_2) \qquad (4.14)$$

If we let $t_0 = 10, t_1 = 15$ and so on in (4.14) and substitute $t = 16$ to the resulting polynomial, we get:

$$v(16) = \mathbf{392.19\ m/s}$$

Thus, we can now compute for the relative approximate error associated with the third-order polynomial approximation, given by

$$|RAE| = \left|\frac{392.06 - 392.19}{392.06}\right| \times 100 = \mathbf{0.033269}\%$$

For comparison, we can compute for the velocity using a first-order polynomial approximation and compare how the relative approximate error changes as we add more and more terms to the approximation. Using $t = 15$ and $t = 20$ as our two data points, we get an estimate to $v(16)$ equal to

$$v(16) = \boldsymbol{393.7\ m/s}$$

Computing for the relative approximate error associated with the second-order polynomial approximation,

$$|RAE| = \left|\frac{392.19 - 393.7}{392.19}\right| \times 100 = \mathbf{0.38410}\%$$

Thus, as in the earlier example, we can see that the relative approximate error actually decreases as we add more and more terms. Hence, in the program that follows, we will strive to obtain the most accurate polynomial approximation possible.

Lagrange interpolation – Fortran example

To implement Lagrange interpolation in Fortran, we first need to see the algorithm in a flowchart. The algorithm is summarized in Figure 4.4

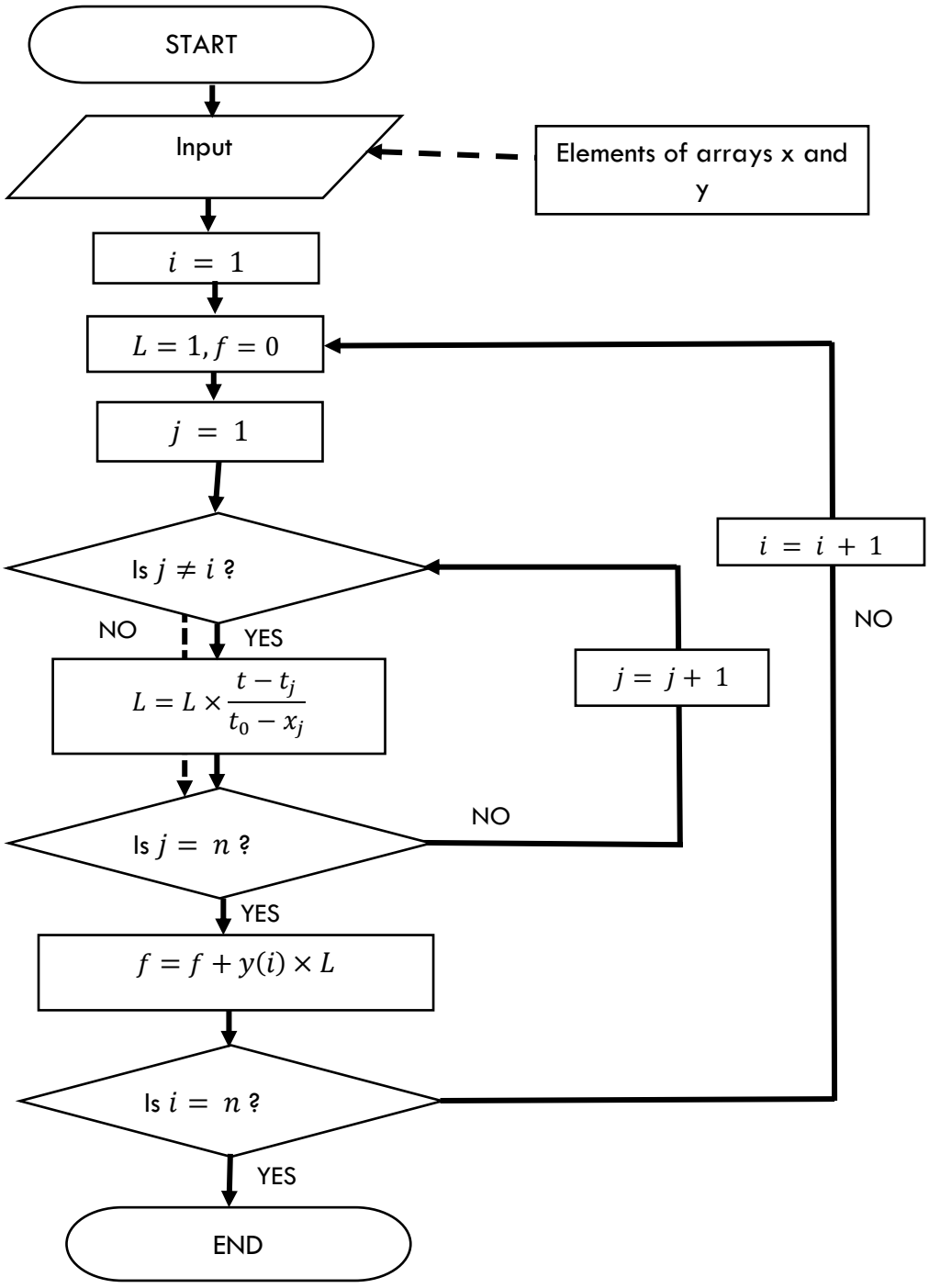

Figure 4.4. *Flowchart for Lagrange interpolation*

```
!=============================================================================================
! Code_6_lagrange.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program interpolates the polynomial for a given set of points using the Lagrange
method.
!=============================================================================================

program lagrange

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	integer :: n, i, j
	real*16 :: point, a = 1.0, b = 1.0, total = 0.0
	real*16, dimension(:), allocatable :: x, y
	character :: confirmed
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Ask user to specify the number of data points
	write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
	read(*,*) n
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Allocate n
	allocate(x(n))
	allocate(y(n))
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables. `n` stores the number of unknowns while `i` and `j` are counters. The dimensions of arrays `x` and `y` are made allocatable so that the user can specify them. `x` and `y` represent the arrays for the independent and dependent variable, respectively. `point` stores the point of interest, `total` stores the estimate for the point of interest and `a` and `b` store elements of L_i. `confirmed` is use to track whether or not the user specified the values of the matrices to his/her taste. The user is then prompted to specify the number of data points to be used, after which the program allocates the dimensions of the arrays based on the user's specification.

```
!---------------------------------------------------------------------------------------
! Ask user to specify values
do while(confirmed /= "Y")
	write(*,*)
	write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
	write(*,*) "These are the values of your INDEPENDENT variables."
```

```
        do i = 1, n
                read(*,*) x(i)
        end do

        write(*,*) "Your list of independent variable values is:"
        do i = 1, n
                write(*,"(f10.5)") x(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
        write(*,*)
        write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
        write(*,*) "These are the values of your DEPENDENT variables."

        do i = 1, n
                read(*,*) y(i)
        end do

        write(*,*) "Your list of dependent variable values is:"
        write(*,"(a10, 2x, a10)") "x", "y"
        do i = 1, n
                write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

write(*,"(a44, 1x)", advance = "no") "At what point would you like to interpolate?"
read(*,*) point
!-------------------------------------------------------------------------------
```

The user is then prompted to specify the data points. The user is asked to confirm his selection so that he/she may see whether the values had been inputted correctly to the program. The user is also asked to specify the point to be evaluated using the interpolating polynomial.

```
!-------------------------------------------------------------------------------
! Implement Lagrange interpolation
do i = 1, n
        a = 1.0
        b = 1.0
```

```
        do j = 1, n
                if(j /= i) then
                        a = a * (point - x(j))
                        b = b * (x(i) - x(j))
                end if
        end do
        total = total + y(i) * (a / b)
end do
write(*,"(a67, 1x, f20.10)") "The value of your chosen point at the interpolating
polynomial is: ", total
!-------------------------------------------------------------------------------------------
```

The preceding code snippet implements the algorithm for Lagrange interpolation and is fairly straightforward based on the computations in Example 2. The numerator (a) and denominator (b) of L_i are first computed, taking note that $j \neq i$ (see 4.11). $f(x_i)$ is then multiplied by L_i and these products are added together. This is done until $i = n$, as in (4.10).

The full program is as follows:

```
!===========================================================================================
! Code_6_lagrange.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program interpolates the polynomial for a given set of points using the Lagrange
! method.
!===========================================================================================

program lagrange

        implicit none
        !-------------------------------------------------------------------------------------
        ! Declare variables
        integer :: n, i, j
        real*16 :: point, a = 1.0, b = 1.0, total = 0.0
        real*16, dimension(:), allocatable :: x, y
        character :: confirmed
        !-------------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------------
        ! Ask user to specify the number of data points
        write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
        read(*,*) n
        !-------------------------------------------------------------------------------------
```

```
!-------------------------------------------------------------------------------
! Allocate n
allocate(x(n))
allocate(y(n))
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Ask user to specify values
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your INDEPENDENT variables."

       do i = 1, n
              read(*,*) x(i)
       end do

       write(*,*) "Your list of independent variable values is:"
       do i = 1, n
              write(*,"(f10.5)") x(i)
       end do

       write(*,*) "Please confirm. Y/N"
       read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your DEPENDENT variables."

       do i = 1, n
              read(*,*) y(i)
       end do

       write(*,*) "Your list of dependent variable values is:"
       write(*,"(a10, 2x, a10)") "x", "y"
       do i = 1, n
              write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
       end do

       write(*,*) "Please confirm. Y/N"
       read(*,*) confirmed
end do

write(*,"(a44, 1x)", advance = "no") "At what point would you like to interpolate?"
read(*,*) point
!-------------------------------------------------------------------------------
```

```
        !-------------------------------------------------------------------------------
        ! Implement Lagrange interpolation
        do i = 1, n
                a = 1.0
                b = 1.0
                do j = 1, n
                        if(j /= i) then
                                a = a * (point - x(j))
                                b = b * (x(i) - x(j))
                        end if
                end do
                total = total + y(i) * (a / b)
        end do
        write(*,"(a67, 1x, f20.10)") "The value of your chosen point at the interpolating
        polynomial is: ", total
        !-------------------------------------------------------------------------------
end program lagrange
```

Executing the program,

```
 How many points do you have? 4

 Please specify  4 constants.
 These are the values of your INDEPENDENT variables.
10
12
20
22.5
 Your list of independent variable values is:
  10.00000
  12.00000
  20.00000
  22.50000
 Please confirm. Y/N
Y
```

Figure 4.5. *User specifying the number of points and the x-values*

```
  15.00000
  20.00000
  22.50000
 Please confirm. Y/N
Y

 Please specify  4 constants.
 These are the values of your DEPENDENT variables.
227.04
362.78
517.35
602.97
 Your list of dependent variable values is:
         x             y
  10.00000   227.04000
  15.00000   362.78000
  20.00000   517.35000
  22.50000   602.97000
 Please confirm. Y/N
Y
```

Figure 4.6. *User specifying the y-values*

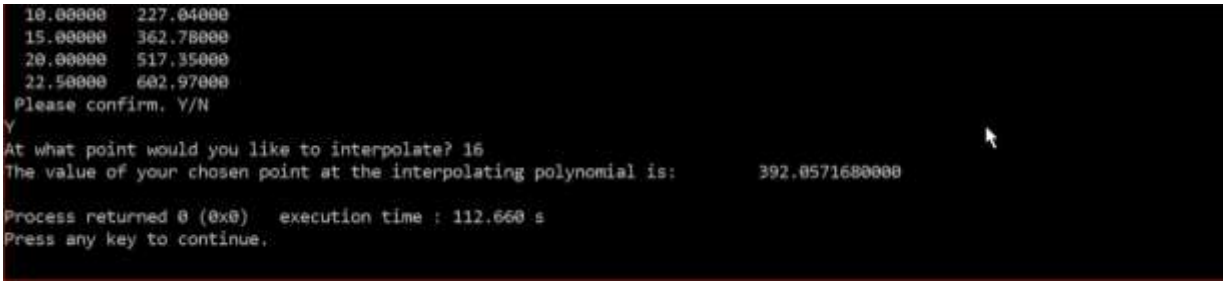

Figure 4.7. *User specifying the interpolation point, output*

we find that the solution to (4.9) is given by

$$v(16) = \mathbf{392.057168}\ \boldsymbol{m/s}$$

Applications – Interpolating from physical data

Polynomial interpolation can conveniently be used to determine the relationship between two variables in an experiment. As an example, we will use the experiment on centripetal force, found here: http://www.dlsu.edu.ph/academics/colleges/cos/physics/experiments.asp.

For the first part of that experiment, the mass of the hanging object and the centripetal force are kept constant, while the radius is varied. For every possible value of the radius r, the time for ten revolutions is recorded and divided by 10 to obtain the period T. This is then squared to obtain T^2. r is then graphed against T^2.

We know from physics that the centripetal force is given by

$$F = \frac{mv^2}{r} \tag{4.15}$$

But then we also know that

$$v = \frac{2\pi r}{T} \tag{4.16}$$

Substituting (4.16) to (4.15), we get

$$F = \frac{4\pi^2 mr}{T^2} \tag{4.17}$$

Since r is graphed against T^2 in the first part of the experiment, we solve for r. This yields

$$r = \frac{F}{4\pi^2 m} T^2 \tag{4.18}$$

However, for our interpolation, we will also make use of constant T and T^3 terms, which will serve as correction terms to account for other factors in the physical system.

Using the following data,

r (m)	T (s)
0.7	0.782
0.8	0.833
0.9	0.882
1.0	0.932

Table 4.3. *Data from centripetal force experiment*

and noting that $m = 0.0656\ \text{kg}$ and that $F = (0.0656\ \text{kg})\left(\frac{9.8\text{m}}{\text{s}^2}\right) = 0.64288\ \text{N}$, we can now proceed with the interpolation. Using a slightly edited version of `Code_5_polynomial.f90`, we get

```
How many points do you have? 4

Please specify  4 constants.
These are the values of your INDEPENDENT variables.
0.7
0.8
0.9
1.0
Your list of independent variable values is:
  0.70000
  0.80000
  0.90000
  1.00000
Please confirm. Y/N
Y

Please specify  4 constants.
These are the values of your DEPENDENT variables.
0.782
0.833
0.882
0.932
Your list of dependent variable values is:
        x          y
  0.70000    0.78200
  0.80000    0.83300
  0.90000    0.88200
  1.00000    0.93200
Please confirm. Y/N
```

Figure 4.7. *User specifying the x and y values*

$$\boldsymbol{r = 0.1170 + 1.6150T - 1.3000T^2 + 0.5000T^3} \qquad (4.19)$$

CHAPTER 5: *NUMERICAL DIFFERENTIATION*

Introduction

The derivative is a key idea in describing physical phenomena, as it describes the rate of change. Given the constantly changing nature of the universe, most concepts in physics can be aptly described by a derivative. From simple kinematics to electric circuits, the derivative finds constant use in the explanation of physical phenomena in different fields.

With the advent of symbolic calculators like MATLAB and Mathematica, finding the derivative of a messy function is simply a few keystrokes away. However, as students of computational physics, we must know how to compute derivatives not only analytically but also numerically.

In this chapter, we will discuss the different methods of finding the derivative of a (1) continuous and (2) discrete function.

Differentiation of continuous functions

We know from calculus that

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{5.1}$$

However, instead of letting Δx approach zero, we can approximate $f'(x)$ by simply choosing a value of Δx that is approximately close to zero. If we do this, we end up with

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{5.2}$$

(5.2) is referred to as the *forward difference approximation of the first derivative*. However, we can motivate this derivation further by invoking $f(x)$'s Taylor expansion. We can Taylor expand $f(x_{i+1})$ around x_i, provided that its derivatives are continuous between x_i and x_{i+1}, yielding

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \cdots \tag{5.3}$$

We let $\Delta x = x_{i+1} - x_i$. Thus, substituting this to (5.3),

$$f(x_{i+1}) = f(x_i) + f'(x_i)(\Delta x) + \frac{f''(x_i)}{2!}(\Delta x)^2 + \frac{f'''(x_i)}{3!}(\Delta x)^3 + \cdots \tag{5.4}$$

Solving for $f'(x_i)$,

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} - \frac{f''(x_i)}{2!}(\Delta x) + \cdots$$

$$\boldsymbol{f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} + O(\Delta x)} \tag{5.5}$$

where $O(\Delta x)$ shows that the approximation error is roughly of the order of Δx. This ends up being the better way of deriving the formula, as it shows us roughly how large the approximation error is.

We can likewise Taylor expand $f(x_{i-1})$ around x_i, provided that its derivatives are continuous between x_{i-1} and x_i, yielding

$$f(x_{i-1}) = f(x_i) - f'(x_i)(\Delta x) + \frac{f''(x_i)}{2!}(\Delta x)^2 - \frac{f'''(x_i)}{3!}(\Delta x)^3 + \cdots \tag{5.6}$$

where $\Delta x = x_{i-1} - x_i < 0$

Using (5.6) to solve for $f'(x_i)$,

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta x} - \frac{f''(x_i)}{2!}(\Delta x) + \cdots$$

$$\boldsymbol{f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta x} + O(\Delta x)} \tag{5.7}$$

(5.7) is known as the *backward difference approximation of the first derivative*. Both the approximation errors in the forward and backward difference methods are of the same order of magnitude. We find that we can reduce the approximation error even further by taking (5.4) and subtracting (5.6) from it. Doing this, we get

$$f(x_{i+1}) - f(x_{i-1}) = f'(x_i)(2\Delta x) + \frac{2f'''(x_i)}{3!}(\Delta x)^3 + \cdots \tag{5.8}$$

Thus,

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} - \frac{f'''(x_i)}{3!}(\Delta x)^2 + \cdots$$

$$\boldsymbol{f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} + O(\Delta x)^2} \tag{5.9}$$

As you can see, (5.9) actually improves on the previous methods, since its error is now roughly of the order of $(\Delta x)^2$. This method of approximation is called the *central difference approximation of the first derivative* and is the most accurate of the three methods.

In addition to estimating first derivatives, we can extend this argument to second derivatives. For this, we use both the Taylor expansion of $f(x_{i+2})$ and $f(x_{i+1})$ in (5.4).

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2\Delta x) + \frac{f''(x_i)}{2!}(2\Delta x)^2 + \frac{f'''(x_i)}{3!}(2\Delta x)^3 + \cdots \tag{5.10}$$

Subtracting twice of (5.4) to (5.10) to eliminate the $f'(x_i)$ term, we get

$$f(x_{i+2}) - 2f(x_{i+1}) = -f(x_i) + f''(x_i)(\Delta x)^2 + f'''(x_i)(\Delta x)^3 \tag{5.11}$$

Solving for $f''(x_i)$ in (5.11), we get

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{(\Delta x)^2} - f'''(x_i)(\Delta x)$$

$$\boldsymbol{f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{(\Delta x)^2} + O(\Delta x)} \quad (5.12)$$

(5.12) is the *forward difference approximation of the second derivative*. By extension and using the same logic,

$$\boldsymbol{f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{(\Delta x)^2} + O(\Delta x)} \quad (5.13)$$

(5.13) is the *backward difference approximation of the second derivative*.

Using (5.4) and (5.6), we can also derive the *central difference approximation of the second derivative*. Adding (5.4) and (5.6),

$$f(x_{i+1}) + f(x_{i-1}) = 2f(x_i) + f''(x_i)(\Delta x)^2 + f^{(4)}(x_i)\frac{(\Delta x)^4}{12} + \cdots$$

so that

$$\boldsymbol{f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(\Delta x)^2} + O(\Delta x)^2} \quad (5.14)$$

Example 1

This example is adapted from Kaw (2011).

The velocity of a rocket is given by

$$v(t) = 2000\ln\left[\frac{14 \times 10^4}{14 \times 10^4 - 2100t}\right] - 9.8t, 0 \le t \le 30$$

d) Find the acceleration at $t = 16$s using forward, backward and central difference approximation, using a step size of $\Delta t = 2$s.
e) Calculate the absolute relative true error for each case in (a).
f) The second derivative of velocity is known as the jerk. Find the jerk at $t = 16$s using forward, backward and central difference approximation, using a step size of $\Delta t = 2$s.
g) Calculate the absolute relative true error for each case in (c).

Solutions

d) From (5.5), (5.7) and (5.9),

$$a(t)_{forward} = \frac{v(t + \Delta t) - v(t)}{\Delta t}$$

$$a(16)_{forward} = \frac{v(18) - v(16)}{2}$$

$$= \frac{453.02 - 392.07}{2} = \boldsymbol{30.475\ m/s^2}$$

$$a(t)_{backward} = \frac{v(t) - v(t - \Delta t)}{\Delta t}$$

$$a(16)_{backward} = \frac{v(16) - v(14)}{2}$$

$$= \frac{392.07 - 334.24}{2} = \mathbf{28.915\ m/s^2}$$

$$a(t)_{central} = \frac{v(t + \Delta t) - v(t - \Delta t)}{2\Delta t}$$

$$a(16)_{central} = \frac{v(18) - v(14)}{2(2)}$$

$$= \frac{453.02 - 334.24}{4} = \mathbf{29.695\ m/s^2}$$

(b) We know from physics that

$$a = \frac{dv}{dt}$$

Since

$$v(t) = 2000\ln\left[\frac{14 \times 10^4}{14 \times 10^4 - 2100t}\right] - 9.8t$$

we simply find its derivative.

$$a(t) = \frac{dv}{dt}$$

$$= \frac{d}{dt}\left(2000\ln\left[\frac{14 \times 10^4}{14 \times 10^4 - 2100t}\right] - 9.8t\right)$$

$$= 2000\left(\frac{14 \times 10^4 - 2100t}{14 \times 10^4}\right)\left(-\frac{14 \times 10^4}{(14 \times 10^4 - 2100t)^2}\right)(-2100) - 9.8$$

$$= \frac{-4040 - 29.4t}{-200 + 3t}$$

At t =16s,

$$a(16) = \frac{-4040 - 29.4(16)}{-200 + 3(16)} = \mathbf{29.674\ m/s^2}$$

Thus, using the formula for the relative true error,

$$|RTE| = \left|\frac{true\ -approximate}{true}\right|$$

$$|RTE_{forward}| = \left|\frac{29.674 - 30.475}{29.674}\right| \times 100 = \mathbf{2.6993}\%$$
$$|RTE_{backward}| = \left|\frac{29.674 - 28.915}{29.674}\right| \times 100 = \mathbf{2.5578}\%$$
$$|RTE_{central}| = \left|\frac{29.674 - 29.695}{29.674}\right| \times 100 = \mathbf{0.07077}\%$$

c) From (5.12), (5.13) and (5.14),

$$j(t)_{forward} = \frac{v(t + 2\Delta t) - 2v(t + \Delta t) + v(t)}{(\Delta t)^2}$$
$$j(16)_{forward} = \frac{v(20) - 2v(18) + v(16)}{(2)^2}$$
$$= \frac{517.35 - 2(453.02) + 392.07}{4} = \mathbf{0.8450\ m/s^3}$$

$$j(t)_{backward} = \frac{v(t) - 2v(t - \Delta t) + v(t - 2\Delta t)}{(\Delta t)^2}$$
$$j(16)_{backward} = \frac{v(16) - 2v(14) + v(12)}{(2)^2}$$
$$= \frac{392.07 - 2(334.24) + 279.30}{4} = \mathbf{0.7225\ m/s^3}$$

$$j(t)_{central} = \frac{v(t + \Delta t) - 2v(t) + v(t - \Delta t)}{(\Delta t)^2}$$
$$j(16)_{central} = \frac{v(18) - 2v(16) + v(14)}{(2)^2}$$
$$= \frac{453.02 - 2(392.07) + 334.24}{4} = \mathbf{0.78\ m/s^3}$$

(b) We know from physics that

$$j = \frac{da}{dt}$$

Since

$$a(t) = \frac{-4040 - 29.4t}{-200 + 3t}$$

we simply find its derivative.

$$j(t) = \frac{da}{dt}$$

$$= \frac{d}{dt}\left(\frac{-4040 - 29.4t}{-200 + 3t}\right)$$
$$= \frac{(-200 + 3t)(-29.4) - (-4040 - 29.4t)(3)}{(-200 + 3t)^2}$$
$$= \frac{18000}{(-200 + 3t)^2}$$

At t =16s,

$$j(16) = \frac{18000}{[-200 + 3(16)]^2} = \mathbf{0.7791\ m/s^3}$$

Thus, using the formula for the relative true error,

$$|RTE| = \left|\frac{true\ -approximate}{true}\right|$$

$$\left|RTE_{forward}\right| = \left|\frac{0.7791\ - 0.8450}{0.7791}\right| \times 100 = \mathbf{8.4585\%}$$

$$|RTE_{backward}| = \left|\frac{0.7791 - 0.7225}{0.7791}\right| \times 100 = \mathbf{7.2648\%}$$

$$|RTE_{central}| = \left|\frac{0.7791 - 0.78}{0.7791}\right| \times 100 = \mathbf{0.11552\%}$$

Observe that in both cases, the central difference approximation gives the most accurate estimate, as we had established earlier.

Differentiation of continuous functions – Fortran example

Implementing this in Fortran is fairly straightforward based on the formulae. As such, there is no need to diagrammatically discuss the algorithm using a flowchart and we can jump straight to the code.

```
!=============================================================================================
! Code_7_numerical_diff.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the derivative at a given point using the forward, backward and
! central difference approximation methods.
!=============================================================================================

program numerical_diff

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	real*16 :: num, step, diff
	integer :: choice1, choice2
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "Given the function f(x) = 2000*ln(14x10^4/(14x10^4 - 2100*i)) - 9.8*i,"

	write(*,*) "Choose a value at which to evaluate the 1st/2nd derivative of f(x): "
	read(*,*)  num

	write(*,*) "Choose a step size: "
	read(*, *) step

	write(*,*) "Choose a method: "
	write(*,*) "(1) forward difference"
	write(*,*) "(2) backward difference"
	write(*,*) "(3) central difference"
	read(*,*) choice1

	write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
	read(*,*) choice2
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`num` stores the number at which the derivative will be computed, `step` stores the step size and `diff` stores the derivative. The user is given 2 choices: (1) the choice of method to use (stored in `choice1`) and (2) whether to evaluate the first or second derivative (stored in `choice2`). The user is then prompted to enter values for `num`, `step`, `choice1` and `choice2`.

```
!-------------------------------------------------------------------------------------------
! Implement forward, central and backward difference methods.
if(choice2 == 1) then
       if(choice1 == 1) then
              diff = (fx(num + step) - fx(num))/step
       elseif(choice1 == 2) then
              diff = (fx(num) - fx(num - step))/step
       elseif(choice1 == 3) then
              diff = (fx(num + step) - fx(num - step))/(2 * step)
       else
              write(*,*) "Try again. Please enter 1, 2 or 3."
       end if

       write(*,"(a30, 1x, f20.10, 1x, a4, f20.10, a2)") "The derivative of f(x) at x = ",
       num, " is ", diff, " ."
       write(*,"(a37, f20.10, a2)") "The actual value of the derivative is", deriv(num),
       " ."

elseif(choice2 == 2) then
       if(choice1 == 1) then
              diff = (fx(num + 2 * step) - 2 * fx(num + step) + fx(num))/step ** 2
       elseif(choice1 == 2) then
              diff = (fx(num) - 2 * fx(num - step) + fx(num - 2 * step))/step ** 2
       elseif(choice1 == 3) then
              diff = (fx(num + step) - 2 * fx(num) + fx(num - step))/step ** 2
       else
              write(*,*) "Try again. Please enter 1, 2 or 3."
       end if

       write(*,"(a30, 1x, f20.10, 1x, a4, f20.10, a2)") "The derivative of f(x) at x = ",
       num, " is ", diff, " ."
       write(*,"(a37, f20.10, a2)") "The actual value of the derivative is", deriv2(num),
       " ."

else
       write(*,*) "Try again. Please enter either 1 or 2."

end if
!-------------------------------------------------------------------------------------------
```

This snippet implements the (1) forward, (2) backward and (3) central difference approximation methods for the first and second derivatives, from the formulae in (5.5), (5.7), (5.9), (5.12), (5.13) and (5.14). The estimated value of the derivative, along with its true value (obtained from functions `deriv` and `deriv2`, to be discussed later) are also printed. The user is also prompted if (s)he enters an invalid number. The program terminates, and the user would have to start all over again if (s)he makes an invalid choice.

```
contains
!=============================================================================================
! This function specifies the function whose derivative is to be found and its first and
! second derivatives. In this case, it is 2000 * log(14e4/(14e4 - 2100 * i)) - 9.8 * i.
! Edit if needed.
!=============================================================================================
       function fx(i) result(fxeval)
           real*16 :: fxeval, i

           fxeval = 2000 * log(14e4/(14e4 - 2100 * i)) - 9.8 * i
        end function fx

        function deriv(i) result(deriveval)
           real*16 :: deriveval, i

           deriveval = (-4040 - 29.4 * i) / (-200 + 3 * i)
        end function deriv

        function deriv2(i) result(deriv2eval)
           real*16 :: deriv2eval, i

           deriv2eval = 18000/(-200 + 3 * i) ** 2
        end function deriv2
```

The functions `fx`, `deriv` and `deriv2` specify the function of interest and its first and second derivatives, respectively. These functions can be edited to suit the user's needs.

The full program is as follows:

```
!=============================================================================================
! Code_7_numerical_diff.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the derivative at a given point using the forward, backward and
! central difference approximation methods.
!=============================================================================================

program numerical_diff

       implicit none
       !-------------------------------------------------------------------------------------
       ! Declare variables
       real*16 :: num, step, diff
       integer :: choice1, choice2
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! User input
       write(*,*) "Given the function f(x) = 2000*ln(14x10^4/(14x10^4 - 2100*i)) - 9.8*i,"
```

```
write(*,*) "Choose a value at which to evaluate the 1st/2nd derivative of f(x): "
read(*,*)  num

write(*,*) "Choose a step size: "
read(*, *) step

write(*,*) "Choose a method: "
write(*,*) "(1) forward difference"
write(*,*) "(2) backward difference"
write(*,*) "(3) central difference"
read(*,*) choice1

write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
read(*,*) choice2
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Implement forward, central and backward difference methods.
if(choice2 == 1) then
       if(choice1 == 1) then
              diff = (fx(num + step) - fx(num))/step
       elseif(choice1 == 2) then
              diff = (fx(num) - fx(num - step))/step
       elseif(choice1 == 3) then
              diff = (fx(num + step) - fx(num - step))/(2 * step)
       else
              write(*,*) "Try again. Please enter 1, 2 or 3."
       end if

       write(*,"(a30, 1x, f20.10, 1x, a4, f20.10, a2)") "The derivative of f(x) at x
       = ", num, " is ", diff, " ."
       write(*,"(a37, f20.10, a2)") "The actual value of the derivative is",
       deriv(num), " ."

elseif(choice2 == 2) then
       if(choice1 == 1) then
              diff = (fx(num + 2 * step) - 2 * fx(num + step) + fx(num))/step ** 2
       elseif(choice1 == 2) then
              diff = (fx(num) - 2 * fx(num - step) + fx(num - 2 * step))/step ** 2
       elseif(choice1 == 3) then
              diff = (fx(num + step) - 2 * fx(num) + fx(num - step))/step ** 2
       else
              write(*,*) "Try again. Please enter 1, 2 or 3."
       end if

       write(*,"(a30, 1x, f20.10, 1x, a4, f20.10, a2)") "The derivative of f(x) at x
       = ", num, " is ", diff, " ."
       write(*,"(a37, f20.10, a2)") "The actual value of the derivative is",
       deriv2(num), " ."
```

```
        else
               write(*,*) "Try again. Please enter either 1 or 2."

        end if
        !-------------------------------------------------------------------------------

        contains
        !===============================================================================
        ! This function specifies the function whose derivative is to be found and its first
        ! and second derivatives. In this case, it is 2000 * log(14e4/(14e4 - 2100 * i)) -
        ! 9.8 * i. Edit if needed.
        !===============================================================================
               function fx(i) result(fxeval)
                    real*16 :: fxeval, i

                    fxeval = 2000 * log(14e4/(14e4 - 2100 * i)) - 9.8 * i
                end function fx

                function deriv(i) result(deriveval)
                    real*16 :: deriveval, i

                    deriveval = (-4040 - 29.4 * i) / (-200 + 3 * i)
                end function deriv

                function deriv2(i) result(deriv2eval)
                    real*16 :: deriv2eval, i

                    deriv2eval = 18000/(-200 + 3 * i) ** 2
                end function deriv2

end program numerical_diff
```

Executing the program,

```
Given the function f(x) = 2000*ln(14x10^4/(14x10^4 - 2100*i)) - 9.8*i,
Choose a value at which to evaluate the 1st/2nd derivative of f(x):
16
Choose a step size:
2
Choose a method:
(1) forward difference
(2) backward difference
(3) central difference
3
Would you like to compute the (1) first or (2) second derivative?
1
The derivative of f(x) at x =        16.0000000000  is       29.6942054686 .
The actual value of the derivative is       29.6736841704 .

Process returned 0 (0x0)   execution time : 16.408 s
Press any key to continue.
```

Figure 5.7. *Solving for* $a(16)_{central}$ *using* *Code_7_numerical_diff.f90*

we find that the answers to Example 1 are given by

$$a(16)_{forward} = \mathbf{30.47390\ m/s^2}$$

$$a(16)_{backward} = \mathbf{28.91451\ m/s^2}$$

$$a(16)_{central} = \mathbf{29.69421\ m/s^2}$$

$$j(16)_{forward} = \mathbf{0.84515\ m/s^3}$$

$$j(16)_{backward} = \mathbf{0.72156\ m/s^3}$$

$$j(16)_{central} = \mathbf{0.77970\ m/s^3}$$

Differentiation of discrete functions

It is still possible to use the methods discussed for continuous functions, especially if the data points are equally spaced and available for a wide range of values. However, because data is often not available in this form, it is much better for us to interpolate a polynomial from the data and to differentiate the polynomial to find the first, second and succeeding derivatives.

There are two ways of interpolating a function discussed in Chapter 4: (1) the polynomial interpolation/direct method and (2) the Lagrange interpolation method. To get the derivative for a discrete set of points, we can simply use either method to interpolate a polynomial passing through all points and differentiate the polynomial afterwards.

The polynomial obtained through the first method is given by

$$P_n(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \ldots + a_nx^n \tag{5.15}$$

Thus, the first derivative is given by

$$P_n{}'(x) = a_1 + 2a_2x + 3a_3x^2 + \cdots + na_nx^{n-1} \tag{5.16}$$

and the second derivative by

$$P_n{}'(x) = 2a_2 + 6a_3x + \cdots + n(n-1)a_nx^{n-2} \tag{5.17}$$

In the case of the Lagrange polynomial, the interpolating polynomial is given by

$$f_n(x) = \sum_{i=0}^{n} L_i(x)f(x_i) \tag{5.18}$$

where $f_n(x)$ denotes that (4.10) is the nth order Lagrange polynomial for the given set of points and

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{5.19}$$

To get the first derivative, we derive $f_n(x)$ with respect to x. Since $f(x_i)$ is constant, it is simply a matter of differentiating $L_i(x)$ with respect to x. Hence,

$$f_n{}'(x) = \sum_{i=0}^{n} f(x_i)\frac{dL_i(x)}{dx}$$

Using the product rule,

$$\frac{dL_i(x)}{dx} = \sum_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j} \prod_{\substack{k=0 \\ k\neq i,j}}^{n} \frac{x - x_k}{x_i - x_k}$$

$$= \sum_{\substack{j=0 \\ j \neq i}}^{n} \frac{\prod_{\substack{k=0 \\ k \neq i,j}}^{n} x - x_k}{\prod_{\substack{k=0 \\ k \neq i}}^{n} x_i - x_k} \tag{5.20}$$

Thus,

$$f_n{}'(x) = \sum_{i=0}^{n} f(x_i) \sum_{\substack{j=0 \\ j \neq i}}^{n} \frac{\prod_{\substack{k=0 \\ k \neq i,j}}^{n} x - x_k}{\prod_{\substack{k=0 \\ k \neq i}}^{n} x_i - x_k} \tag{5.21}$$

Doing this again to (5.20) for the second derivative,

$$\frac{d^2 L_i(x)}{dx^2} = \sum_{\substack{j=0 \\ j \neq i}}^{n} \frac{1}{x_i - x_j} \sum_{\substack{k=0 \\ k \neq i,j}}^{n} \frac{\prod_{\substack{l=0 \\ l \neq i,j,k}}^{n} x - x_j}{\prod_{\substack{l=0 \\ l \neq i,j}}^{n} x_i - x_j} \tag{5.22}$$

This is much harder to simply, and so we leave it as is. Substituting,

$$f_n{}''(x) = \sum_{i=0}^{n} f(x_i) \sum_{\substack{j=0 \\ j \neq i}}^{n} \frac{1}{x_i - x_j} \sum_{\substack{k=0 \\ k \neq i,j}}^{n} \frac{\prod_{\substack{l=0 \\ l \neq i,j,k}}^{n} x - x_j}{\prod_{\substack{l=0 \\ l \neq i,j}}^{n} x_i - x_j} \tag{5.23}$$

The preceding formulae will not be very helpful in determining the derivatives analytically. However, they will come in handy when we implement both of these methods in code later on.

Example 2

This example is adapted from Kaw (2011).

Using the same example as in Chapter 4,

The velocity of a rocket is a function of time, with the values shown in Table 1.

t (s)	v(t) (m/s)
10	227.04
15	362.78
20	517.35
22.5	602.97

Table 5.1. Velocity as a function of time

d) Find the value of the acceleration of the rocket at t =16s using a third-order polynomial corresponding to these values.

e) Find the value of the acceleration of the rocket at t =16s using a second-order Lagrange polynomial (use t =10s, 15s and 20s as interpolation points) corresponding to these values.

Solutions

a) This problem is made easier by the fact that we have already computed the third-order polynomial for this table of values in Chapter 4.

The third-order polynomial is given by

$$v(t) = -4.2540 + 21.2166t + 0.1324t^2 + 0.0054347t^3$$

Thus,

$$\begin{aligned} a(t) &= \frac{dv}{dt} \\ &= \frac{d}{dt}(-4.2540 + 21.2166t + 0.1324t^2 + 0.0054347t^3) \\ &= 21.2166 + 0.2648t + 0.0163041t^2 \end{aligned}$$

Therefore,

$$\begin{aligned} a(16) &= 21.2166 + 0.2684(16) + 0.0163041(16)^2 \\ &= \mathbf{29.634\ m/s^2} \end{aligned}$$

b) Given (5.18) and (5.19), we know that

$$\begin{aligned} v(t) &= L_0(t)v(t_0) + L_1(t)v(t_1) + L_2(t)v(t_2) \\ &= \left(\frac{t-t_1}{t_0-t_1}\right)\left(\frac{t-t_2}{t_0-t_2}\right)v(t_0) + \left(\frac{t-t_0}{t_1-t_0}\right)\left(\frac{t-t_2}{t_1-t_2}\right)v(t_1) + \left(\frac{t-t_0}{t_2-t_0}\right)\left(\frac{t-t_1}{t_2-t_1}\right)v(t_2) \end{aligned}$$

Using the product rule, we find that

$$\begin{aligned} a(t) &= \frac{t-t_1+t-t_2}{(t_0-t_1)(t_0-t_2)}v(t_0) + \frac{t-t_0+t-t_2}{(t_1-t_0)(t_1-t_2)}v(t_1) + \frac{t-t_0+t-t_1}{(t_2-t_0)(t_2-t_1)}v(t_2) \\ &= \frac{2t-(t_1+t_2)}{(t_0-t_1)(t_0-t_2)}v(t_0) + \frac{2t-(t_0+t_2)}{(t_1-t_0)(t_1-t_2)}v(t_1) + \frac{2t-(t_0+t_1)}{(t_2-t_0)(t_2-t_1)}v(t_2) \end{aligned}$$

Thus,

$$\begin{aligned} a(16) &= \frac{2(16)-(15+20)}{(10-15)(10-20)}(227.04) + \frac{2(16)-(10+20)}{(15-10)(15-20)}(362.78) \\ &\quad + \frac{2(16)-(10+15)}{(20-10)(20-15)}(517.35) \\ &= \mathbf{29.784\ m/s^2} \end{aligned}$$

Polynomial differentiation – Fortran example

The Fortran code for this method draws heavily from the code on polynomial interpolation, discussed in Chapter 4. In fact, huge chunks of that program are re-used in this program, and only certain parts have been added and/or edited.

Thus, only these parts will be dealt with in the succeeding discussion. One can examine the code for polynomial interpolation discussed in Chapter 4 to understand the rest of the program.

```
!-----------------------------------------------------------------------------------------
! Ask user if first or second derivative will be computed
write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
read(*,*) choice
!-----------------------------------------------------------------------------------------

!-----------------------------------------------------------------------------------------
! Compute for derivatives
if(choice == 1) then
       do i = 1, n
              total = total + coeff(i) * (i - 1) * point ** (i - 2)
       end do
else if(choice == 2) then
       do i = 1, n
              total = total + coeff(i) * (i - 1) * (i - 2) * point ** (i - 3)
       end do
else
       write(*,*) "Try again. Please enter either 1 or 2."
end if

write(*,"(a57, 1x, f20.10)") "The value of the derivative at your chosen point is: ", total
!-----------------------------------------------------------------------------------------
```

This is the only part of the `Code_5_polynomial.f90` program that had significantly been edited. The user is of course asked whether he would like to compute the first or the second derivative, after which formulas (5.16) and (5.17) are applied to yield the first and second derivatives, respectively. Note that the formula and the code implementation differ by 1. This is because in the formula, n runs from 0 to $n-1$, while in the code, the counter `i` runs from 0 to n. The user is also prompted if (s)he enters an invalid number. The program terminates, and the user would have to start all over again if (s)he makes an invalid choice.

The full program is as follows:

```
!=========================================================================================
! Code_8_polynomial_deriv.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the derivative at a given point using the polynomial method.
!=========================================================================================
program polynomial_deriv
```

```
implicit none
!-------------------------------------------------------------------------------
! Declare variables
integer :: n, i, j, choice
real*16 :: point, total = 0.0
real*16, dimension(:,:), allocatable :: a
real*16, dimension(:), allocatable :: x, y, coeff
character :: confirmed
!-------------------------------------------------------------------------------

write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
read(*,*) n

!-------------------------------------------------------------------------------
! Allocate n
allocate(x(n))
allocate(y(n))
allocate(coeff(n))
allocate(a(n,n))
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! User input
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your INDEPENDENT variables."

       do i = 1, n
              read(*,*) x(i)
       end do

       write(*,*) "Your list of independent variable values is:"
       do i = 1, n
              write(*,"(f10.5)") x(i)
       end do

       write(*,*) "Please confirm. Y/N"
       read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
       write(*,*)
       write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
       write(*,*) "These are the values of your DEPENDENT variables."
```

```
        do i = 1, n
                read(*,*) y(i)
            end do

        write(*,*) "Your list of dependent variable values is:"
        write(*,"(a10, 2x, a10)") "x", "y"
        do i = 1, n
        write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
        end do

write(*,"(a52, 1x)", advance = "no") "At what point would you like to find the
derivative?"
read(*,*) point
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Fill matrix a
do i = 1, n
        do j = 1, n
                a(i, 1) = 1
                a(i, j) = x(i) ** (j - 1)
        end do
end do
!-------------------------------------------------------------------------------

coeff = gaussian(n, a, y)

!-------------------------------------------------------------------------------
! Ask user if first or second derivative will be computed
write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
read(*,*) choice
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Compute for derivatives
if(choice == 1) then
        do i = 1, n
                total = total + coeff(i) * (i - 1) * point ** (i - 2)
        end do
else if(choice == 2) then
        do i = 1, n
                total = total + coeff(i) * (i - 1) * (i - 2) * point ** (i - 3)
        end do
else
```

```
            write(*,*) "Try again. Please enter either 1 or 2."
        end if

        write(*,"(a57, 1x, f20.10)") "The value of the derivative at your chosen point is:
        ", total
        !-------------------------------------------------------------------------------

        contains
            function gaussian(n, a, b) result(x)
                !-------------------------------------------------------------------
                ! Declare variables
                integer :: n, i, j, k
                real*16 :: total
                real*16, dimension(n, n) :: a
                real*16, dimension(n) :: b, x
                !-------------------------------------------------------------------

                ! Implement Naive Gaussian algorithm

                !-------------------------------------------------------------------
                ! Forward elimination
                    do k = 1, n - 1
                        do i = k + 1, n
                            a(i, k) = a(i, k) / a(k, k)

                            do j = k + 1, n
                                a(i, j) = a(i, j) - a(i, k) * a(k, j)
                            end do

                            b(i) = b(i) - a(i, k) * b(k)
                        end do
                    end do
                !-------------------------------------------------------------------

                !-------------------------------------------------------------------
                ! Back substitution
                    x(n) = b(n) / a(n, n)

                    do i = n - 1, 1, -1
                        total = b(i)
                        do j = i + 1, n
                            total = total - a(i, j) * x(j)
                        end do

                        x(i) = total / a(i, i)
                    end do
                !-------------------------------------------------------------------
            end function gaussian

end program polynomial_deriv
```

Executing the program,

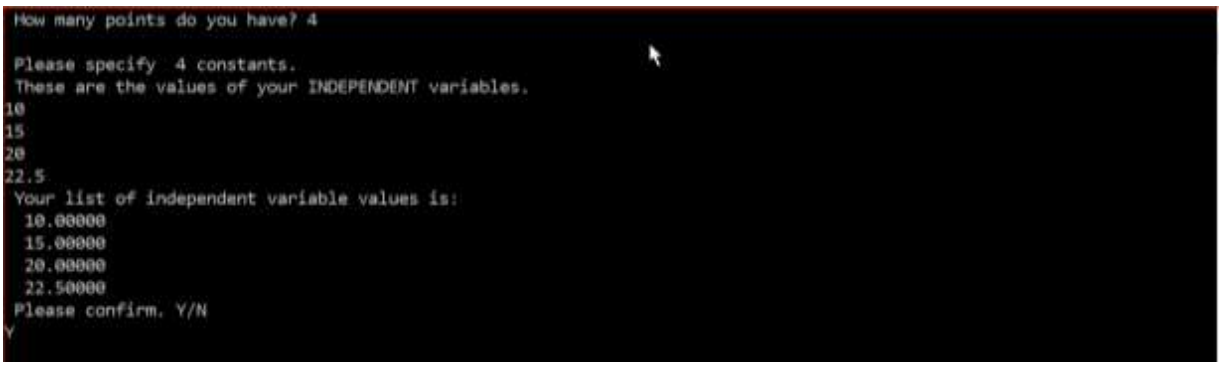

Figure 5.2. *User specifying the number of points and the x-values*

Figure 5.3. *User specifying the y-values*

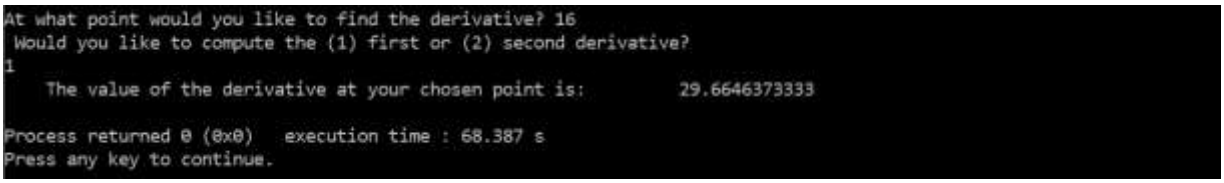

Figure 5.4. *User specifying the point for derivative evaluation, output*

we find that the solution to Example 2(a) is given by

$$a(16) = \mathbf{29.66464\ m/s^2}$$

Lagrange differentiation – Fortran example

The Fortran code for this method draws heavily from the code on Lagrange interpolation, discussed in Chapter 4. In fact, huge chunks of that program are re-used in this program, and only certain parts have been added and/or edited.

Thus, only these parts will be dealt with in the succeeding discussion. One can examine the code for polynomial interpolation discussed in Chapter 4 to understand the rest of the program.

```
!-----------------------------------------------------------------------------------------
! Ask user if first or second derivative will be computed
write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
read(*,*) choice
!-----------------------------------------------------------------------------------------

!-----------------------------------------------------------------------------------------
! Implement Lagrange differentiation
if(choice == 1) then
       do i = 1, n
              l_i = 0

              do j = 1, n
                     if(j /= i) then

                            num = 1
                            do k = 1, n
                                   if((k /= i) .and. (k /= j)) then
                                          num = num * (point - x(k))
                                   end if
                            end do

                            den = 1
                            do k = 1, n
                                   if(k /= i) then
                                          den = den * (x(i) - x(k))
                                   end if
                            end do

                            l_i = l_i + (num / den)
                     end if
              end do
              total = total + y(i) * l_i
       end do
```

```
else if(choice == 2) then
       do i = 1, n
              l_i = 0

              do j = 1, n
                     l_j = 0

                     do k = 1, n
                            if((k /= i) .and. (k /= j)) then
                                   num = 1
                                   do l = 1, n
                                          if((l /= i) .and. (l /= j) .and. (l /= k)) then
                                                 num = num * (point - x(l))
                                          end if
                                   end do

                                   den = 1
                                   do l = 1, n
                                          if((l /= i) .and. (l /= j)) then
                                                 den = den * (x(i) - x(l))
                                          end if
                                   end do

                                   l_j = l_j + (num / den)
                            end if
                     end do

                     if(j /= i) then
                            l_i = l_i + l_j / (x(i) - x(j))
                     end if
              end do

              total = total + y(i) * l_i
       end do

else
       write(*,*) "Try again. Please enter either 1 or 2."

end if

write(*,"(a57, 1x, f20.10)") "The value of the derivative at your chosen point is: ", total
!-------------------------------------------------------------------------------------------
```

This is the only part of the Code_6_lagrange.f90 program that had significantly been edited. The user is of course asked whether he would like to compute the first or the second derivative, after which formulas (5.21) and (5.23) are applied to yield the first and second derivatives, respectively. The user is also prompted if (s)he enters an invalid number. The program terminates, and the user would have to start all over again if (s)he makes an invalid choice.

The full program is as follows:

```
!=============================================================================================
! Code_9_lagrange_deriv.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the derivative at a given point using the Lagrange
! polynomial.
!=============================================================================================

program lagrange_deriv

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	integer :: n, i, j, k, l, choice
	real*16 :: point, num = 1.0, den = 1.0, total = 0.0, l_i, l_j
	real*16, dimension(:), allocatable :: x, y
	character :: confirmed
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Ask user to specify the number of data points
	write(*,"(a29, 1x)", advance = "no") "How many points do you have?"
	read(*,*) n
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Allocate n
	allocate(x(n))
	allocate(y(n))
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! Ask user to specify values
	do while(confirmed /= "Y")
		write(*,*)
		write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
		write(*,*) "These are the values of your INDEPENDENT variables."

		do i = 1, n
			read(*,*) x(i)
		end do

		write(*,*) "Your list of independent variable values is:"
		do i = 1, n
			write(*,"(f10.5)") x(i)
		end do
```

```
        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

confirmed = "N"
do while(confirmed /= "Y")
        write(*,*)
        write(*,"(a15, 1x, i2, 1x, a10)") "Please specify", n, "constants."
        write(*,*) "These are the values of your DEPENDENT variables."

        do i = 1, n
                read(*,*) y(i)
        end do

        write(*,*) "Your list of dependent variable values is:"
        write(*,"(a10, 2x, a10)") "x", "y"
        do i = 1, n
                write(*,"(f10.5, 2x, f10.5)") x(i), y(i)
        end do

        write(*,*) "Please confirm. Y/N"
        read(*,*) confirmed
end do

write(*,"(a52, 1x)", advance = "no") "At what point would you like to find the
derivative?"
read(*,*) point
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Ask user if first or second derivative will be computed
write(*,*) "Would you like to compute the (1) first or (2) second derivative?"
read(*,*) choice
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Implement Lagrange differentiation
if(choice == 1) then
        do i = 1, n
                l_i = 0

                do j = 1, n
                        if(j /= i) then

                                num = 1
                                        do k = 1, n
                                                if((k /= i) .and. (k /= j)) then
                                                        num = num * (point - x(k))
                                                end if
```

```
                        end do

                        den = 1
                        do k = 1, n
                               if(k /= i) then
                                      den = den * (x(i) - x(k))
                               end if
                        end do

                        l_i = l_i + (num / den)
                    end if
                end do
                total = total + y(i) * l_i
            end do

        else if(choice == 2) then
            do i = 1, n
                l_i = 0

                do j = 1, n
                    l_j = 0

                    do k = 1, n
                        if((k /= i) .and. (k /= j)) then
                            num = 1
                            do l = 1, n
                                if((l /= i) .and. (l /= j) .and. (l /= k)) then
                                    num = num * (point - x(l))
                                end if
                            end do

                            den = 1
                            do l = 1, n
                                if((l /= i) .and. (l /= j)) then
                                    den = den * (x(i) - x(l))
                                end if
                            end do

                            l_j = l_j + (num / den)

                        end if
                    end do

                    if(j /= i) then
                        l_i = l_i + l_j / (x(i) - x(j))
                    end if
                end do

                total = total + y(i) * l i
```

```
            end do

        else
            write(*,*) "Try again. Please enter either 1 or 2."

        end if

        write(*,"(a57, 1x, f20.10)") "The value of the derivative at your chosen point
        is: ", total
        !------------------------------------------------------------------------
end program lagrange_deriv
```

Executing the program,

```
How many points do you have? 3

 Please specify  3 constants.
 These are the values of your INDEPENDENT variables.
10
15
20
 Your list of independent variable values is:
  10.00000
  15.00000
  20.00000
 Please confirm. Y/N
Y
```

Figure 5.5. *User specifying the number of points and the x-values*

```
 Please specify  3 constants.
 These are the values of your INDEPENDENT variables.
10
15
20
 Your list of independent variable values is:
  10.00000
  15.00000
  20.00000
 Please confirm. Y/N
Y

 Please specify  3 constants.
 These are the values of your DEPENDENT variables.
227.04
362.78
517.35
 Your list of dependent variable values is:
         x             y
  10.00000   227.04000
  15.00000   362.78000
  20.00000   517.35000
 Please confirm. Y/N
Y
```

Figure 5.6. *User specifying the y-values*

```
At what point would you like to find the derivative? 16
 Would you like to compute the (1) first or (2) second derivative?
1
    The value of the derivative at your chosen point is:         29.7842000000

Process returned 0 (0x0)   execution time : 120.240 s
Press any key to continue.
```

Figure 5.7. *User specifying the point for derivative evaluation, output*

we find that the solution to Example 2(b) is given by

$$a(16) = \mathbf{29.78420\ m/s^2}$$

Application – Projectile motion using Euler's Method

The succeeding discussion is adapted from Giordano and Nakanishi (2006).

An important consequence of the forward difference approximation for the derivative discussed earlier in this chapter is Euler's method, which is commonly used to solve differential equations. Differential equations are prevalent in physics and Euler's method was one of the first ways by which physicists and mathematicians alike numerically solved differential equations which could not be solved analytically.

From (5.2), the formula for the forward difference approximation for the derivative is

$$f'(t) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

In physics, the independent variable is normally time. Thus, we can recast (5.2) as

$$f'(t) \approx \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{5.24}$$

If we wanted to know the value of the function after the passage of a certain amount of time Δt, we can simply solve for it in (5.24). In doing so, we get

$$f(t + \Delta t) \approx f(t) + f'(t)\Delta t \tag{5.25}$$

(5.25) is the basis for Euler's method. To show how this is normally applied, we will use a common example in kinematics – projectile motion.

Using the common assumptions made in basic physics (no air resistance, no acceleration along the x-axis), projectile motion is commonly defined by the succeeding equations, which follow from Newton's second law of motion

$$\frac{d^2x}{dt^2} = 0 \tag{5.26}$$

$$\frac{d^2y}{dt^2} = -g \tag{5.27}$$

To apply Euler's method, however, we need first-order differential equations. Hence, we must rewrite (5.26) and (5.27) in the following manner

$$\frac{dx}{dt} = v_x$$

$$\frac{dv_x}{dt} = 0 \tag{5.28}$$

$$\frac{dy}{dt} = v_y$$

$$\frac{dv_y}{dt} = -g \tag{5.29}$$

We can even take the drag force of air F_{drag} into account. If we consider drag to be proportional to the square of the velocity, then

$$F_{drag} = -Bv^2 \tag{5.30}$$

where B is the drag coefficient.

If we separate the drag force into its constituent x and y components,

$$\begin{aligned} F_{drag_x} &= -Bv^2 cos\theta \\ &= -Bv^2 \left(\frac{v_x}{v}\right) \\ &= -Bvv_x \end{aligned} \tag{5.31}$$

By the same logic,

$$F_{drag_y} = -Bvv_y \tag{5.32}$$

Plugging these into (5.28) and (5.29), we get

$$\begin{aligned} \frac{dx}{dt} &= v_x \\ \frac{dv_x}{dt} &= -\frac{Bvv_x}{m} \end{aligned} \tag{5.33}$$

$$\begin{aligned} \frac{dy}{dt} &= v_y \\ \frac{dv_y}{dt} &= -g - \frac{Bvv_y}{m} \end{aligned} \tag{5.34}$$

Applying (5.25) to (5.33) and (5.34),

$$\begin{aligned} x_{i+1} &= x_i + v_{x_i}\,\Delta t \\ v_{x\,i+1} &= v_{x_i} - \frac{Bvv_{x_i}}{m}\,\Delta t \\ y_{i+1} &= y_i + v_{y_i}\,\Delta t \\ v_{y_{i+1}} &= v_{y_i} - \left(g + \frac{Bvv_{y_i}}{m}\right)\Delta t \end{aligned} \tag{5.35}$$

The set of equations in (5.35) can now be used to calculate the position of the projectile at a given period of time, which can then be graphed.

One can use gnuplot to graph the trajectory of the projectile, but in our case, we will simply be exporting the output to a .dat file, which can then be saved as a .csv file, and then opened and graphed in Excel.

The full program is as follows:

NOTE BEFORE RUNNING: The program generates a plot.dat file every time it runs. Delete the old plot.dat file every time you run the program again. Also note that the array size is finite and the program will throw an error if it goes out of range.

```
!=============================================================================================
! Code_10_projectile.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program plots the trajectory of a projectile computed using Euler's method.
! Assumptions:
! - projectile is thrown from the ground
! - projectile is thrown with initial velocity specified by user
! - air resistance can be present and drag coefficient is specified by user
!=============================================================================================
program projectile
       implicit none
       !-------------------------------------------------------------------------------------
       ! Declare variables
       integer :: i = 1
       real*16 :: drag, step, v_0, theta, mass, PI, g = 9.8
       real*16, dimension(5000) :: time_array, x_array, y_array, vx_array, vy_array
       !-------------------------------------------------------------------------------------

       !-------------------------------------------------------------------------------------
       ! User input
       write(*,*) "Please specify the following values: "

       write(*,"(a28, 1x)", advance = "no") "initial velocity (in m/s) = "
       read(*,*) v_0

       write(*,"(a30, 1x)", advance = "no") "angle of throw (in degrees) = "
       read(*,*) theta

       ! Convert to radians
       PI = 4 * atan(1.0)
       theta = theta / 180 * PI

       write(*,"(a28, 1x)", advance = "no") "drag coefficient (in 1/m) = "
       read(*,*) drag

       write(*,"(a29, 1x)", advance = "no") "mass of projectile (in kg) = "
       read(*,*) mass

       write(*,"(a19, 1x)", advance = "no") "time step (in s) = "
       read(*,*) step
       !-------------------------------------------------------------------------------------
```

```
        !-------------------------------------------------------------------------------
        ! Initialize arrays
        time_array(1) = 0
        x_array(1) = 0
        y_array(1) = 0
        vx_array(1) = v_0 * cos(theta)
        vy_array(1) = v_0 * sin(theta)
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! Implement Euler's method, write to .dat file
        open(1, file = "plot.dat", status = "new")
        write(1,*) "time,x,y"
        write(1,*) ""
        write(1,*) time_array(i), ",", x_array(i), ",", y_array(i)

        do while(0 == 0)
                time_array(i + 1) = time_array(i) + step

                x_array(i + 1) = x_array(i) + vx_array(i) * step
                y_array(i + 1) = y_array(i) + vy_array(i) * step

                vx_array(i + 1) = vx_array(i) - (drag * v_0 * vx_array(i) / mass) * step
                vy_array(i + 1) = vy_array(i) - (g + (drag * v_0 * vy_array(i) / mass)) *
                step

                write(1,*) time_array(i + 1), ",", x_array(i + 1), ",", y_array(i + 1)

                if(y_array(i + 1) <= 0) then
                        stop
                end if

                i = i + 1
        end do

        close(1)
        !-------------------------------------------------------------------------------
end program projectile
```

Executing the program,

```
 Please specify the following values:
initial velocity (in m/s) =  700
angle of throw (in degrees) =  55
drag coefficient (in 1/m) =  0.00004
mass of projectile (in kg) =  1
time step (in s) =  0.1
```

Figure 5.8. *Specifications for* *projectile.f90*

Giordano and Nakanishi (2006) model a large cannon shell with an initial velocity of 700 m/s and a drag coefficient-to-mass ratio of $4 \times 10^{-5}\ \mathrm{m}^{-1}$, the same specifications inputted in the program per Figure 5.8. The program was slightly modified to give the trajectory with and without air resistance, so that they can be compared in one plot.

Once the program finishes executing, it will generate a file called plot.dat. You can change its file extension to .csv by renaming it or save it as a .csv file (if you want to keep a copy of the .dat file) so that it can be opened and graphed in Excel.

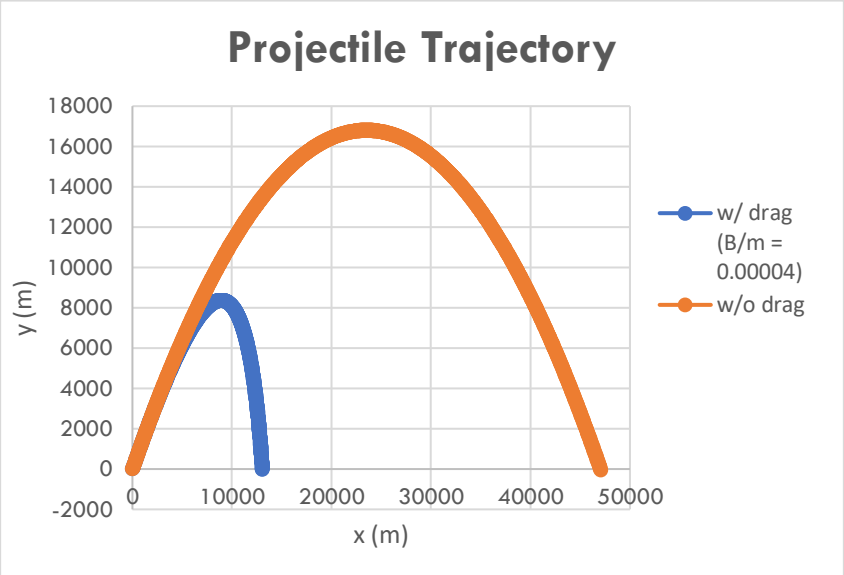

Figure 5.9. *Trajectory with and without drag*

Figure 5.9 shows the trajectory of the projectile with and without air resistance. As shown in the graph, the range and the maximum height are much smaller when there is air resistance, particularly because drag force impedes the motion of the projectile. Notice that in both cases, the projectile doesn't technically land but overshoots the x-axis. This is because the y value never reaches zero for the given time step. To get the exact time at which the projectile lands, one would have to interpolate the value from the time it is closest to the ground and the time it would have been "below the ground."

CHAPTER 6: *SOLUTIONS OF NONLINEAR EQUATIONS*

Introduction

Equations in physics are hardly ever simple. We can often make simplifying assumptions that would reduce these complicated equations into a tractable linear form but often, making such assumptions would not be wise. Thus, as a budding physicist, one must know how to solve nonlinear equations numerically, if they cannot otherwise be solved algebraically.

Note that we have a *linear equation* in our hands if the equation can be recast so that it takes the form

$$y = mx + b \tag{6.1}$$

where m is the slope and b is the y-intercept. If the equation cannot possibly be recast into (6.1), we say that it is *nonlinear*. Often, equations that cannot be solved algebraically are nonlinear in form, and so the methods outlined in this chapter are often billed as those that solve nonlinear equations.

In particular, we will be focusing on the *root-finding problem*. We can always recast our nonlinear equation so that the right-hand side is zero. Our problem then is now finding the *zeroes or roots* of the nonlinear equation (i.e., the values where $f(x)$ is zero).

In this chapter, we will discuss several methods of doing exactly that. First, we will focus on the exact solution of *quadratic* (degree two) and *cubic* (degree three) equations, which normally occur not only in physics but in other fields as well. We will then discuss the methods one can use to numerically solve nonlinear equations, namely (1) bisection method, (2) Newton-Raphson method and (3) secant method.

Solving quadratic equations

A *quadratic equation* is an equation of order two, normally of the form

$$ax^2 + bx + c = 0 \text{ , where } a \neq 0 \tag{6.2}$$

To derive the solution of a quadratic equation in general, we do the following:

We first factor out a

$$x^2 + \frac{b}{a}x + \frac{c}{a} = 0$$

and complete the square.

$$x^2 + \frac{b}{a}x + \frac{c}{a} + \left(\frac{b^2}{4a^2} - \frac{b^2}{4a^2}\right) = 0$$

Simplifying,

$$x^2 + \frac{b}{a}x + \frac{b^2}{4a^2} + \frac{4ac - b^2}{4a^2} = 0$$

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

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

$$x=\frac{\boldsymbol{-b\pm\sqrt{b^2-4ac}}}{\boldsymbol{2a}} \tag{6.3}$$

which is the quadratic formula. From (6.3), one can see that the quadratic equation has two roots and the nature of its roots depend on b^2-4ac (otherwise known as the *discriminant*). Note that if:

- $b^2-4ac<0$, both roots will be complex
- $b^2-4ac>0$, both roots will be real
- $b^2-4ac=0$, the roots will be real and repeated

Example 1

An object is thrown from the top of a building at a speed of 10 m/s. The building it is thrown from is 300 m tall. Find the time it takes for the ball to reach the ground.

Solution

We know that the acceleration of a freely-falling body is given by

$$a=-g \tag{6.4}$$

We can integrate this once to get the velocity (integrate from v_0 to v)

$$v=v_0-gt$$

and another time to get the distance (integrate from s_0 to s)

$$s=s_0+v_0t-\frac{1}{2}gt^2 \tag{6.5}$$

where $s_0=\text{initial distance (m)}$

$v_0=\text{initial velocity}\left(\frac{\text{m}}{s}\right)$

$g=\text{acceleration due to gravity}\left(\frac{\text{m}}{s^2}\right)$

If we set $s_0=0$ at the bottom of the building and substitute the remaining values to (6.5), we get

$$s=-4.9t^2+10t+300 \tag{6.6}$$

We solve (6.6) using (6.3), yielding

$$t=\frac{-10\pm\sqrt{(10)^2-4(-4.9)(300)}}{2(-4.9)}$$

$$t = \frac{-10 \pm 77.3305}{-9.8}$$

$$t = \mathbf{8.9113\ s}, \mathbf{-6.8705\ s}$$

Since a negative value for time makes no sense, we choose $t = \mathbf{8.9113\ s}$

Thus, it takes 8.9113 s for an object dropped in these conditions to reach the ground.

Solving quadratic equations – Fortran example

Implementing (6.3) in Fortran is fairly straightforward. As such, there is no need to diagrammatically discuss the algorithm using a flowchart and we can jump straight to the code.

```
!=============================================================================================
! Code_11_quadratic.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a quadratic equation of the form a*x^2 + b*x + c = 0
!=============================================================================================

program quadratic

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	complex :: a, b, c, r1, r2
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "The quadratic equation is of the form a*x^2 + b*x + c"
	write(*,*) "Please specify your values for a, b and c in the form (real part,
	complex part)"
	write(*,*) "For example, (2,3) for 2 + 3*i"
	write(*, "(a4)", advance = "no") "a = "
	read(*,*) a
	write(*, "(a4)", advance = "no") "b = "
	read(*,*) b
	write(*, "(a4)", advance = "no") "c = "
	read(*,*) c
	write(*,*)
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`a`, `b` and `c` store the coefficients of the quadratic equation of the form $ax^2 + bx + c = 0$ while `r1` and `r2` store the roots. All the variables are of the `complex` type to allow for coefficients and roots that are complex.

The user is then prompted to specify the values for `a`, `b` and `c`.

```
	!---------------------------------------------------------------------------------------
	! Apply the quadratic formula
	r1 = (-b + sqrt(b ** 2 - 4.0 * a * c))/(2.0 *a)
	r2 = (-b - sqrt(b ** 2 - 4.0 * a * c))/(2.0 *a)
	!---------------------------------------------------------------------------------------
```

```
!-----------------------------------------------------------------------------------------
! Print out results
write(*,*) "The roots of the above quadratic equation are: "

if(imag(r1) < 0) then
       write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r1), "-", -imag(r1), "*i"
else
       write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r1), "+", imag(r1), "*i"
end if

if(imag(r2) < 0) then
       write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r2), "-", -imag(r2), "*i"
else
        write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r2), "+", imag(r2), "*i"
end if
!-----------------------------------------------------------------------------------------
```

The quadratic formula in (6.3) is then used to calculate roots `r1` and `r2`. The roots are then printed afterwards and formatted so that they are presented properly.

The full program is as follows:

```
!=========================================================================================
! Code_11_quadratic.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a quadratic equation of the form a*x^2 + b*x + c = 0
!=========================================================================================

program quadratic

       implicit none
       !-----------------------------------------------------------------------------------
       ! Declare variables
       complex :: a, b, c, r1, r2
       !-----------------------------------------------------------------------------------

       !-----------------------------------------------------------------------------------
       ! User input
       write(*,*) "The quadratic equation is of the form a*x^2 + b*x + c"
       write(*,*) "Please specify your values for a, b and c in the form (real part,
       complex part)"
       write(*,*) "For example, (2,3) for 2 + 3*i"
       write(*, "(a4)", advance = "no") "a = "
       read(*,*) a
```

```
        write(*, "(a4)", advance = "no") "b = "
        read(*,*) b
        write(*, "(a4)", advance = "no") "c = "
        read(*,*) c
        write(*,*)
        !------------------------------------------------------------------------------

        !------------------------------------------------------------------------------
        ! Apply the quadratic formula
        r1 = (-b + sqrt(b ** 2 - 4.0 * a * c))/(2.0 *a)
        r2 = (-b - sqrt(b ** 2 - 4.0 * a * c))/(2.0 *a)
        !------------------------------------------------------------------------------

        !------------------------------------------------------------------------------
        ! Print out results
        write(*,*) "The roots of the above quadratic equation are: "

        if(imag(r1) < 0) then
               write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r1), "-", -imag(r1), "*i"
        else
               write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r1), "+", imag(r1), "*i"
        end if

        if(imag(r2) < 0) then
               write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r2), "-", -imag(r2), "*i"
        else
                write(*,"(f10.5, 1x, a1, 1x, f10.5, 1x, a2)") real(r2), "+", imag(r2), "*i"
        end if
        !------------------------------------------------------------------------------
end program quadratic
```

Executing the program,

```
 The quadratic equation is of the form a*x^2 + b*x + c
 Please specify your values for a, b and c in the form (real part, complex part)
 For example, (2,3) for 2 + 3*i
a = (-4.9, 0)
b = (10, 0)
c = (300, 0)

 The roots of the above quadratic equation are:
  -6.87045 +   -0.00000 *i
   8.91127 +    0.00000 *i

Process returned 0 (0x0)   execution time : 152.527 s
Press any key to continue.
```

Figure 6.8. *Solving quadratic equations using* *Code_11_quadratic.f90*

we find that the solution to (6.6) is given by

$$t = \mathbf{8.91127}\ \boldsymbol{m/s}$$

Solving cubic equations

A *quadratic equation* is an equation of order two, normally of the form

$$ax^3 + bx^2 + cx + d = 0 \text{ , where } a \neq 0 \tag{6.7}$$

To derive the solution of a cubic equation in general, we do the following:

We first make the substitution

$$x = y - \frac{b}{3a} \tag{6.8}$$

to get

$$a\left(y - \frac{b}{3a}\right)^3 + b\left(y - \frac{b}{3a}\right)^2 + c\left(y - \frac{b}{3a}\right) + d = 0 \tag{6.9}$$

Using the identity $(x \pm y)^3 = x^3 \pm 3x^2y + 3xy^2 \pm y^3$, we reduce (6.9) to

$$ay^3 + \left(c - \frac{b^2}{3a}\right)y + \left(d + \frac{2b^3}{27a^2} - \frac{bc}{3a}\right) = 0 \tag{6.10}$$

(6.10) is known as the *depressed cubic* since the quadratic term has vanished. We then factor out a to get

$$y^3 + \frac{1}{a}\left(c - \frac{b^2}{3a}\right)y + \frac{1}{a}\left(d + \frac{2b^3}{27a^2} - \frac{bc}{3a}\right) = 0 \tag{6.11}$$

Let

$$e = \frac{1}{a}\left(c - \frac{b^2}{3a}\right) \tag{6.12}$$

$$f = \frac{1}{a}\left(d + \frac{2b^3}{27a^2} - \frac{bc}{3a}\right) \tag{6.13}$$

so that (6.11) becomes

$$y^3 + ey + f = 0 \tag{6.14}$$

Making *Vieta's substitution*

$$y = z + \frac{s}{z} \tag{6.15}$$

to (6.14) gives us

$$\left(z + \frac{s}{z}\right)^3 + e\left(z + \frac{s}{z}\right) + f = 0 \tag{6.16}$$

which simplifies to

$$z^6 + (3s + e)z^4 + fz^3 + s(3s + e)z^2 + s^3 = 0 \tag{6.17}$$

We let

$$s = -\frac{e}{3} \tag{6.18}$$

to cancel out two of the terms, leaving us with

$$z^6 + fz^3 + s^3 = 0 \tag{6.19}$$

We can now let

$$w = z^3 \tag{6.20}$$

which reduces (6.19) to

$$w^2 + fw + -\frac{e^3}{27} = 0 \tag{6.21}$$

which is a quadratic equation. We can now use the quadratic formula in (6.3) to this equation, which leads to

$$w = -\frac{f}{2} \pm \sqrt{\frac{f^2}{4} + \frac{e^3}{27}} \tag{6.22}$$

After getting two values for w, one can now back-substitute using the previous substitutions made in (6.20), (6.18), (6.15) and (6.8). Using only one of the values for w is fine in this case because both of them should yield the same roots.

Example 2

Solve $x^3 - 9x^2 + 36x - 80 = 0$

Solution

We use (6.12) and (6.13) to find e and f, after which we use (6.22) to find w.

$$e = \frac{1}{a}\left(c - \frac{b^2}{3a}\right)$$
$$= \frac{1}{1}\left(36 - \frac{(-9)^2}{3(1)}\right) = \mathbf{9}$$

$$f = \frac{1}{a}\left(d + \frac{2b^3}{27a^2} - \frac{bc}{3a}\right)$$
$$= \frac{1}{1}\left(-80 + \frac{2(-9)^3}{27(1)^2} - \frac{(-9)(36)}{3(-1)}\right) = \mathbf{-26}$$

Applying (6.22),

$$w = -\frac{f}{2} \pm \sqrt{\frac{f^2}{4} + \frac{e^3}{27}}$$
$$= -\frac{26}{2} \pm \sqrt{\frac{(26)^2}{4} + \frac{(-9)^3}{27}} = \mathbf{27}, \mathbf{-1}$$

We now back-substitute, choosing $w = -1$. Starting with (6.20),

$$z^3 = -1$$
$$z^3 = -1e^{i(0+2\pi n)}$$
$$z = -1e^{i\left(0+\frac{2\pi n}{3}\right)} \text{ for } n = 0, 1, 2$$

Thus,

$$z_1 = -\mathbf{1}\boldsymbol{e}^{\boldsymbol{i}(\mathbf{0})}$$
$$= -1(\cos 0 + i \sin 0)$$
$$= \mathbf{-1}$$

$$z_2 = -\mathbf{1}\boldsymbol{e}^{\boldsymbol{i}\left(\frac{\mathbf{2\pi}}{\mathbf{3}}\right)}$$
$$= -1(\cos\frac{2\pi}{3} + i \sin\frac{2\pi}{3})$$
$$= \frac{\mathbf{1}}{\mathbf{2}} - \frac{\sqrt{\mathbf{3}}}{\mathbf{2}}\boldsymbol{i}$$

$$z_3 = -\mathbf{1}\boldsymbol{e}^{\boldsymbol{i}\left(\frac{\mathbf{4\pi}}{\mathbf{3}}\right)}$$
$$= -1(\cos\frac{4\pi}{3} + i \sin\frac{4\pi}{3})$$
$$= \frac{\mathbf{1}}{\mathbf{2}} + \frac{\sqrt{\mathbf{3}}}{\mathbf{2}}\boldsymbol{i}$$

From (6.18),

$$s = -\frac{e}{3}$$
$$= -\frac{9}{3} = \mathbf{-3}$$

We then proceed with (6.15). At this point, we will switch between the polar and rectangular form of z_1, z_2 and z_3, whichever is more convenient to use.

$$y = z + \frac{s}{z}$$
$$y_1 = z_1 + \frac{s}{z_1}$$
$$= -1 + \frac{-3}{-1} = \mathbf{2}$$

$$y_2 = z_2 + \frac{s}{z_2}$$
$$= \left(\frac{1}{2} - \frac{\sqrt{3}}{2}i\right) + \frac{-3}{-1e^{i\left(\frac{2\pi}{3}\right)}}$$
$$= \left(\frac{1}{2} - \frac{\sqrt{3}}{2}i\right) + 3e^{i\left(-\frac{2\pi}{3}\right)}$$
$$= \left(\frac{1}{2} - \frac{\sqrt{3}}{2}i\right) + \left(-\frac{3}{2} - \frac{3\sqrt{3}}{2}i\right) = \mathbf{-1 - 2\sqrt{3}i}$$

$$y_3 = z_3 + \frac{s}{z_3}$$
$$= \left(\frac{1}{2} + \frac{\sqrt{3}}{2}i\right) + \frac{-3}{-1e^{i\left(\frac{4\pi}{3}\right)}}$$
$$= \left(\frac{1}{2} + \frac{\sqrt{3}}{2}i\right) + 3e^{i\left(-\frac{4\pi}{3}\right)}$$
$$= \left(\frac{1}{2} + \frac{\sqrt{3}}{2}i\right) + \left(-\frac{3}{2} + \frac{3\sqrt{3}}{2}i\right) = \mathbf{-1 + 2\sqrt{3}i}$$

Finally, using (6.8),

$$x = y - \frac{b}{3a}$$

$$\frac{b}{3a} = \frac{-9}{3(1)} = \mathbf{-3}$$

$$x_1 = y_1 + 3$$
$$= 2 + 3 = \mathbf{5}$$

$$x_2 = y_2 + 3$$
$$= -1 - 2\sqrt{3}i + 3 = \mathbf{2 - 2\sqrt{3}i}$$

$$x_3 = y_3 + 3$$
$$= -1 + 2\sqrt{3}i + 3 = \mathbf{2 + 2\sqrt{3}i}$$

Thus, the roots of $x^3 - 9x^2 + 36x - 80 = 0$ are at x = $\mathbf{5}$, x = $\mathbf{2 - 2\sqrt{3}i}$ and x = $\mathbf{2 + 2\sqrt{3}i}$.

Solving cubic equations – Fortran example

Implementing the solution to a cubic equation in Fortran is fairly straightforward. As such, there is no need to diagrammatically discuss the algorithm using a flowchart and we can jump straight to the code.

```
!=============================================================================================
! Code_12_cubic.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a cubic equation of the form a*x^3 + b*x^2 + c*x + d
! = 0
!=============================================================================================

program cubic

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables
	complex :: a, b, c, d, e, f, qr1, qr2, cr1, cr2, cr3, s, y1, y2, y3, x1, x2, x3
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "The cubic equation is of the form a*x^3 + b*x^2 + c*x + d"
	write(*,*) "Please specify your values for a, b, c and d in the form (real part,
	complex part)"
	write(*,*) "For example, (2,3) for 2 + 3*i"
	write(*, "(a4)", advance = "no") "a = "
	read(*,*) a
	write(*, "(a4)", advance = "no") "b = "
	read(*,*) b
	write(*, "(a4)", advance = "no") "c = "
	read(*,*) c
	write(*, "(a4)", advance = "no") "d = "
	read(*,*) d
	write(*,*)
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`a`, `b`, `c` and `d` store the coefficients of the cubic equation of the form $ax^3 + bx^2 + cx + d = 0$. `qr1` and `qr2` (quadratic root) represent w from the previous discussion. `cr1`, `cr2` and `cr3` (cubic root) represent z from the previous discussion. All the other variables are named as they are in the previous discussion.

All variables are of the `complex` type to allow for coefficients and roots that are complex. The user is then prompted to specify the values for `a`, `b`, `c` and `d`.

```
!-------------------------------------------------------------------------------------------
! Calculate e and f
e = (c/a) - (b**2/(3.0*a**2))
f = (d/a) + ((2.0*b**3)/(27.0*a**3)) - ((b*c)/(3.0*a**2))
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! Substitute to quadratic formula
qr1 = -(f/2.0) + sqrt(f ** 2 / 4.0 + e ** 3 / 27.0)
qr2 = -(f/2.0) - sqrt(f ** 2 / 4.0 + e ** 3 / 27.0)
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! Use w = z^3 and get cube roots of one of the quadratic roots.
call cube_root(qr1, cr1, cr2, cr3)
!call cube_root(qr2, cr1, cr2, cr3)
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! Use y = z + (s/z), where s = -e/3, to get y.
s = -e/3.0
y1 = cr1 + (s/cr1)
y2 = cr2 + (s/cr2)
y3 = cr3 + (s/cr3)
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! Use x = y - (b/3*a) to get x.
x1 = y1 - (b/(3.0*a))
x2 = y2 - (b/(3.0*a))
x3 = y3 - (b/(3.0*a))
!-------------------------------------------------------------------------------------------
```

The code snippet above follows the algorithm outlined in the example to the letter and is fairly straightforward. The subroutine `cube_root` (to be discussed later) was used to extract the 3 cubic roots of `qr1`, outputting `cr1`, `cr2` and `cr3`.

```
!-------------------------------------------------------------------------------------------
! Print out results
write(*,*) "The roots of the above cubic equation are: "

if(imag(x1) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x1), "-", -imag(x1), "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x1), "+", imag(x1), "*i"
end if
```

```
if(imag(x2) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x2), "-", -imag(x2), "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x2), "+", imag(x2), "*i"
end if

if(imag(x3) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x3), "-", -imag(x3), "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x3), "+", imag(x3), "*i"
end if
!-------------------------------------------------------------------------------------
```

As with quadratic.f90, the roots are printed afterwards and formatted so that they are presented properly.

```
contains
!=====================================================================================
! This subroutine finds the cube root of a complex number z and outputs the roots
! z1, z2 and z3.
!=====================================================================================

       subroutine cube_root(z, z1, z2, z3)
              implicit none
              !-------------------------------------------------------------------------
              ! Declare variables
              complex :: z, z1, z2, z3
              real*16 :: r, theta, r_cr, theta_cr1, theta_cr2, theta_cr3, PI

              PI = 4*atan(1.0)
              !-------------------------------------------------------------------------

              !-------------------------------------------------------------------------
              ! Express real(z) + imag(z) * i as r*e^(i*theta)
              ! Find r and theta
              r = sqrt(real(z) ** 2 + imag(z) ** 2)
              theta = atan2(imag(z), real(z))
              !-------------------------------------------------------------------------

              !-------------------------------------------------------------------------
              ! r*e^(i*theta) = r*e^(i * theta + 2*PI)
              ! Therefore, cube root of z = r^(1/3) * e^(i * (theta + 2*PI*j)/3) for j = 0,
              1, 2
              r_cr = r ** (1.0/3.0)
              theta_cr1 = theta / 3
              theta_cr2 = (theta + 2*PI)/3
```

```
        theta_cr3 = (theta + 4*PI)/3
        !-------------------------------------------------------------------------------

        !-------------------------------------------------------------------------------
        ! Compute for cubic roots
        ! Express them as z = r*cos(theta) + r*sin(theta) * i
        z1 = complex(r_cr * cos(theta_cr1), r_cr * sin(theta_cr1))
        z2 = complex(r_cr * cos(theta_cr2), r_cr * sin(theta_cr2))
        z3 = complex(r_cr * cos(theta_cr3), r_cr * sin(theta_cr3))
        !-------------------------------------------------------------------------------

    end subroutine cube_root
```

The subroutine `cube_root` takes `z` as an input and outputs its 3 cubic roots `z1`, `z2` and `z3`. It first converts `z` to its polar form. Note that `atan2` is used instead of `atan`, because the latter only returns angles between 0 and π.

After this, we get the cube root of `z`. We get the cube root of its modulus `r_cr` by conventional means, but its argument `theta` is expressed as `theta + 2*PI*j` and divided by 3, resulting in `(theta + 2*PI*j)/3` for j = 0, 1, 2. A do loop could have been used here, but it's easier to simply list the values of the three resulting arguments `theta_cr1`, `theta_cr2` and `theta_cr3`.

The modulus `r_cr` and arguments `theta_cr1`, `theta_cr2` and `theta_cr3` are then combined and expressed in rectangular form, after which they are stored in the output variables `z1`, `z2` and `z3`.

The full program is as follows:

```
!================================================================================================
! Code_12_cubic.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a cubic equation of the form a*x^3 + b*x^2 + c*x + d
! = 0
!================================================================================================

program cubic

    implicit none
    !-----------------------------------------------------------------------------------------
    ! Declare variables
    complex :: a, b, c, d, e, f, qr1, qr2, cr1, cr2, cr3, s, y1, y2, y3, x1, x2, x3
    !-----------------------------------------------------------------------------------------

    !-----------------------------------------------------------------------------------------
    ! User input
    write(*,*) "The cubic equation is of the form a*x^3 + b*x^2 + c*x + d"
```

```
write(*,*) "Please specify your values for a, b, c and d in the form (real part,
complex part)"
write(*,*) "For example, (2,3) for 2 + 3*i"
write(*, "(a4)", advance = "no") "a = "
read(*,*) a
write(*, "(a4)", advance = "no") "b = "
read(*,*) b
write(*, "(a4)", advance = "no") "c = "
read(*,*) c
write(*, "(a4)", advance = "no") "d = "
read(*,*) d
write(*,*)
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Calculate e and f
e = (c/a) - (b**2/(3.0*a**2))
f = (d/a) + ((2.0*b**3)/(27.0*a**3)) - ((b*c)/(3.0*a**2))
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Substitute to quadratic formula
qr1 = -(f/2.0) + sqrt(f ** 2 / 4.0 + e ** 3 / 27.0)
qr2 = -(f/2.0) - sqrt(f ** 2 / 4.0 + e ** 3 / 27.0)
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Use w = z^3 and get cube roots of one of the quadratic roots.
call cube_root(qr1, cr1, cr2, cr3)
!call cube_root(qr2, cr1, cr2, cr3)
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Use y = z + (s/z), where s = -e/3, to get y.
s = -e/3.0
y1 = cr1 + (s/cr1)
y2 = cr2 + (s/cr2)
y3 = cr3 + (s/cr3)
!-------------------------------------------------------------------------------

!-------------------------------------------------------------------------------
! Use x = y - (b/3*a) to get x.
x1 = y1 - (b/(3.0*a))
x2 = y2 - (b/(3.0*a))
x3 = y3 - (b/(3.0*a))
!-------------------------------------------------------------------------------
```

```
!-------------------------------------------------------------------------------
! Print out results
write(*,*) "The roots of the above cubic equation are: "

if(imag(x1) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x1), "-", -imag(x1),
       "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x1), "+", imag(x1), "*i"
end if

if(imag(x2) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x2), "-", -imag(x2),
       "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x2), "+", imag(x2), "*i"
end if

if(imag(x3) < 0) then
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x3), "-", -imag(x3),
       "*i"
else
       write(*,"(f15.10, 1x, a1, 1x, f15.10, 1x, a2)") real(x3), "+", imag(x3), "*i"
end if
!-------------------------------------------------------------------------------

contains
!===============================================================================
! This subroutine finds the cube root of a complex number z and outputs the roots
! z1, z2 and z3.
!===============================================================================

       subroutine cube_root(z, z1, z2, z3)
              implicit none
              !-----------------------------------------------------------------
              ! Declare variables
              complex :: z, z1, z2, z3
              real*16 :: r, theta, r_cr, theta_cr1, theta_cr2, theta_cr3, PI

              PI = 4*atan(1.0)
              !-----------------------------------------------------------------

              !-----------------------------------------------------------------
              ! Express real(z) + imag(z) * i as r*e^(i*theta)
              ! Find r and theta
              r = sqrt(real(z) ** 2 + imag(z) ** 2)
```

```
                    theta = atan2(imag(z), real(z))
                    !------------------------------------------------------------------

                    !------------------------------------------------------------------
                    ! r*e^(i*theta) = r*e^(i * theta + 2*PI)
                    ! Therefore, cube root of z = r^(1/3) * e^(i * (theta + 2*PI*j)/3) for
                    j = 0, 1, 2
                    r_cr = r ** (1.0/3.0)
                    theta_cr1 = theta / 3
                    theta_cr2 = (theta + 2*PI)/3
                    theta_cr3 = (theta + 4*PI)/3
                    !------------------------------------------------------------------

                    !------------------------------------------------------------------
                    ! Compute for cubic roots
                    ! Express them as z = r*cos(theta) + r*sin(theta) * i
                    z1 = complex(r_cr * cos(theta_cr1), r_cr * sin(theta_cr1))
                    z2 = complex(r_cr * cos(theta_cr2), r_cr * sin(theta_cr2))
                    z3 = complex(r_cr * cos(theta_cr3), r_cr * sin(theta_cr3))
                    !------------------------------------------------------------------
        end subroutine cube_root
end program cubic
```

Executing the program,

```
 Please specify your values for a, b, c and d in the form (real part, complex part)
 For example, (2,3) for 2 + 3*i
a = (1,0)
b = (-9,0)
c = (36,0)
d = (-80,0)

 The roots of the above cubic equation are:
   5.0000000000 +    0.0000000000 *i
   1.9999997616 +    3.4641015530 *i
   2.0000000000 -    3.4641020298 *i

Process returned 0 (0x0)   execution time : 11.891 s
Press any key to continue.
```

Figure 6.2. *Solving* $x^3 - 9x^2 + 36x - 80 = 0$ *using* *Code_12_cubic.f90*

we find that the solution to Example 2 is given by

$$x_1 = \mathbf{5}$$

$$x_2 = \mathbf{1.9999997616 + 3.4641015530}\boldsymbol{i}$$

$$x_3 = \mathbf{1.9999997616 - 3.4641015530}\boldsymbol{i}$$

Bisection method

The first numerical method we will be discussing in this chapter is the bisection method. This method is an example of a *bracketing method*, where we "bracket" the root by making two guesses, between which we expect to find the root.

To make sure that our guesses are correct, we use the *intermediate value theorem*, which is as follows:

Intermediate Value Theorem

Let f be continuous over a closed interval $[a, b]$ and suppose $f(a) \neq f(b)$. Then for all k between $f(a)$ and $f(b)$, there exists c in (a, b) for which $f(c) = k$.

Thus, we can glean from this theorem that if $f(a)$ and $f(b)$ are of opposite signs (i.e., $f(a)f(b) < 0$), then there exists s ($a < s < b$) for which $f(s) = 0$. Hence, as a corollary:

Corollary (adapted from Kaw (2011))

An equation $f(x) = 0$ where $f(x)$ is a real continuous function has at least one root between a and b if $f(a)f(b) < 0$

This is illustrated in Figure 6.3. However, the converse is not necessarily true and a root can still exist between a and b if $f(a)f(b) > 0$ (as shown in Figure 6.4).

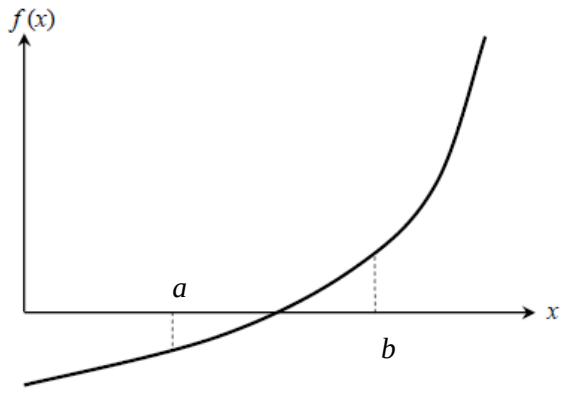

Figure 6.3. *A root exists in [a,b] if $f(a)f(b) < 0$*

(Adapted from Kaw (2011))

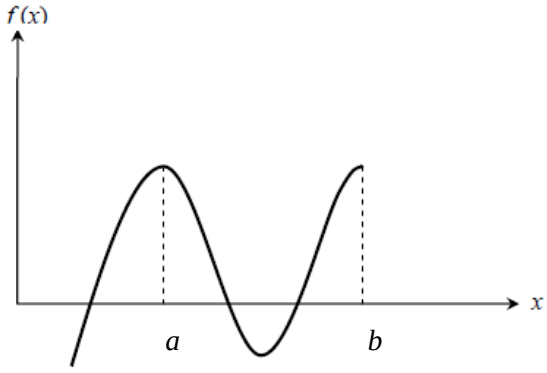

Figure 6.4. *…but the converse is not true!*

(Adapted from Kaw (2011))

Given this information, we can now outline the bisection algorithm from the following steps:

1. Choose two guesses a and b, such that the $f(a)f(b) < 0$.
2. Compute for the midpoint m between the two points, given by

$$m = \frac{a + b}{2} \tag{6.23}$$

3. Use the intermediate value theorem and its corollary to decide which interval to inspect next:

a. If $f(a)f(m) < 0$, then from the intermediate value theorem, we know that the root lies in $[a, m]$. Therefore, $a = a$ and $b = m$.
b. If $f(b)f(m) < 0$, then from the intermediate value theorem, we know that the root lies in $[m, b]$. Therefore, $a = m$ and $b = b$.
c. If $f(a)f(m) < 0$ and $f(b)f(m) < 0$, then we know that m is the root. Therefore, we stop the algorithm.

4. With the new interval, continue steps (2) and (3). For each iteration, find the absolute relative approximate error as discussed in Chapter 2, given by

$$|RAE| = \left|\frac{present - previous}{present}\right| \times 100 \tag{6.24}$$

Set a tolerance and once $|RAE|$ is less than the pre-set tolerance, stop the algorithm. It can happen that the algorithm arrives at the exact root before $|RAE|$ becomes less than the tolerance, so take this into account in the program.

Since this is a mainly numerical method of computing roots, we can jump right into coding the program, since computing this by hand would be unusually tedious.

Bisection method – Fortran example

To implement the bisection method in Fortran, we first need to see the algorithm in a flowchart. The algorithm is summarized in Figure 6.5

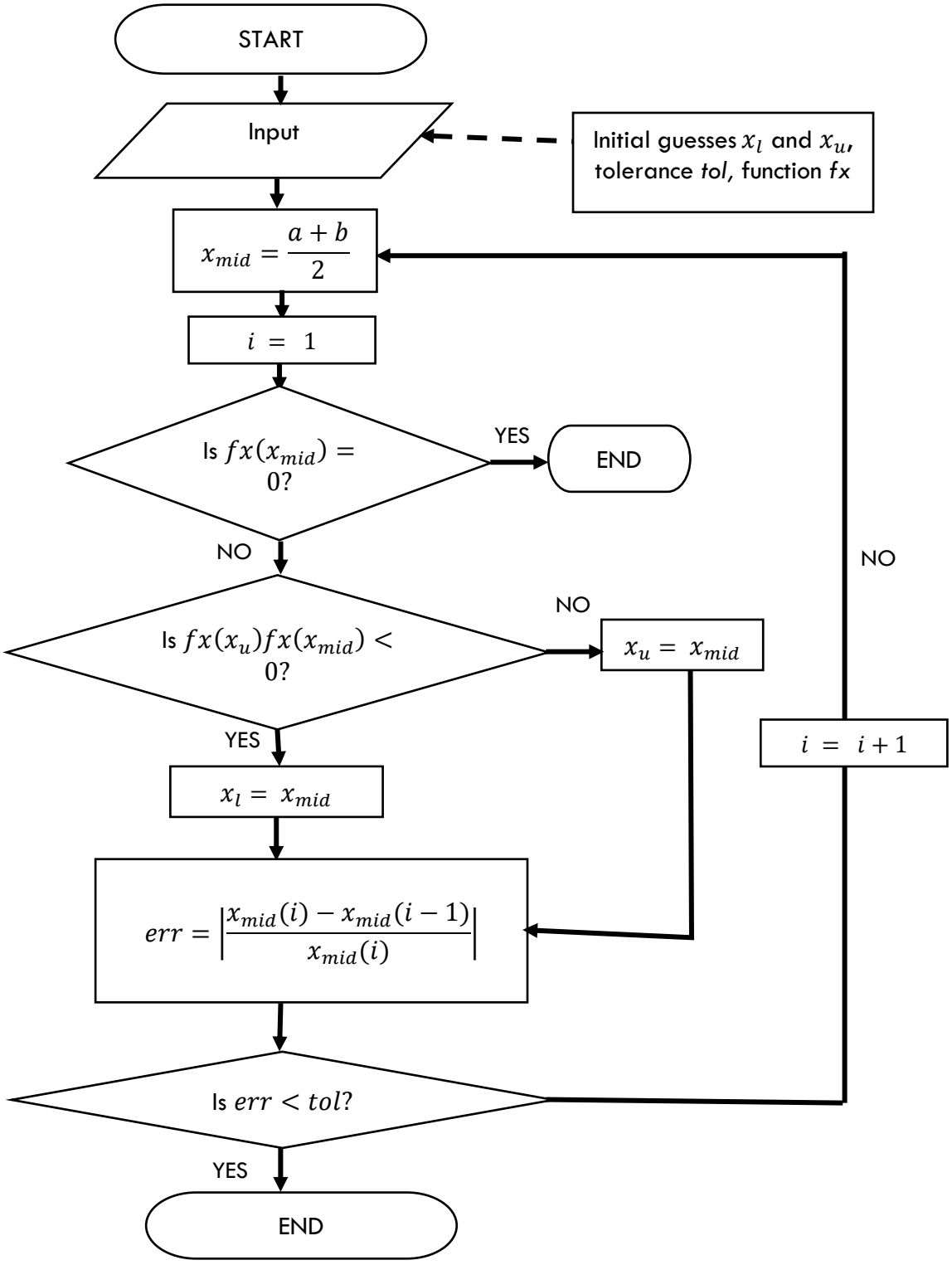

Figure 6.5. *Flowchart for bisection method*

```
!=========================================================================================
! Code_13_bisection.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the bisection method.
!=========================================================================================

program bisection

	implicit none
	!-----------------------------------------------------------------------------------
	! Declare variables
	integer :: i = 1
	real*16 :: x_l = 1.0, x_u = 1.0, x_mid, error = 0.5, tol = 1e-10
	real*16, dimension(100) :: x_mid_array
	!-----------------------------------------------------------------------------------

	!-----------------------------------------------------------------------------------
	! User input
	do while(fx(x_l) * fx(x_u) > 0)
		write(*,*) "Please specify the upper and lower limit of the interval you wish
		to examine."
		write(*, "(a15)", advance = "no") "lower limit = "
		read(*,*) x_l
		write(*, "(a15)", advance = "no") "upper limit = "
		read(*,*) x_u

		! Check if root is at the endpoints.
		if(abs(fx(x_l)) < tol) then
			write(*,*) "The root is at x = ", x_l
			call exit
		else if(abs(fx(x_u)) < tol) then
			write(*,*) "The root is at x = ", x_u
			call exit
		end if

		! Check if there is a root in the interval
		if(fx(x_l) * fx(x_u) > 0) then
			write(*,*) "The root does not exist between these two values. Try
			again"
			write(*,*)
		end if
	end do
	!-----------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`x_l` and `x_u` are the two initial guesses, while `x_mid` refers to the midpoint. `error` stores the RAE per iteration while `tol` stores the pre-specified tolerance, which in this case is 10^{-10}. `i` is used as

a counter, while x_mid_array is used to keep track of all the computed x_mid to make error computations easier.

The user is then prompted to specify the values for x_l and x_u. The program then checks if the root is at any of the endpoints. It also checks if fx(x_l) * fx(x_u) > 0 to see if a root exists at the interval. The user will be notified if (s)he chose a root as one of the endpoints or whether or not a root exists in the interval (s)he chose.

```
!-------------------------------------------------------------------------------------
! This implements the bisection method.
! This also prints out x_l, x_u, x_mid and the RAE per iteration.
write(*,"(a20, 2x, a20, 2x, a20, 2x, a20)") "x_l", "x_u", "x_mid", "error"

do while(error > tol)
! Evaluate midpoint
       x_mid = (x_l + x_u) / 2.0
       x_mid_array(i) = x_mid

       ! Compute error
       if(i > 1) then
              error = abs((x_mid_array(i) - x_mid_array(i - 1))/x_mid_array(i))
       end if

       ! Print results
       if(i == 1) then
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, a20)") x_l, x_u, x_mid, "-"
       else
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, f20.10)") x_l, x_u, x_mid,
              error
       end if

       ! Determine if root is at midpoint
       if(abs(fx(x_mid)) < tol) then
              write(*,*)
              write(*,*) "The root is at x = ", x_mid
              stop
       end if

       ! Evaluate limits
       if(fx(x_u) * fx(x_mid) < 0) then
              x_l = x_mid
       else
              x_u = x_mid
       end if

       ! Update counter
       i = i + 1
end do
!-------------------------------------------------------------------------------------
```

As the comments suggest, this code snippet implements the bisection method and outputs x_l, x_u, x_mid and error per iteration. The code is fairly straightforward and follows the algorithm specified in Figure 6.5.

```
contains
!=============================================================================================
! This function specifies the nonlinear equation to be solved. In this case, it is
! x^3 - 9*x^2 + 36*x - 80. Edit if needed.
!=============================================================================================
       function fx(j) result(fxeval)
              real*16 :: j, fxeval

              fxeval = j**3 - 9*j**2 + 36*j - 80
       end function fx
```

The last snippet contains the function whose roots we are trying to determine using the bisection method. The user can edit the function to his/her needs.

The full program is as follows:

```
!=============================================================================================
! Çode_13_bisection.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the bisection method.
!=============================================================================================

      !---------------------------------------------------------------------------------------
      ! Declare variables
      integer :: i = 1
      real*16 :: x_l = 1.0, x_u = 1.0, x_mid, error = 0.5, tol = 1e-10
      real*16, dimension(100) :: x_mid_array
      !---------------------------------------------------------------------------------------

      !---------------------------------------------------------------------------------------
      ! User input
      do while(fx(x_l) * fx(x_u) > 0)
            write(*,*) "Please specify the upper and lower limit of the interval you wish
            to examine."
            write(*, "(a15)", advance = "no") "lower limit = "
            read(*,*) x_l
            write(*, "(a15)", advance = "no") "upper limit = "
            read(*,*) x_u

            ! Check if root is at the endpoints.
            if(abs(fx(x_l)) < tol) then
                  write(*,*) "The root is at x = ", x_l
                  call exit
            else if(abs(fx(x_u)) < tol) then
                  write(*,*) "The root is at x = ", x_u
                  call exit
            end if

            ! Check if there is a root in the interval
            if(fx(x_l) * fx(x_u) > 0) then
                  write(*,*) "The root does not exist between these two values. Try
                  again"
                  write(*,*)
            end if
      end do
      !---------------------------------------------------------------------------------------
```

```
        !-------------------------------------------------------------------------------
        ! This implements the bisection method.
        ! This also prints out x_l, x_u, x_mid and the RAE per iteration.
        write(*,"(a20, 2x, a20, 2x, a20, 2x, a20)") "x_l", "x_u", "x_mid", "error"

        do while(error > tol .or. abs(fx(x)) > tol)
        ! Evaluate midpoint
               x_mid = (x_l + x_u) / 2.0
               x_mid_array(i) = x_mid

               ! Compute error
               if(i > 1) then
                      error = abs((x_mid_array(i) - x_mid_array(i - 1))/x_mid_array(i))
               end if

               ! Print results
               if(i == 1) then
                      write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, a20)") x_l, x_u, x_mid,
                      "-"
               else
                      write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, f20.10)") x_l, x_u,
                      x_mid, error
               end if

               ! Determine if root is at midpoint
               if(abs(fx(x_mid)) < tol) then
                      write(*,*)
                      write(*,*) "The root is at x = ", x_mid
                      stop
               end if

               ! Evaluate limits
               if(fx(x_u) * fx(x_mid) < 0) then
                      x_l = x_mid
               else
                      x_u = x_mid
        end if

        ! Update counter
        i = i + 1
end do
!-------------------------------------------------------------------------------
```

```
        contains
        !================================================================================
        ! This function specifies the nonlinear equation to be solved. In this case, it is
        ! x^3 - 9*x^2 + 36*x - 80. Edit if needed.
        !================================================================================
               function fx(j) result(fxeval)
                      real*16 :: j, fxeval

                      fxeval = j**3 - 9*j**2 + 36*j - 80
               end function fx
end program bisection
```

Executing the program,

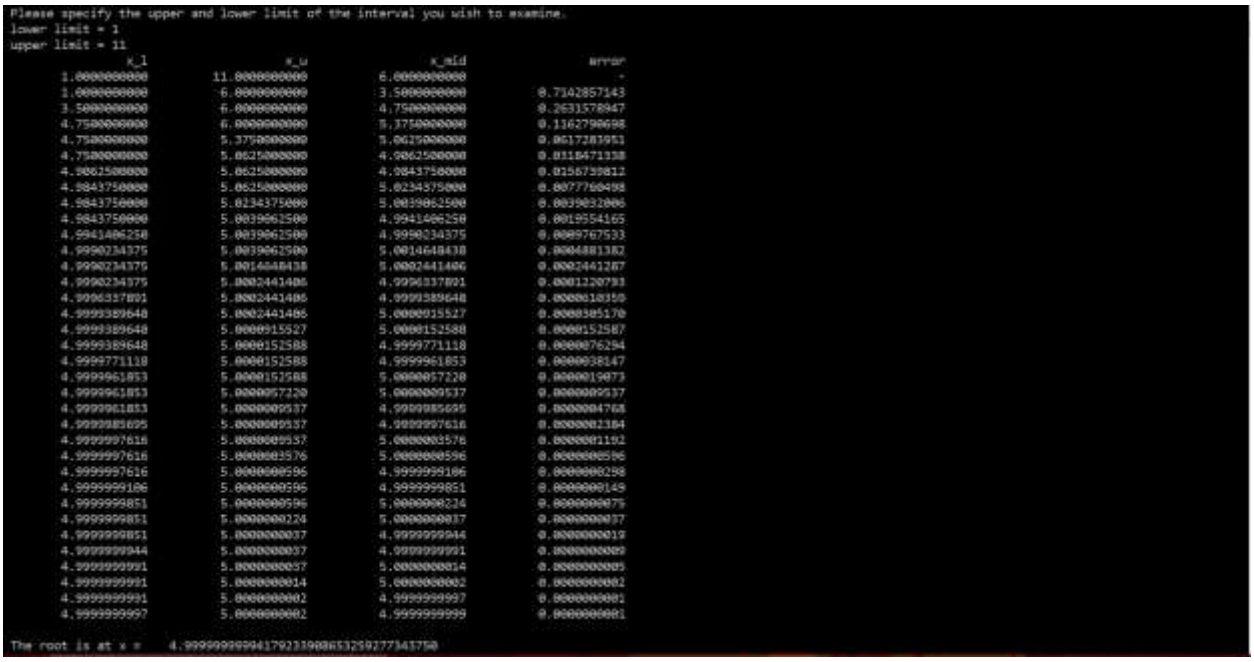

Figure 6.6. *Solving* $x^3 - 9x^2 + 36x - 80 = 0$ *using* *Code_13_bisection.f90*

we find that the solution to $x^3 - 9x^2 + 36x - 80 = 0$ is given by

$$x = \mathbf{4.99999}$$

Newton-Raphson and Secant method

As opposed to being a bracketing method, the Newton-Raphson method is an *open method*, since one only needs to make one *unconditional* initial guess. If this initial guess is at x_0, we can draw a tangent line to f at $(x_0, f(x_0))$ and extend it until it intersects the x-axis. Let x_1 be this intersection point.

We can repeat this process until we obtain an x_n that is reasonably close to the root. We can solve for an expression for x_n, the x-intercept of the line tangent to f at $(x_{n-1}, f(x_{n-1}))$ so that we can solve for the root numerically.

$$f'(x_{n-1}) = \frac{f(x_{n-1}) - 0}{x_{n-1} - x_n}$$

$$x_{n-1} - x_n = \frac{f(x_{n-1})}{f'(x_{n-1})}$$

$$\boldsymbol{x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}} \tag{6.25}$$

One can now use (6.25), known as the Newton-Raphson formula, to compute for the root of f numerically, using the following steps:

1. Solve for f'(x)
2. Choose an initial guess x_n
3. Estimate the new root using (6.25)
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
4. For each iteration, find the absolute relative approximate error using (6.24)

$$|RAE| = \left|\frac{present - previous}{present}\right| \times 100$$

Set a tolerance and once $|RAE|$ is less than the pre-set tolerance, stop the algorithm. It can happen that the algorithm arrives at the exact root before $|RAE|$ becomes less than the tolerance, so take this into account in the program.

However, the Newton-Raphson method assumes that $f'(x)$ exists, which may either be too tedious to compute or may not even exist at the given interval. In this case, we use the *secant method*. Using two initial unconditional guesses, the secant method simply assumes that

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}} \tag{6.26}$$

Carrying on with the rest of the derivation, we end up substituting (6.26) to (6.25), leading to

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}$$

$$x_n = \frac{x_{n-1}[f(x_{n-1}) - f(x_{n-2})] - [f(x_{n-1})x_{n-1} - f(x_{n-1})x_{n-2}]}{f(x_{n-1}) - f(x_{n-2})}$$

$$\boldsymbol{x_n = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}} \tag{6.27}$$

One can now use (6.27) instead of (6.25), using the steps outlined previously.

Having established both methods theoretically, we can now use (6.25) and (6.27) to compute for the roots numerically using Fortran.

Newton-Raphson method – Fortran example

To implement the bisection method in Fortran, we first need to see the algorithm in a flowchart. The algorithm is summarized in Figure 6.7

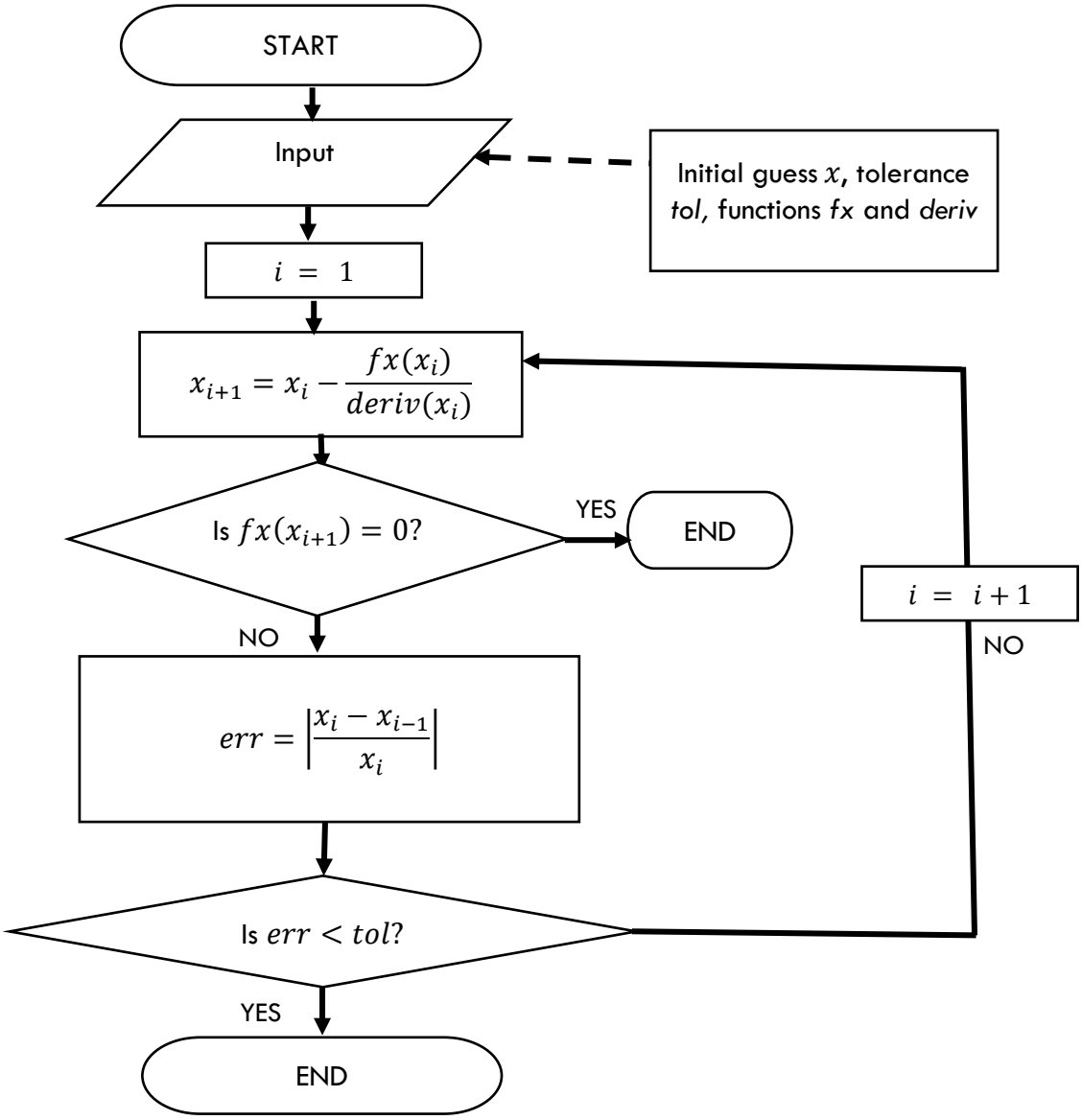

Figure 6.7. *Flowchart for Newton-Raphson method*

```
!=============================================================================================
! Code_14_newton.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the Newton-Raphson
! method.
!=============================================================================================

program newton

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables.
	integer :: i = 1
	real*16 :: x, error = 0.5, tol = 1e-10
	real*16, dimension(100) :: x_array
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "Please specify your estimate of the root."
	write(*, "(a15)", advance = "no") "estimate = "
	read(*,*) x
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`x` stores the initial guess, while `x_array` is used to keep track of all the updated guesses, for the error computation. `error` stores the RAE per iteration while `tol` stores the pre-specified tolerance, which in this case is 10^{-10}. `i` is used as a counter.

The user is then prompted to specify his/her initial guess. Note that while convergence is always guaranteed for bracketing methods such as the bisection method, this is not so for open methods like the Newton-Raphson method. Thus, some degree of intelligence must be exercised in choosing a guess for the algorithm.

```
!---------------------------------------------------------------------------------------
! This implements the Newton-Raphson method.
! This also prints out x, f(x), f'(x) and the RAE per iteration.

! Check if root is at the estimate.
if(abs(fx(x)) < tol) then
	write(*,*) "The root is at x = ", x
	call exit
end if

write(*,"(a20, 2x, a20, 2x, a20, 2x, a20)") "x", "f(x)", "f'(x)", "error"
```

```
do while(error > tol .or. abs(fx(x)) > tol)
       ! Initialize
       x_array(i) = x

       ! Compute error
       if(i > 1) then
              error = abs((x_array(i) - x_array(i - 1))/x_array(i))
       end if

       ! Print results
       if(i == 1) then
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, a20)") x, fx(x), deriv(x), "-"
       else
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, f20.10)") x, fx(x), deriv(x),
              error
       end if

       ! Evaluate new x
       x = x - (fx(x) / deriv(x))

       ! Update counter
       i = i + 1

end do
!-------------------------------------------------------------------------------------
```

As the comments suggest, this code snippet implements the Newton-Raphson method and outputs `x`, `fx(x)`, `deriv(x)` and `error` per iteration. The program first checks if the root is at `x`.

From then on, the code is fairly straightforward and follows the algorithm specified in Figure 6.7.

```
contains
       !=====================================================================================
       ! These functions specify the nonlinear equation to be solved and its first
       ! derivative. In this case, it is x^3 - 9*x^2 + 36*x - 80. Edit if needed.
       !=====================================================================================
       function fx(j) result(fxeval)
              real*16 :: j, fxeval

              fxeval = j**3 - 9*j**2 + 36*j - 80
       end function fx

       function deriv(j) result(deriveval)
              real*16 :: j, deriveval

              deriveval = 3*j**2 - 18*j + 36
       end function deriv
```

As Figure 6.7 points out, the program requires two functions: `fx` and its first derivative `deriv`. Both must be specified by the user in the code and can be edited to the user's tastes. One primary drawback of the Newton-Raphson method is that in spite of the speed of its convergence, it requires the user to symbolically evaluate the derivative of a function. In some cases, the derivative may be particularly difficult to compute by hand.

The full program is as follows:

```
!=============================================================================================
! Code_14_newton.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the Newton-Raphson
! method.
!=============================================================================================

program newton

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables.
	integer :: i = 1
	real*16 :: x, error = 0.5, tol = 1e-10
	real*16, dimension(100) :: x_array
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "Please specify your estimate of the root."
	write(*, "(a15)", advance = "no") "estimate = "
	read(*,*) x
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! This implements the Newton-Raphson method.
	! This also prints out x, f(x), f'(x) and the RAE per iteration.

	! Check if root is at the estimate.
	if(abs(fx(x)) < tol) then
		write(*,*) "The root is at x = ", x
		call exit
	end if

	write(*,"(a20, 2x, a20, 2x, a20, 2x, a20)") "x", "f(x)", "f'(x)", "error"
```

```
do while(error > tol .or. abs(fx(x)) > tol)
       ! Initialize
       x_array(i) = x

       ! Compute error
       if(i > 1) then
              error = abs((x_array(i) - x_array(i - 1))/x_array(i))
       end if

       ! Print results
       if(i == 1) then
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, a20)") x, fx(x),
       deriv(x), "-"
       else
              write(*,"(f20.10, 2x, f20.10, 2x, f20.10, 2x, f20.10)") x, fx(x),
              deriv(x), error
       end if

       ! Evaluate new x
       x = x - (fx(x) / deriv(x))

       ! Update counter
       i = i + 1

end do
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! Print out root
write(*,*)
write(*,*) "The root is at x = ", x
!------------------------------------------------------------------------------

contains
       !==============================================================================
       ! These functions specify the nonlinear equation to be solved and its first
       ! derivative. In this case, it is x^3 - 9*x^2 + 36*x - 80. Edit if needed.
       !==============================================================================
       function fx(j) result(fxeval)
              real*16 :: j, fxeval

              fxeval = j**3 - 9*j**2 + 36*j - 80
       end function fx
```

```
        function deriv(j) result(deriveval)
                real*16 :: j, deriveval

                deriveval = 3*j**2 - 18*j + 36
        end function deriv

end program newton
```

Executing the program,

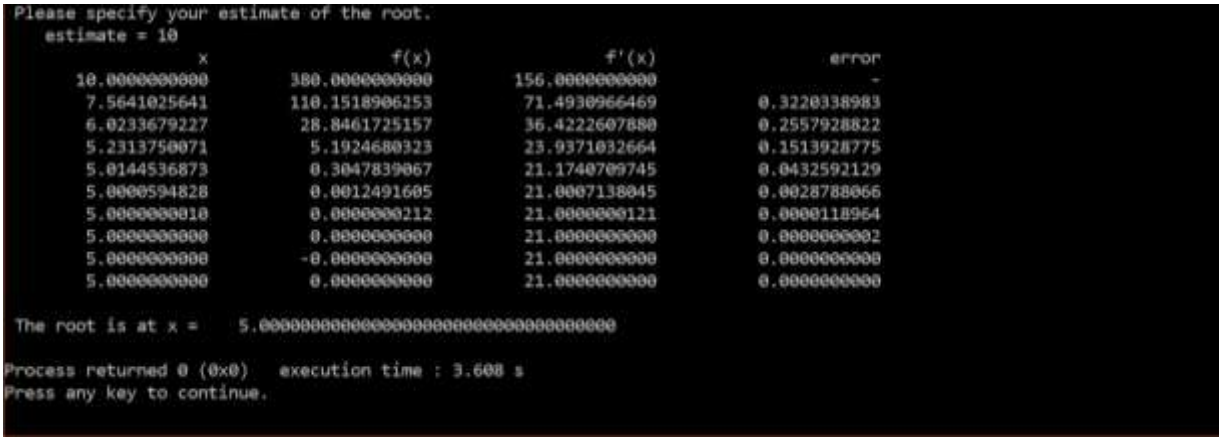

Figure 6.8. *Solving* $x^3 - 9x^2 + 36x - 80 = 0$ *using Code_14_newton. f90*

we find that the solution to $x^3 - 9x^2 + 36x - 80 = 0$ is given by

$$x = \mathbf{5.00000}$$

Secant method – Fortran example

To implement the bisection method in Fortran, we first need to see the algorithm in a flowchart. The algorithm is summarized in Figure 6.9

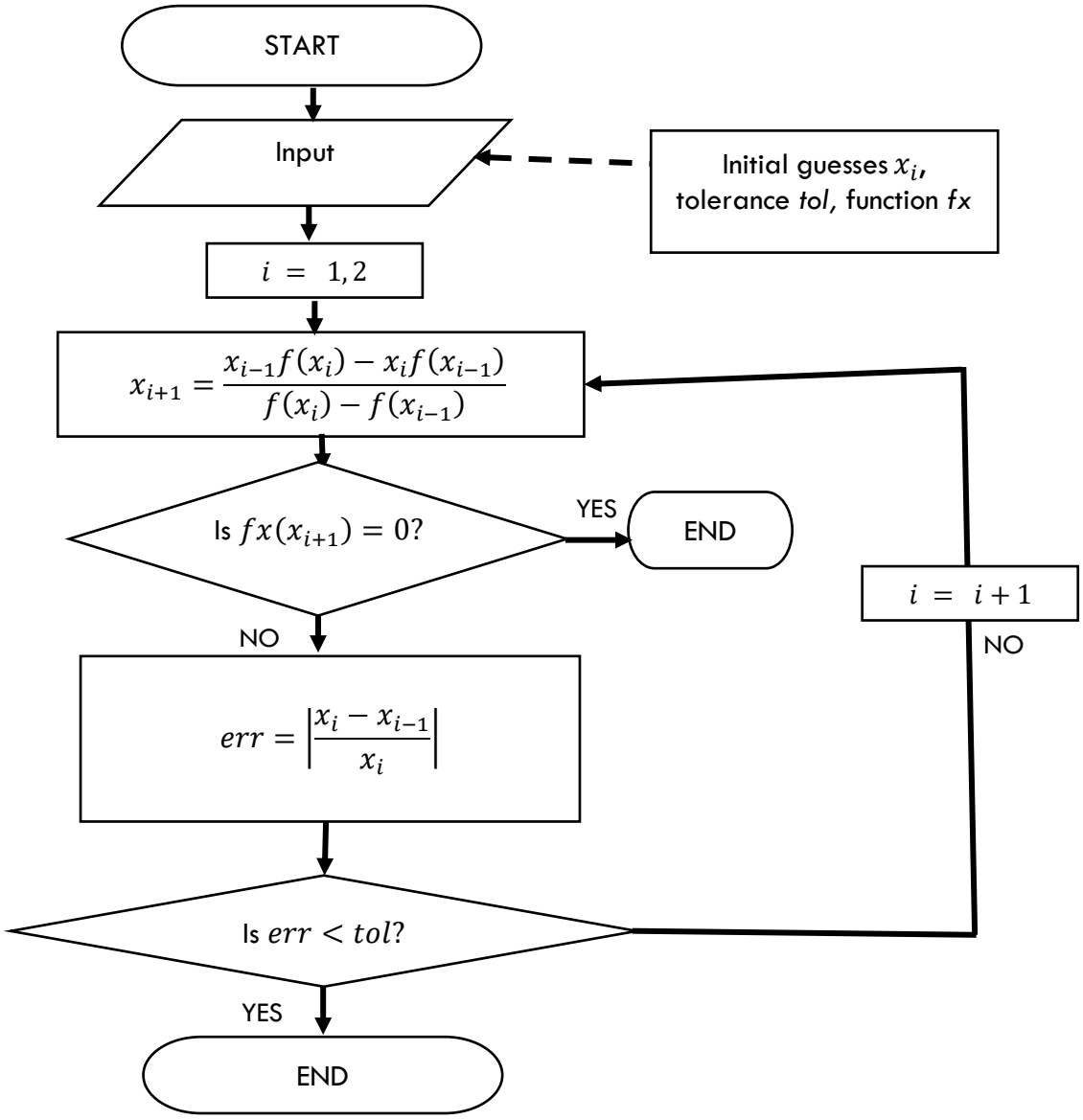

Figure 6.9. *Flowchart for secant method*

```
!=============================================================================================
! Code_15_secant.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the secant method.
!=============================================================================================

program secant

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables.
	integer :: i = 1
	real*16 :: x, error = 0.5, tol = 1e-10
	real*16, dimension(100) :: x_array
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "Please specify two estimates of the root."
	write(*, "(a15)", advance = "no") "estimate = "
	read(*,*) x
	x_array(1) = x

	! Check if root is at the estimate.
	if(abs(fx(x)) < tol) then
		write(*,*) "The root is at x = ", x
		call exit
	end if

	write(*, "(a15)", advance = "no") "estimate = "
	read(*,*) x
	x_array(2) = x

	! Check if root is at the estimate.
	if(abs(fx(x)) < tol) then
		write(*,*) "The root is at x = ", x
		call exit
	end if
	!---------------------------------------------------------------------------------------
```

The program starts by declaring all the necessary variables.

`x` stores both initial guesses (one at a time, of course), while `x_array` is used to keep track of all the updated guesses, for the error computation. `error` stores the RAE per iteration while `tol` stores the pre-specified tolerance, which in this case is 10^{-10}. `i` is used as a counter.

The user is then prompted to specify his/her initial guesses. Note that while convergence is always guaranteed for bracketing methods such as the bisection method, this is not so for open methods

like the Newton-Raphson method. Thus, some degree of intelligence must be exercised in choosing a guess for the algorithm.

The program checks whether the guesses are roots or not and then proceeds with the implementation of the secant method.

```
!-------------------------------------------------------------------------------------------
! Print results
! This implements the secant method.
! This also prints out x, f(x), f'(x) and the RAE per iteration.
write(*,"(a20, 2x, a20, 2x, a20)") "x", "f(x)", "error"
write(*,"(f20.10, 2x, f20.10, 2x, a20)") x_array(1), fx(x_array(1)), "-"

do while(error > tol .or. abs(fx(x)) > tol)
       ! Initialize
       x_array(i) = x

       ! Compute error
       if(i > 1) then
              error = abs((x_array(i) - x_array(i - 1))/x_array(i))
       end if

       ! Print results
       write(*,"(f20.10, 2x, f20.10, 2x, f20.10)") x, fx(x), error

       ! Evaluate new x
       x = (x_array(i - 1) * fx(x_array(i)) - x_array(i) * fx(x_array(i -
       1)))/(fx(x_array(i)) - fx(x_array(i - 1)))

       ! Update counter
       i = i + 1

end do
!-------------------------------------------------------------------------------------------

!-------------------------------------------------------------------------------------------
! Print out root
write(*,*)
write(*,*) "The root is at x = ", x
!-------------------------------------------------------------------------------------------
```

As the comments suggest, this code snippet implements the secant method and outputs `x`, `fx(x)` and `error` per iteration. The code is fairly straightforward and follows the algorithm specified in Figure 6.9.

```
contains
	!=============================================================================
	! This function specifies the nonlinear equation to be solved.
	! In this case, it is x^3 - 9*x^2 + 36*x - 80. Edit if needed.
	!=============================================================================
	function fx(j) result(fxeval)
		real*16 :: j, fxeval

		fxeval = j**3 - 9*j**2 + 36*j - 80
	end function fx
```

As opposed to the Newton-Raphson method, the secant method only requires `fx`, the nonlinear function whose roots are to be evaluated. `fx` must be specified by the user in the code and can be edited to the user's tastes.

The full program is as follows:

```
!=============================================================================================
! Code_15_secant.f90
! Justin Gabrielle A. Manay
! COMPHY2, Physics Department, De La Salle University
! This program calculates the roots of a nonlinear equation using the secant method.
!=============================================================================================

program secant

	implicit none
	!---------------------------------------------------------------------------------------
	! Declare variables.
	integer :: i = 1
	real*16 :: x, error = 0.5, tol = 1e-10
	real*16, dimension(100) :: x_array
	!---------------------------------------------------------------------------------------

	!---------------------------------------------------------------------------------------
	! User input
	write(*,*) "Please specify two estimates of the root."
	write(*, "(a15)", advance = "no") "estimate = "
	read(*,*) x
	x_array(1) = x

	! Check if root is at the estimate.
	if(abs(fx(x)) < tol) then
		write(*,*) "The root is at x = ", x
		call exit
	end if
```

```
write(*, "(a15)", advance = "no") "estimate = "
read(*,*) x
x_array(2) = x

! Check if root is at the estimate.
if(abs(fx(x)) < tol) then
       write(*,*) "The root is at x = ", x
       call exit
end if
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! Print results
! This implements the secant method.
! This also prints out x, f(x), f'(x) and the RAE per iteration.
write(*,"(a20, 2x, a20, 2x, a20)") "x", "f(x)", "error"
write(*,"(f20.10, 2x, f20.10, 2x, a20)") x_array(1), fx(x_array(1)), "-"

do while(error > tol .or. abs(fx(x)) > tol)
       ! Initialize
       x_array(i) = x

       ! Compute error
       if(i > 1) then
              error = abs((x_array(i) - x_array(i - 1))/x_array(i))
       end if

       ! Print results
       write(*,"(f20.10, 2x, f20.10, 2x, f20.10)") x, fx(x), error

       ! Evaluate new x
       x = (x_array(i - 1) * fx(x_array(i)) - x_array(i) * fx(x_array(i -
       1)))/(fx(x_array(i)) - fx(x_array(i - 1)))

       ! Update counter
       i = i + 1

end do
!------------------------------------------------------------------------------

!------------------------------------------------------------------------------
! Print out root
write(*,*)
write(*,*) "The root is at x = ", x
!------------------------------------------------------------------------------
```

```
contains
	!=============================================================================
	! This function specifies the nonlinear equation to be solved.
	! In this case, it is x^3 - 9*x^2 + 36*x - 80. Edit if needed.
	!=============================================================================
	function fx(j) result(fxeval)
		real*16 :: j, fxeval

		fxeval = j**3 - 9*j**2 + 36*j - 80
	end function fx

end program secant
```

Executing the program,

```
Please specify two estimates of the root.
   estimate = 12
   estimate = 19
                  x                  f(x)                 error
      12.0000000000       784.0000000000                     -
      19.0000000000      4214.0000000000        0.3684210526
      10.4000000000       445.8240000000        0.8269230769
       9.3825086726       291.4431125686        0.1084455515
       7.4616696077       102.9712332761        0.2574275150
       6.4122230367        44.4394279776        0.1636634541
       5.6154458817        15.4301199878        0.1418902740
       5.1916382159         4.2517917210        0.0816327425
       5.0304386267         0.6447984214        0.0320448377
       5.0016220384         0.0340785973        0.0057614486
       5.0000140525         0.0002951029        0.0003215963
       5.0000000065         0.0000001367        0.0000028092
       5.0000000000         0.0000000000        0.0000000013
       5.0000000000         0.0000000000        0.0000000000

 The root is at x =    5.000000000000000000000000000000000000

Process returned 0 (0x0)   execution time : 6.646 s
Press any key to continue.
```

Figure 6.10. *Solving* $x^3 - 9x^2 + 36x - 80 = 0$ *using* *Code_15_secant. f90*

we find that the solution to $x^3 - 9x^2 + 36x - 80 = 0$ is given by

$$x = \mathbf{5.00000}$$

Application – Finite square well: determining its energy spectrum

Among the many soluble potentials in introductory quantum mechanics is the finite square well potential, whose solution requires solving for the roots of a transcendental function. This cannot be done analytically; hence, we will use the methods learned earlier in this chapter to determine the energy spectrum of a finite square well potential.

To do this, we must begin with the finite square well potential itself. It is given by

$$V(x) = \begin{cases} V_0 & \text{if } -\frac{L}{2} \le x \le \frac{L}{2} \\ 0 & \text{if } x < -\frac{L}{2} or\ x > \frac{L}{2} \end{cases} \tag{6.28}$$

We substitute (6.28) to the Schrödinger equation

$$-\frac{\hbar}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi = E\psi \tag{6.29}$$

and solve for $\psi(x)$.

This yields the following solutions for the bound states (when $E < V_0$):

$$\psi(x) = \begin{cases} Ce^{\beta x} & \text{for } x \le -\frac{L}{2} \\ A\sin(\alpha x) + B\cos(\alpha x) & \text{for } -\frac{L}{2} \le x \le \frac{L}{2} \\ De^{-\beta x} & \text{for } x > \frac{L}{2} \end{cases} \tag{6.30}$$

where $\alpha = \frac{\sqrt{2mE}}{\hbar}$ and $\beta = \frac{\sqrt{2m(V_0 - E)}}{\hbar}$

The details of this are left for the reader to find out using a textbook on introductory quantum mechanics, like Griffiths (2005). However, the bound states will have both even and off solutions. Also, because the solution and its first derivative must satisfy the continuity condition, we end up with the following conditions:

For even solutions:

$$\alpha\tan\left(\alpha\frac{L}{2}\right) = \beta \tag{6.31}$$

and for odd ones:

$$\alpha\cot\left(\alpha\frac{L}{2}\right) = \beta \tag{6.32}$$

We can recast (6.31) and (6.32) so that for the even states:

$$\beta\cos\left(\alpha\frac{L}{2}\right) - \alpha\sin\left(\alpha\frac{L}{2}\right) = 0 \tag{6.33}$$

and for odd ones:

$$\alpha\cos\left(\alpha\frac{L}{2}\right)+\beta\sin\left(\alpha\frac{L}{2}\right)=0 \tag{6.34}$$

We can now find the roots for (6.33) and (6.34) to get the energy eigenvalues E.

REFERENCES

Alexander, C.K. & Sadiku, M. A. (2006). *Fundamentals of electric circuits*. McGraw-Hill.

Atkinson, K. E. (2008). An introduction to numerical analysis. John Wiley & Sons.

Cheney, W., & Kincaid, D. (2008). Numerical methods and computing. Brooks.

Giordano, N. J. & Nakanishi H. (2006). Computational physics. Pearson Education India.

Gould, H., Tobochnik, J., & Christian, W. (1988). An introduction to computer simulation methods. New York: Addison-Wesley.

Kaw, A. K., Kalu, E.K., & Nguyen, D., "Numerical Methods with Applications" (2011). *Textbooks Collection*.

Meyer, C. D. (2000). Matrix analysis and applied linear algebra. SIAM.

Physics Department. (2012). Physics Laboratory 1 (Compiled Experiments in Mechanics). De La Salle University – Manila. Retrieved from http://www.dlsu.edu.ph/academics/colleges/cos/physics/experiments.asp

Physics Department. (2012). Physics Laboratory 2 (Compiled Experiments in Heat, Electricity & Magnetism, Optics). De La Salle University – Manila. Retrieved from http://www.dlsu.edu.ph/academics/colleges/cos/physics/experiments.asp

Weber-Wulff, D. (1992). Rounding error changes parliament makeup. The Risks Digest, 13(37).