

LECTURE NOTES ON ENGINEERING COMPUTING

Joseph M. Powers

Department of Aerospace and Mechanical Engineering
University of Notre Dame
Notre Dame, Indiana 46556-5637
USA

updated
29 April 2021, 3:45pm

Contents

Preface	7
1 Overview	9
1.1 Motivation	9
1.2 A simple program	14
1.2.1 Downloading the program	14
1.2.2 Saving the program onto the AFS server	15
1.2.3 Accessing the campus Linux cluster	15
1.2.4 Compiling a Fortran program	18
1.2.5 Editing text	19
1.3 Speed test: Fortran versus MATLAB	20
1.4 Some definitions	21
1.5 Introduction to engineering plotting with MATLAB	22
1.5.1 A bad MATLAB plot.	23
1.5.2 A better MATLAB plot	24
1.5.3 Another bad MATLAB plot	25
1.5.4 Another better MATLAB plot	26
1.6 Introduction to L ^A T _E X	27
1.7 Introduction to HTML	28
1.8 UNIX	30
1.9 Online compilers	33
2 Introduction to problem solving	35
3 Introduction to programming languages	37
4 Introduction to programming	39
4.1 Statement types	39
4.2 Example program	39
4.3 Data types	41
4.4 Another example	42

5	Arithmetic	45
5.1	Operations	45
5.2	Integer division	46
5.3	Binary representation of integers	48
5.4	Overflow and underflow limits	49
5.5	Assignment of undefined variables	51
6	Arrays 1	53
6.1	Dimension of an array	53
6.2	One-dimensional arrays from time series	54
6.2.1	Fixed array size	54
6.2.2	Variable array size via <code>parameter</code> declaration	59
6.3	Simple writing to files and <code>MATLAB</code> plotting	60
7	Arrays 2	63
7.1	Varying the array size at run time	63
7.2	Higher dimension arrays	65
7.3	Two-dimensional arrays of variable size	69
7.4	Non-standard <code>do</code> loop increment syntax	70
8	Whole array features	73
8.1	Dot product	74
8.1.1	Real vectors	74
8.1.2	Complex vectors	75
8.2	Matrix transpose	76
8.3	Matrix-vector multiplication	78
8.4	Matrix-matrix multiplication	80
8.5	Addition	84
8.6	Element-by-element matrix “multiplication”	86
9	Output of results	89
9.1	Unformatted printing	89
9.2	Formatted printing	90
9.3	Formatted writing to files	96
10	Reading data	99
10.1	Unformatted reading from a file	99
10.2	Formatted reading from a file	102
11	I/O Concepts	105

12 Functions	107
12.1 Pre-defined functions	107
12.2 User-defined functions	109
12.2.1 Simple structure	110
12.2.2 Use of dummy variables	112
12.2.3 Sharing data using <code>module</code>	113
12.2.4 Splitting functions into separate files	115
12.2.5 Creation of executable script files	117
13 Control structures	119
14 Characters	127
15 Complex	129
16 Logical	135
17 Introduction to derived types	137
18 An introduction to pointers	139
19 Introduction to subroutines	141
19.1 Simple subroutine	141
19.2 Subroutines for solution of a quadratic equation	143
19.3 Subroutines for solving systems of ordinary differential equations	145
19.3.1 One linear ordinary differential equation	146
19.3.2 Three non-linear ordinary differential equations	150
20 Subroutines: 2	161
21 Modules	163
22 Converting from Fortran 77	165
22.1 A <code>hello world</code> program	165
22.2 A simple program using arrays	166
22.3 A program using subroutines	168
23 Python and Fortran	173
24 Introduction to C	175
24.1 A <code>hello world</code> program	175
24.2 A program using the Euler method	176

25 Introduction to Microsoft Excel	179
25.1 A hello world program	179
25.2 A program using the Euler method	180
26 Introduction to VBA	183
26.1 A hello world program	183
26.2 A program using the Euler method	184
26.3 Another VBA Example	187
26.3.1 Creating a calculation macro	187
26.3.2 Clearing data	190
26.3.2.1 Manual clearing	190
26.3.2.2 Automated clearing	191
26.3.3 Automated plotting	193
26.3.4 Miscellaneous	194
27 Introduction to Mathematica	195
27.1 A hello world program	195
27.2 A simple code	195
27.3 Precision	196
27.4 Lists, vectors, and matrices	197
27.5 Algebra	200
27.6 Some plots	201
27.7 Calculus	206
27.8 The Euler method, DSolve, and NDSolve	207
Bibliography	211
Index	213

Preface

These are lecture notes for AME 20214, Introduction to Engineering Computing, a one-hour sophomore-level undergraduate course taught in the Department of Aerospace and Mechanical Engineering at the University of Notre Dame.

The key objective of the course is to introduce students to the **UNIX** operating system and structured high performance computing with a compiled language, here chosen to be **Fortran 90** and its descendants, as well as some others, such as the interpreted languages **MATLAB**, **Python**, **Mathematica**, **VBA**, and **Microsoft Excel**, and a very brief introduction to the compiled language **C**. Thanks to Mr. Travis Brown for help with a **C** example program and Ms. Laura Paquin for help with **VBA** example programs and description.

An important secondary objective is to introduce the student to the process of scientific computing: the art of using a computer to solve problems of scientific and engineering relevance. To achieve that objective, some attention is focused on numerically solving physically motivated systems of ordinary differential equations. The most important of these will be the forced mass-spring-damper system, which will be addressed with a variety of methods.

The notes draw often on the text specified for the course, Chivers' and Sleightholme's *Introduction to Programming with Fortran, Third Edition*, Springer, London, 2015. The notes will highlight aspects of this text, and augment it in places. The notes and course data can be found at <https://www3.nd.edu/~powers/ame.20214>. At this stage, anyone is free to make copies for their own use. I would be happy to receive suggestions for improvement.

Joseph M. Powers

powers@nd.edu

<https://www3.nd.edu/~powers>

Notre Dame, Indiana; USA

 29 April 2021

The content of this book is licensed under Creative Commons Attribution-Noncommercial-No Derivative Works 3.0.

Chapter 1

Overview

Read CES, Chapter 1.

1.1 Motivation

We are faced with a problem in constrained optimization. We seek to maximize student understanding of

- mathematical methods to computationally solve problems arising in aerospace and mechanical engineering,
- programming languages for efficient solution of large-scale mathematical problems arising in engineering, and
- operating systems beyond those found by default in consumer-market computers,

subject to the constraints

- finite time, embodied in the one credit hour allocated to this task,
- finite student skills, and
- finite instructor skills.

We might conjecture that the understanding of these topics, Υ , is a function of the mathematical method μ , the language chosen, λ , the operating system, σ , the pedagogy ρ , chosen, the time τ allotted for study, the student and instructor skill sets, κ_1 and κ_2 . In a mathematically inspired language, we might say

$$\Upsilon = f(\mu, \lambda, \sigma, \rho, \tau, \kappa_1, \kappa_2). \tag{1.1}$$

Our constraints are for time τ , student skills κ_1 and instructor skills κ_2 . Our constrained optimization problem might be stated as, select μ , λ , σ , and ρ so as to

$$\text{Maximize: } \Upsilon = f(\mu, \lambda, \sigma, \rho, \tau, \kappa_1, \kappa_2), \quad (1.2)$$

$$\text{Subject to: } \tau = 1, \quad (1.3)$$

$$\kappa_1 = \text{constant}, \quad (1.4)$$

$$\kappa_2 = \text{constant}. \quad (1.5)$$

We are presented with a wide variety of mathematical problems μ of engineering relevance including

- linear systems of algebraic equations,
- non-linear systems of algebraic equations,
- linear systems of ordinary differential equations,
- non-linear systems of differential equations,
- partial differential equations, or
- optimization problems.

We are presented with a wide variety of computing languages to formulate these problems, including

- Fortran (from “FORMula TRANslating system”),
- COBOL (from “Common Business-Oriented Language”),
- ALGOL (from “ALGORithmic Language”),
- Ada,
- BASIC (from “Beginner’s All-purpose Symbolic Instruction Code”),
- VBA (from “Visual Basic for Applications”),
- Pascal,
- C,
- C++,
- C#,
- Java,

- MATLAB (from “MATrix LABoratory”),
- Mathematica, or
- Microsoft Excel.

We are presented with a wide variety of operating system environments in which to execute our programs including

- Microsoft Windows,
- DOS,
- OS-X,
- UNIX, or
- Linux.

Following considerable consideration, coupled with some experimentation, we believe the near-optimal solution, reflecting the severe constraint of $\tau = 1$ *credit hour* is given by the following: student understanding, Υ , of engineering mathematics, relevant languages, and advanced operating systems is given by choosing

- $\lambda = \text{Fortran 2003}$,
- $\sigma = \text{Linux}$.

Near neighbors within the solution space are also perfectly acceptable, e.g. Fortran 90, Fortran 95, Fortran 2008, UNIX, OS-X. If the constraints of the curriculum evolve, and languages and operating systems evolve, the optimal solution may evolve!

All choices have advantages and disadvantages. Certainly one chooses modes of transportation depending on a variety of needs. We depict rough and imperfect analogies between some modern transportation choices and modern computing languages in Fig. 1.1.¹ For transportation purposes, one might imagine that

- The racing cycle has speed, power, and agility but requires a highly trained user to achieve peak performance. It maps into C.
- The modern muscle car has all features expected by most mass-market drivers, draws heavily on past experiences, but still may not be the right machine for all market segments. It maps into Fortran.

¹images from

http://motorsportsnewswire.files.wordpress.com/2011/05/infineon2011_brammo_empulse_rr.jpg
http://www.netcarshow.com/ford/2013-mustang_boss_302/1600x1200/wallpaper_01.htm
<http://www.motortrend.com/news/2017-dodge-grand-caravan-first-drive-not-dead-yet/>
http://www.motorzoeker.nl/public/motorfotos/large/vespa/5611_foto1_k25j3xt2j8.jpg
https://commons.wikimedia.org/wiki/File:Campagna_T-Rex_14-R_rear_left.jpg



Figure 1.1: Loose transportation analogies with computing languages.

- The minivan is appropriate for a wide variety of drivers and passengers, is durable, and optimized for safety; it will not compete well in a race. It maps into **MATLAB**.
- The scooter is affordable, maneuverable, and is appropriate for many urban environments; it is not right for the highway. It maps into **Excel**.
- The motorized reverse tricycle is stable for some highway use, but unconventional. It maps into **VBA**.

Now focusing on our actual computing systems, we list some advantages and disadvantages associated with our present choices. First for **Fortran**:

- **Fortran** advantages
 - a *compiled language* which yields very fast binary executable code widely used for large scale problems in computational science, e.g. the widely used Top 500 supercomputing benchmark problem is based on **Fortran**-implemented algorithms.
 - used in many modern supercomputing applications, e.g. the Department of Energy's major initiative in scientific computing for the 21st century, Predictive Science Academic Alliance Program (PSAAP),
 - explicitly designed for mathematical problems common in engineering; pedagogically straightforward implementation of standard notions such as sine, cosine, exponential, vectors, and matrices,
 - widely used for decades in engineering with a large data base of legacy code and packaged software,²

²see <http://www.netlib.org/>

- another language in the engineer’s multi-lingual arsenal for problem-solving,
 - a language which opens the door to learning a wide variety of compiled languages, such as **C**, which was designed for device drivers and operating systems (not scientific computing), and
 - forces care and precision to be used in code writing—thus reinforcing fundamental engineering design principles for quality control which transcend programming.
- **Fortran** disadvantages
 - forces care and precision to be used in code writing—which induces frustration when the code fails to work properly,
 - relative to some other languages, does not interact as easily with the operating system and input/output data, thus preventing use in many hardware applications, e.g. robotics,
 - no longer commonly used in the computer science community, especially the large fraction of that community who no longer are concerned with scientific computing of mathematics-based engineering problems based on differential equations, and
 - not particularly relevant to the large user community who primarily run pre-programmed specialty software packages for engineering analysis and design, but do not generate original source code.

For **Linux**, we have

- **Linux** advantages
 - widely used,
 - allows user to more easily understand computer system architecture,
 - freely available
 - very similar to **UNIX** and **OS-X**.
- **Linux** disadvantages
 - more effort to learn, relative to What-You-See-Is-What-You-Get (**WYSIWYG**) systems such as **Windows** or the front end of **OS-X**.
 - users must manage their own systems.

Whatever language and operating system we use, and this course will consider **Fortran** with short excursions into **C**, **MATLAB**, **Microsoft Excel**, **VBA**, and **Mathematica**, the essential job of nearly any programming language is executing so-called three R’s:

- Readin’
- ‘Ritin’

- ‘Rithmetic

A typical program will change the order (and we will now use proper spelling) and first

- *Read* input data from a file or the screen,
- Perform *arithmetic* operations on the data, and
- *Write* output data to a file or the screen.

Recognizing the various methods that various languages use to execute the three R’s is the key to learning a new computer language. There are a small number of typical operations in scientific computing. Once the student learns how to do an operation in one language, it is much easier to translate these steps to any of the other languages. Thus, in some sense, learning *a* computer language may be more important than learning any particular computer language. That is because learning *a* language typically entails learning the more complex generalities associated with all languages.

1.2 A simple program

Let us consider one of the simplest possible **Fortran** that one can imagine, a so-called “hello world” program.

1.2.1 Downloading the program

The program, with file name `helloworld.f90`, is given here:

```
program test
print*, 'hello world'
end program test
```

One can download this program from the web by clicking the following web link,

<https://www3.nd.edu/~powers/ame.20214/2e/helloworld.f90>

The first line of the program is the program name, where the name is arbitrary. The second line tells the program to print to the screen, indicated by a *, the words within the single quotation marks. The program is ended by a program end statement. The file name which ends in “.f90” is essential and tells the compiler that this file follows the syntax of **Fortran 90** and beyond, beyond including **Fortran 90**, **95**, **2003**, and **2008**. It should be noted that our local system can in fact invoke the **Fortran 2003** standard, but for nearly all problems we will consider, the **Fortran 90** standard will suffice.

1.2.2 Saving the program onto the AFS server

After downloading, one should save the file onto the campus file server known as **AFS** and choose to name the saved file as **helloworld.f90**. The letters **AFS** are an acronym for “Andrew File System.” Files saved onto your **AFS** space can be accessed and edited from a wide variety of campus and off-campus machines. In many ways storing files on **AFS** is like storing them on the “cloud,” though **AFS** has a much longer history than the cloud. Importantly, it is possible to compile and execute *binary executable* files from **AFS**.

There are a variety of ways to save the file. Most browsers have some variety of **save as** option. If exercised, be sure to save the file as plain text with a **.f90** extension in its name. Or one can cut and paste the necessary text into a variety of text editors such as **gedit**, described in more detail in Sec. 1.2.5.

Note that the **AFS** space is distinct from the more common campus space in **netfile**. For purposes of running compiled programs which are binary executable, such as those originating in **Fortran** or **C** source codes, **netfile** is completely unusable. In short, while you could certainly save a text file within **netfile**, you could never compile or execute the program.

1.2.3 Accessing the campus Linux cluster

To compile and execute the program, you will need to log onto a machine which has a **Fortran** compiler.

It is strongly recommended to directly log in by actually sitting at a campus Linux cluster machine when first learning the Linux operating system. Indeed, everything that can be achieved sitting in front of the Linux machine can also be achieved remotely via a laptop; however, many things can go wrong. After brief experience in front of the actual machine, it is not a difficult step to use Linux remotely from a laptop.

That said, many users do have sufficient experience to go directly to remote login from a laptop, and that procedure is described here. One important machine for remote login is the campus **Linux** machine **darrow.cc.nd.edu**. The machine **darrow** is useful because all campus personnel can log in to it from any remote site, on or off campus. Here is what one might do if logging in from the **Terminal** application of a personal **Macintosh** machine using the so-called “secure shell” program **ssh**:

```
esc300037:~ powers$ ssh -x darrow.cc.nd.edu -l powers
powers@darrow.cc.nd.edu's password:
Last login: Wed Aug  2 08:52:14 2017 from c-98-253-148-207.hsd1.in.comcast.net

[powers@darrow5 ~]$
```

Here, the argument of the `-l` option is your *netid*. Mine is **powers**. The general form is

```
ssh -X darrow.cc.nd.edu -l netid
```

If `darrow.cc.nd.edu` is overloaded with too many users, a user may log in remotely to any of a set of virtual Linux machines. Their names are

```
remote101.helios.nd.edu
remote102.helios.nd.edu
remote103.helios.nd.edu
remote104.helios.nd.edu
remote105.helios.nd.edu
remote106.helios.nd.edu
remote107.helios.nd.edu
remote108.helios.nd.edu
remote201.helios.nd.edu
remote202.helios.nd.edu
remote203.helios.nd.edu
remote204.helios.nd.edu
remote205.helios.nd.edu
remote206.helios.nd.edu
remote207.helios.nd.edu
remote208.helios.nd.edu
```

For example to log in remotely to one of the virtual Linux machines, one might execute the command sequence

```
esc300037:~ powers$ ssh -X remote103.helios.nd.edu -l powers
The authenticity of host 'remote103.helios.nd.edu (129.74.50.104)' can't be established.
RSA key fingerprint is 80:12:f5:2e:91:c7:97:83:8b:54:6a:2d:10:14:6f:e2.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'remote103.helios.nd.edu,129.74.50.104' (RSA) to the list of
*****
*****
**
**          This system is for the use of AUTHORIZED USERS ONLY.          **
**
**  Anyone using this system expressly consents to such monitoring and is    **
**  advised that if such monitoring reveals possible evidence of criminal    **
**  activity, system personnel may provide the evidence of such monitoring    **
**  to law enforcement officials.                                           **
**
**  See the University of Notre Dame Responsible Use Policy for information  **
```

```

** regarding proper use of this machine (http://oit.nd.edu/policies).      **
**                                                                           **
*****
*****

```

Remote use of this system is not for running big processes. If you need to run big processes for research, please get an account from Center of Research Computing (<http://crc.nd.edu>).

Please email help@esc.nd.edu for any question of suggestion.
powers@remote103.helios.nd.edu's password:
 Last login: Thu Nov 12 15:20:01 2015 from dbrt1551-00.cc.nd.edu
 [powers@remote103 ~]\$

Other campus Linux machines are useful as well, but the user must be sitting at the machine. A practical advantage of these physical machines is they only allow a single user, so they will not become overloaded. Several such Linux machines can be found in

- 149 Fitzpatrick Hall (Engineering Library),
- B-19 Fitzpatrick Hall, and
- 108-110, 212-213 Stinson-Remick Hall.

A list of available machine names and locations is found at

<http://pug.helios.nd.edu:2080/clusterfree.html>

For students with Macintosh laptops, one can download so-called X11, also known as XQuartz, software to create an X Windows environment. One can try the following site for download:

<http://www.xquartz.org>

Once downloaded, take care to open the terminal application from XQuartz rather than the standard Macintosh terminal application. From the XQuartz terminal, one can invoke the command `ssh` command to a Linux machine. Some more information can be found at

<https://en.wikipedia.org/wiki/XQuartz>

For students with Windows laptops, one can download software to create an X Windows terminal environment. Instructions can be found at

<https://www3.nd.edu/~powers/ame.20214/xming.pdf>

When invoking the so-called `xming` software, be sure to examine the “Connection” menu on the left and under SSH, X11, check `enable X11 forwarding`.

Another common way, which requires a little more effort, to bring UNIX functionality into a Windows environment is to install the free, open-source UNIX-emulator, Cygwin, onto a Windows machine.

<https://www.cygwin.com>

1.2.4 Compiling a Fortran program

On `darrow.cc.nd.edu`, the relevant virtual Linux machines, and all physical campus Linux cluster machines reside the Fortran compilers

- `ifort`, and
- `gfortran`.

The compiler `ifort` is the so-called Intel Fortran compiler. The compiler `gfortran` is an open source compiler. The `ifort` compiler works on most campus Linux machines, but may not work for all on `darrow.cc.nd.edu`. The `gfortran` compiler should work on all campus Linux machines. Focus here will be on the `ifort` compiler; though it is a useful exercise to make sure your program compiles and executes on a variety of compilers, as there can be subtle differences. All these compilers work in essentially the same way. For instance to use `ifort` to compile `helloworld.f90`, we perform

```
[powers@remote101 ame.20214]$ ifort helloworld.f90
```

This generates a so-called “binary” executable file, which is not easily viewable, named “`a.out`”. We execute the file `a.out` via the command and get

```
[powers@remote101 ame.20214]$ a.out
hello world.
[powers@remote101 ame.20214]$
```

The output of the program is printed to the screen. Though it is not at all important for this problem, we actually compiled the program with a Fortran 90 compiler. We can compile and execute the program in Fortran 2003 via the command sequence

```
[powers@remote101 ame.20214]$ ifort helloworld.f90 -stand f03
[powers@remote101 ame.20214]$ a.out
hello world.
[powers@remote101 ame.20214]$
```

Here, the so-called compiler option, `-stand f03`, enforced the compilation using the Fortran 2003 standard.

We can compile and execute the program with the `gfortran` compiler via the command sequence

```
[powers@darrow1-p ~]$ gfortran helloworld.f90
[powers@darrow1-p ~]$ a.out
hello world
[powers@darrow1-p ~]$
```

1.2.5 Editing text

You will need to be able to edit text files. This is best achieved with a text editor. For the relevant tasks, tools such as **Microsoft Word** are inappropriate, because they generally do not format the files in plain text; instead they often add extraneous data to the files for purposes of visual display. These visually appealing elements render the files unusable for programming purposes.

Instead, you should employ any of a variety of standard text editors. Perhaps the most straightforward is one known as **gedit**. It is available on the campus **Linux** cluster machines. To edit the file **helloworld.f90**, one can invoke on a campus **Linux** cluster machine the command

```
[powers@darrow1-p ~]$ gedit helloworld.f90 &
```

This command spawns a new window in which the file **helloworld.f90** can be edited and saved. The character **&** is a useful **UNIX** command which commands the system to allow the terminal window to be used simultaneously with the program **gedit**. If one is using **gedit** from a remote terminal or personal laptop machine, one must be sure to have logged in from a so-called X terminal window, using the command **ssh -X darrow.cc.nd.edu -l userid** in order for the edit window to display. This is straightforward on a **Macintosh** computer through use of its **terminal** application found in the **Utilities** folder. On a **Windows** laptop, one may wish to use the terminal available via the **Cygwin** package. This is not an issue if the user is seated at the actual machine in use.

An incomplete list of common text editors available on the campus **Linux** cluster, each with advantages and disadvantages is as follows:

- **gedit**
- **Kate**, invoked by **kate** on the Notre Dame cluster (but not on **darrow**)
- **vi**
- **Emacs**, invoked by **emacs** on the Notre Dame cluster.

One can also use ordinary text editors on personal computing machines and save the file as a plain text file in the appropriate space. Note that one should *not* save a **Fortran** source code file in the common “rich text format” (**.rtf**) and expect it to be able to compile correctly.

1.3 Speed test: Fortran versus MATLAB

Let us check the speed of an ordinary **Fortran** program versus its **MATLAB** equivalent. Speed tests are notorious for being unfair, depending on the options exercised. In this case, we will use options which are widely considered to be default for both programs, and we will execute on the same machine. Both programs will initialize a large matrix, of dimension $n \times n$. We will call the matrix $\mathbf{x}$ and its elements will be denoted x_{ij} . We will initialize so that each element takes on the value of the product of i and j : $x_{ij} = ij$, where $i = 1, \dots, n$, and $j = 1, \dots, n$. Thus $x_{11} = (1)(1) = 1$, $x_{12} = (1)(2) = 2$, $\dots$, $x_{25} = (2)(5) = 10$, and so on. If $n = 5$, we should thus form the matrix

$$\mathbf{x} = \begin{pmatrix} (1)(1) & (1)(2) & (1)(3) & (1)(4) & (1)(5) \\ (2)(1) & (2)(2) & (2)(3) & (2)(4) & (2)(5) \\ (3)(1) & (3)(2) & (3)(3) & (3)(4) & (3)(5) \\ (4)(1) & (4)(2) & (4)(3) & (4)(4) & (4)(5) \\ (5)(1) & (5)(2) & (5)(3) & (5)(4) & (5)(5) \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix}. \quad (1.6)$$

When n is large, it will take a long time to execute the initialization. Let us perform this process via **Fortran** and **MATLAB** on the linux machine, `darrow.cc.nd.edu`. We shall take $n = 5000$, a modestly sized matrix for scientific computing. The **Fortran** program `matrix.f90` and its execution are as follows:

```
[powers@darrow1-p ame.20214]$ cat matrix.f90
program matrix                                ! beginning of the program
implicit none                                ! force all variables to be declared
integer :: i,j                                ! declare i and j to be integers
integer, parameter :: n = 5*10**3           ! give the dimension of the matrix
real :: x(n,n)                                ! declare the matrix x to be n x n and real
do i=1,n                                      ! first do loop
  do j=1,n                                    ! second do loop
    x(i,j) = real(i)*real(j)                 ! initialize the matrix
  enddo                                       ! end of do second loop
enddo                                         ! end of first do loop
print*,n,x(n,n)                             ! print some values
end program matrix                           ! end of program
[powers@darrow1-p ame.20214]$ ifort matrix.f90
[powers@darrow1-p ame.20214]$ a.out
      6000  3.60000000E+07
[powers@darrow1-p ame.20214]$
```

We shall worry about the details of the program later. Note for now that there are many complexities to execute this seemingly simple task. This is typical of **Fortran** codes. We can check the speed of the program by executing with a UNIX utility `time`. We find

```
[powers@darrows2-p ame.20214]$ /usr/bin/time -p a.out
        6000  3.6000000E+07
real 0.46
user 0.40
sys 0.05
[powers@darrows2-p ame.20214]$
```

The execution of this particular code was achieved in

$$t_{\text{Fortran}} \sim 1.42 \text{ s.} \quad (1.7)$$

The MATLAB code, available in `matrix.m`, is given by

```
tic
n = 5*10^3;
for i=1:n
    for j=1:n
        x(i,j) = i*j;
    end
end
n
x(n,n)
toc
```

The MATLAB commands `tic` and `toc` are for timing the code. Relative to the **Fortran** code, the MATLAB code has a similar, but simpler structure and syntax. However, the execution of this particular code was achieved in a much longer time:

$$t_{\text{MATLAB}} \sim 12.724 \text{ s.} \quad (1.8)$$

It must be emphasized that these numbers are typical, but unreliable. The machine may have been performing different background tasks. Also, one could use optimizing **Fortran** compilers to improve **Fortran** execution time. The main reason for the fast execution of the **Fortran** program is that it was compiled prior to execution, whereas the MATLAB program, using a so-called *interpreted language* was not. All things equal, compiled languages are typically much faster than interpreted languages. However, in most cases, interpreted languages have been designed to be more user-friendly. Note that it is possible to use compiled versions of MATLAB to improve its execution time significantly.

1.4 Some definitions

Here are a few key concepts and terms associated with computer systems:

- *Bits and bytes* are the representation of data by combinations of 0 and 1. A byte is usually 8 bits. Most common machines today are based on a 64-bit architecture.

- *Central Processing Unit (CPU)* is, in colloquial terms, the main brain of the computer.
- One of the more important memory devices is *Random Access Memory (RAM)*. This memory is usually transient in that it is only active when the machine is turned on. It is not used for permanent storage of files. Permanent storage is achieved via hard drives, memory sticks, etc.
- A *terminal* is a device which allows direct input and output communications between a user and the CPU of a machine. Terminals come in many flavors, including virtual.
- *Software* is an imprecise term which describes programs which allow interaction between the user and the computer. High level software is often more user-friendly and has several layers of abstraction between the user and CPU. Low level software has fewer levels of abstraction, is often more difficult to understand, but often tightly linked to the CPU architecture. **Microsoft Excel** is a high level software tool. **MATLAB** is somewhat lower level, but still at a high level. **Fortran** is at a lower level still, but in many ways a high level language. So called *assembly language* is much closer to the machine, and is certainly a low level software tool.
- A *compiler*, such as **ifort**, is a tool to convert a high level language such as **Fortran** into a fast executable binary low level language usable by the machine. Use of compiled languages is highly advantageous for large scale problems in scientific computing because of the speed of execution. For small scale problems, non-compiled interpreted languages, such as **MATLAB**, are advantageous because of their relative simplicity of use.
- A *text editor* allows the user to alter the source code. Text editors typically refer to tools which give the user access to the entire **ASCII** (American Standard Code for Information Interchange) text of the file and avoid use of hidden and/or binary text.

1.5 Introduction to engineering plotting with MATLAB

Well done plots are an important part of communication for engineers. One should take care to try to adhere to some general plot design principles:

- *Use font sizes which are close to those of the printed text.* Most software tools today, unfortunately, use default font sizes which are too small for printed text; occasionally they are too large. Whatever the case, the text on your plot should have a font size very similar to that of your main text.
- *Label your axes; include units when appropriate.*
- *In axes labels and text, use fully formatted mathematical symbols.* For example, if the proper symbol is ϕ , use it instead of “phi.” Or if the proper symbol is y_1 , do not substitute y_1 or $y1$ or $y1$. The same holds for units; use as a label ρ ($\frac{\text{kg}}{\text{m}^3}$), not “rho (kg/m3).”

- *If there is more than one curve, be sure the reader knows which curve corresponds to which case.* There are a variety of ways to achieve this. The best is probably to use combinations of solid lines, dashed lines, dotted lines, etc. Different lines can also be assigned different colors, but one should recognize that sometimes the user will print files with black and white printers, obscuring the meaning. Also, some colors are hard to discern on varieties of computer screens and printers.
- *Generally, reserve point markers like small circles or dots for cases where*
 - The data comes from a finite, discrete set of experimental observations, or
 - The data comes from a finite, discrete set of computational runs.

Often it is best not to “connect the dots” especially if there is no reason to believe that another data point, if sampled, would lie exactly on the interpolating curve.

- *Generally, reserve smooth curves with no small circles or dots for cases where*
 - The curve is representing a continuous mathematical function such as $y = x^2 + 2x + 1$; Indeed, one must sample a discrete number of points to generate the plot, but the underlying function is continuous.
 - There is good reason to believe that the discrete data points are actually samples of a continuous function. We may not know the function, but when we continue to sample, we find the continuity assumption holds.
- *Use linear, log-linear, and log-log scaled plots as appropriate for the data at hand.* Sometimes linear scales hide the true nature of the data, and sometimes they reveal it. The same can be said for log-linear and log-log scales. Be sure to examine your data, and choose the most revelatory scaling.

1.5.1 A bad MATLAB plot.

Let us say we have the following data to plot:

x	y
0	1
2	4
3	8
4	7
7	10
10	19

MATLAB defaults typically produce faulty plots. The following MATLAB script is an example:

```
x = [0,1,3,4,7,10];  
y = [2,4,8,7,10,19];  
plot(x,y)
```

This script generates the plot of Fig. 1.2. Figure 1.2 has the following problems:

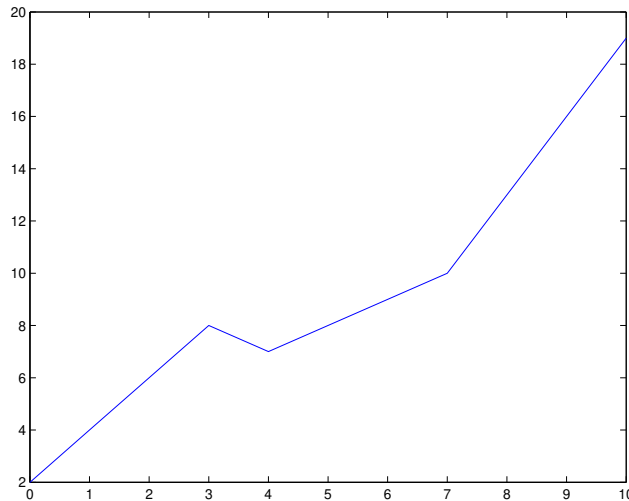


Figure 1.2: Badly formatted plot of some data.

- The numbers on the axes are much smaller than the font size of the surrounding text. This problem is pervasive when MATLAB default values are used.
- The axes are not labeled.
- The raw data should be indicated with distinct points. Though optional, it is probably best not to connect them.

1.5.2 A better MATLAB plot

A few simple changes to the MATLAB script can improve the plot dramatically:

```
x = [0,1,3,4,7,10];  
y = [2,4,8,7,10,19];  
plot(x,y,'o'),...  
    xlabel('x','FontSize',24),...  
    ylabel('y','FontSize',24);...  
    axis([0,12,0,25]),...  
    set(gca,'FontSize',20)
```

This script generates the plot of Fig. 1.3. Here the axes were expanded, the data were plotted as individual points, and the font size was increased.

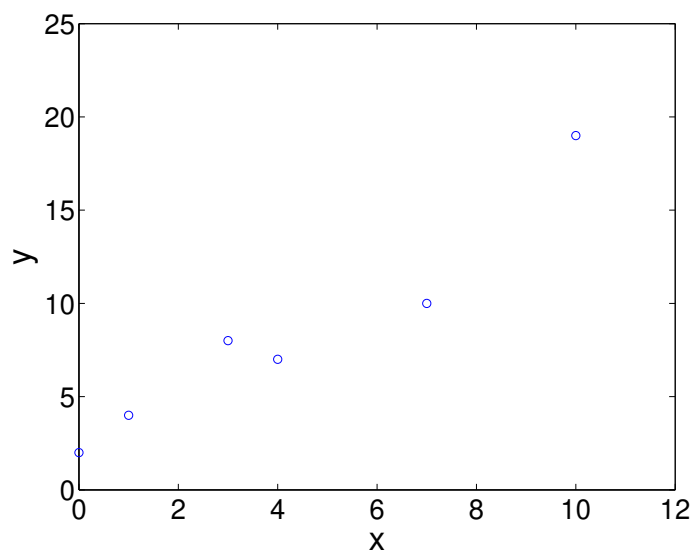


Figure 1.3: Properly formatted plot of some data.

1.5.3 Another bad MATLAB plot

Let us say we want to plot on a log-log scale the two continuous curves

$$y = x^2, \quad y = x^3, \quad x \in [0.01, 1000].$$

Shown next is a MATLAB script which generates a bad plot:

```
clear;
n = 1000;
xmin = 0.01;
xmax = 1000;
dx = (xmax-xmin)/(n-1);
for i=1:n
    x(i) = xmin+(i-1)*dx;
    y1(i) = x(i)^2;
    y2(i) = x(i)^3;
end
loglog(x,y1,'bo',x,y2,'bo')
```

This script generates the plot of Fig. 1.4. Figure 1.4 has the following problems:

- The underlying curves are continuous, but only discrete points have been plotted.
- It is difficult to distinguish the two curves, especially for small x .
- The numbers on the axes are much smaller than the font size of the surrounding text.
- The axes are not labeled.

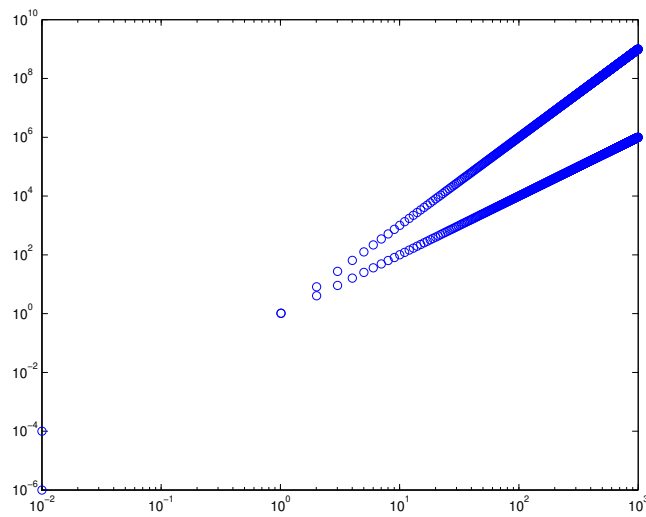


Figure 1.4: Badly formatted plot of the curves $y = x^2$, $y = x^3$.

1.5.4 Another better MATLAB plot

The following MATLAB script gives a better plot of the same two curves:

```
clear;
n = 1000;
xmin = 0.01;
xmax = 1000;
dx = (xmax-xmin)/(n-1);
for i=1:n
    x(i) = xmin+(i-1)*dx;
    y1(i) = x(i)^2;
    y2(i) = x(i)^3;
end
loglog(x,y1,x,y2),...
    xlabel('x','FontSize',24),...
    ylabel('y','FontSize',24)
legend('y=x^2','y=x^3','Location','NorthWest')
set(gca,'FontSize',20)
```

This script generates the plot of Fig. 1.5. Here the data were plotted as continuous curves, the two curves were clearly distinguished and labeled, and the font sizes were increased to be nearly the same as that of the surrounding text.

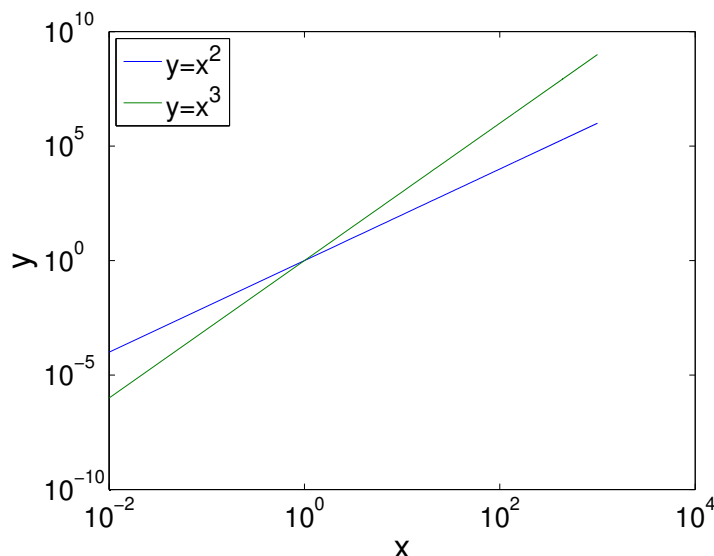


Figure 1.5: Properly formatted log-log plot of the curves $y = x^2$, $y = x^3$.

1.6 Introduction to L^AT_EX

Fortran is an example of a software application that runs in the **UNIX** environment. The text formatter, L^AT_EX, is another example. It is useful for the following reasons:

- It produces the best output of text, figures, and equations of any program I've seen.
- It is machine-independent. It runs on **Linux**, **Macintosh** (see **TeXShop**), and **Windows** (see **TeXnicCenter**) machines. You can e-mail **ASCII** versions of most relevant files.
- It is the tool of choice for many research scientists and engineers. Many journals accept L^AT_EX submissions, and many books are written in L^AT_EX.

Some basic instructions are given next. You can be a little sloppy about spacing. It adjusts the text to look good. You can make the text smaller. You can make the text tiny. You can link to web sites

Skip a line for a new paragraph. You can use italics (*e.g. Computers are everywhere*) or **bold**. Greek letters are a snap: Ψ , ψ , Φ , ϕ . Equations within text are easy—A well known equation for a line is $y = mx + b$. You can also set aside equations like so:

$$m \frac{d^2 x}{dt^2} = \sum F, \quad \text{Newton's second law.} \quad (1.9)$$

Eq. (1.9) is Newton's second law. References³ are available. If you have a postscript file, say **f1.eps**, in the same local directory, you can insert the file as a figure. Figure 1.6 gives

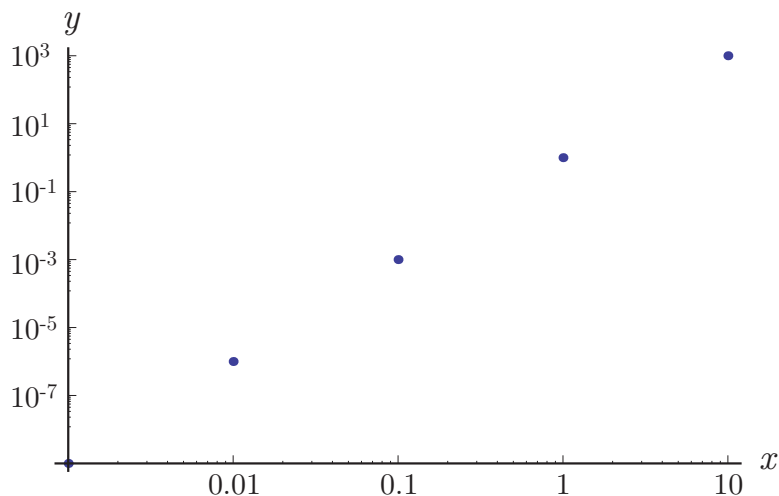


Figure 1.6: Sample log-log plot of some data.

a log-log plot of some data. Figure 1.6 was first created with **MATLAB**, was saved as a **.eps** file, then imported into **Adobe Illustrator** to adjust the fonts to match those of the main text. In this case, it was necessary to download common **L^AT_EX** fonts onto a local machine.

Running L^AT_EX

You can create a **L^AT_EX** file with any text editor (**vi**, **Emacs**, **gedit**, etc.). To get a document, you need to run the **L^AT_EX** application on the text file. The text file must have the suffix “**.tex**” On a **Linux** cluster machine, this is done via the command

```
[powers@darrows2-p ame.20214]$ latex2pdf file.tex
```

This generates **file.pdf**. You should execute this command at least twice to ensure correct figure and equation numbering.

Alternatively, you can use **TeXShop** on a **Macintosh** or **TeXnicCenter** on a **Windows**-based machine.

1.7 Introduction to HTML

To build web pages, one often uses so-called HyperText Markup Language, **HTML**. **HTML** is in many ways like a programming language, and one can do many things with it. Key to the approach is having several publicly available files residing on a disk space which is accessible to the public. On the Notre Dame system, that space exists by default on a server that can be mapped to your local machine.

³Lamport, L., 1986, *L^AT_EX: User's Guide & Reference Manual*, Addison-Wesley: Reading, Massachusetts.

See course TAs for information on how to connect your individual machine to this file space.

The key file to get a web site going must be named `index.html` and must reside in the `www` folder. An example is given in the file `index.html`, listed next.

```
<html>
<head>
<title>My Home Page</title>
</head>
```

This is my home page.

```
<p>
<a href=https://www3.nd.edu/~powers/ame.20214/helloworld.f90>Link</a> to the file
helloworld.f90. Whatever file to which you link must reside in the proper
directory.
```

```
<p>
<a href=https://www.google.com>Link</a> to Google.
```

```
</html>
```

When this file is opened in a browser, one might expect to see something like the screen shot of Fig. 1.7.

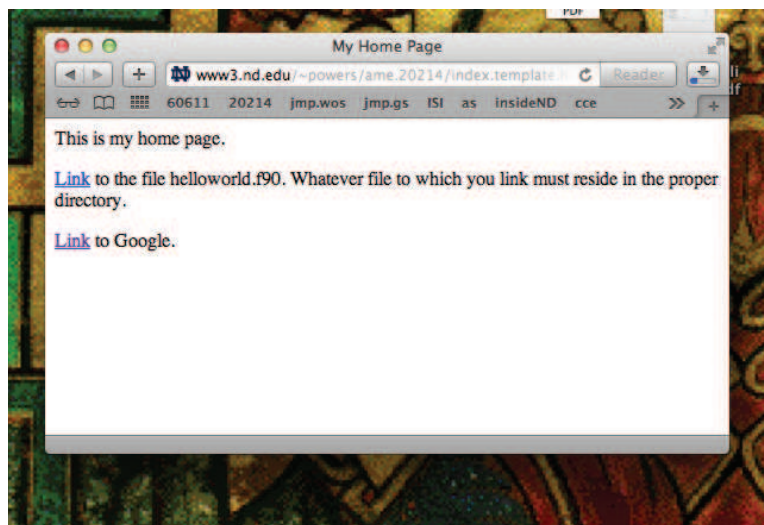


Figure 1.7: Screen shot of a web browser which opened the file `index.html`.

One can note a few points:

- Commands within HTML are begun with the structure `<....>` and ended with the structure `</.....>`.
- Examples include
 - `<html>`, `</html>`, opens and closes the file and identifies it as an html file. The file itself must be named with the `.html` extension.
 - `<head>`, `</head>`, begins and ends the header.
 - `<title>`, `</title>`, begins and ends the title of the header.
 - `<a .....`, `</a>`, begins and ends a link to a page
 - `<p>`, skip a line.
- There are many other commands which can be used, and one can consult one of many references to find information.
- Many applications have WYSIWYG ways to create and store HTML files.

1.8 UNIX

We will focus on the UNIX operating system and its variants. A full understanding of this extensive system is beyond the scope of this course, so we shall focus only on the fundamentals, which are not difficult. A good source for more help with UNIX is found at <https://www.tjhsst.edu/~dhyatt/superap/unixcmd.html>. Here, we simply summarize a few simple commonly used commands relevant to the UNIX operating system.

- `pwd` print working directory. This tells you where you are in the file structure.
- `.` A shortcut for the working directory.
- `..` A shortcut for one directory up the tree structure from the working directory.
- `cd` changes the directory to the so-called home directory.
- `cd directory` This changes directories to the directory named *directory*, which must be a sub-directory of the current working directory.
- `cd ..` This changes directories to one up from the working directory.
- `cd ../../` This changes directories to two up from the working directory.
- `ls` This lists the elements contained within the working directory.

- **ls -lrt** This lists the elements contained within the working directory in detailed form, also giving the ordering by files most recently modified.
- **rm file** This removes the file named *file* from the directory.
- **cp file1 file2** This copies the file named *file1* into the file named *file2*. If *file2* already exists, it is overwritten by *file1*.
- **mv file1 file2** This 1) moves the file named *file1* into the file named *file2*, and 2) removes the file named *file1*. If *file2* already exists, it is overwritten by *file1*.
- **mkdir directory** This creates the directory named *directory* within the working directory.
- **rmdir directory** This removes the directory named *directory*. The directory must be empty in order to remove it.
- **cat file** Concatenate (i.e. print to the screen) the contents of *file*.
- **more file** Print to the screen the contents of *file* one screen at a time and use the space bar to move down in the file.
- **less file** Like **more** except one can use arrows or **j**, **k** keys to navigate up and down in *file*.
- **UNIX command &** Execute the process *UNIX command* in the background so as to keep the terminal window available for user input.
- **UNIX shortcut *** a wildcard character which stands in for arbitrary characters.

One should note that other operating systems such as OS-X or Microsoft Windows have file system devices such as OS-X's "Finder" which give a WYSIWYG view of the UNIX tree structure of organizing files. The two approaches are entirely compatible, and in fact can be applied to arrange the same files. One is simply more visual than the other.

A file can be identified by its local address as long as the user resides in the proper directory. For example to view the file `helloworld.f90`, one first insures it lives within the present directory, and then executes

```
[powers@darrow2-p ame.20214]$ cat helloworld.f90
```

If the user resides in a different directory and still wishes to view the same file without first navigating to that directory, one can use the absolute address, for example

```
[powers@darrow2-p ame.20214]$ cat /afs/nd.edu/users/powers/www/ame.20214/helloworld.f90
```

A highly useful feature of most modern UNIX implementations is the arrow keys' ability to recall previous commands and edit them. It is difficult to display this without an actual demonstration.

Here we illustrate some of the power of UNIX commands. The user may want to copy all of the sample **Fortran** programs from the course home space to the user's personal home space. Since at Notre Dame both the course home space and the user's home space are all within **AFS**, and because the course home space is a public directory, copying is easy and can be done in a few simple commands. The sequence of commands is described as follows:

1. Sign onto a UNIX-based machine.
2. Create a new directory within your home space.
3. Change directories into that new directory.
4. Copy all files with an `.f90` extension into your new directory.
5. List those files to make sure they are present.

A sample set of UNIX commands which achieves this is as follows:

```
[powers@darrows2-p ~]$ mkdir programs
[powers@darrows2-p ~]$ cd programs
[powers@darrows2-p ~/programs]$ ls
[powers@darrows2-p ~/programs]$ cp ~powers/www/ame.20214/2e/*.f90 .
[powers@darrows2-p ~/programs]$ ls -lrt
total 10
-rw-r--r-- 1 powers campus 65 Sep 16 12:41 ch15a.f90
-rw-r--r-- 1 powers campus 894 Sep 16 12:41 ch13d.f90
-rw-r--r-- 1 powers campus 269 Sep 16 12:41 ch13c.f90
-rw-r--r-- 1 powers campus 213 Sep 16 12:41 ch13b.f90
-rw-r--r-- 1 powers campus 210 Sep 16 12:41 ch13a.f90
-rw-r--r-- 1 powers campus 156 Sep 16 12:41 ch12g.f90
-rw-r--r-- 1 powers campus 202 Sep 16 12:41 ch12f.f90
-rw-r--r-- 1 powers campus 217 Sep 16 12:41 ch12e.f90
-rw-r--r-- 1 powers campus 152 Sep 16 12:41 ch12d.f90
-rw-r--r-- 1 powers campus 152 Sep 16 12:41 ch12c.f90
[powers@darrows2-p ~/programs]$
```

Here `mkdir` made the new directory `programs`. We then moved into the directory `programs`, via the `cd` command. We then listed the files within `programs` and found that it was empty. We then copied *all* files with the `.f90` extension into the existing directory, indicated by the “dot”, `[.]`, shorthand for the existing directory. We then re-listed the programs in the directory and found ten new programs listed.

1.9 Online compilers

There exist a variety of open-source web-based compilers of a variety of languages, including **Fortran**. These can be useful for editing and debugging, mainly because the user interface is more intuitive than **UNIX**. There may be challenges in porting input and output files, depending on the web interface chosen. An example interface is found at

http://www.tutorialspoint.com/compile_fortran_online.php.

A screen shot of this web browser-based interface is shown in Fig. 1.8.

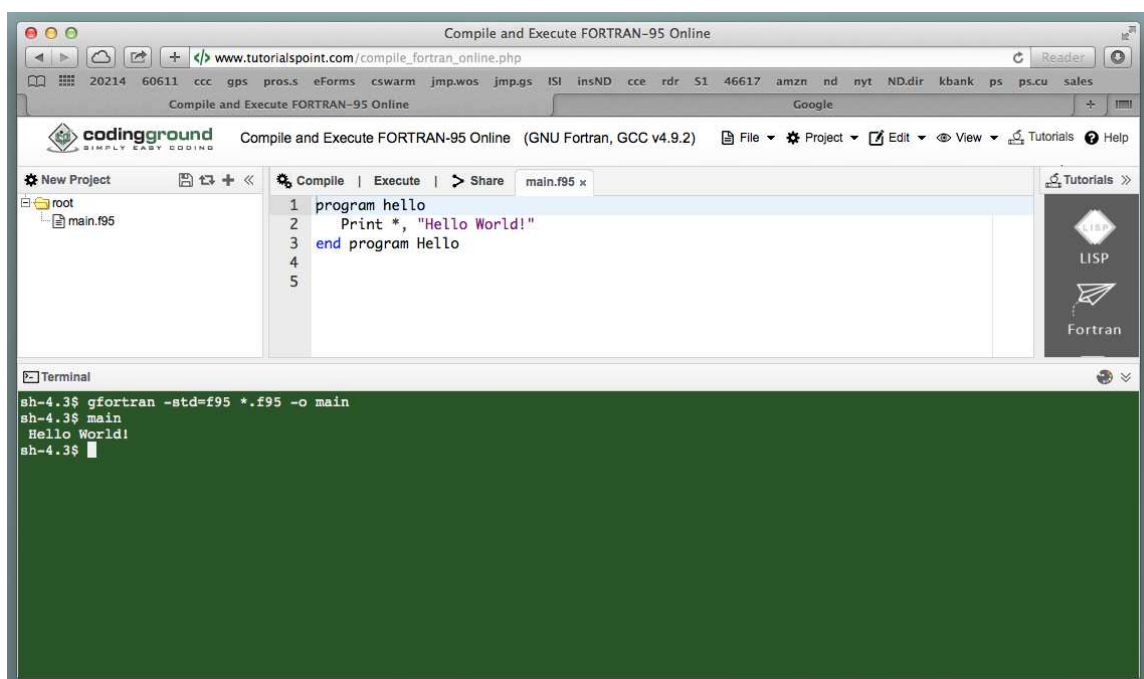


Figure 1.8: Screen shot of a web browser-based Fortran interface.

Chapter 2

Introduction to problem solving

Read CES, Chapter 2.

Here, we summarize a few of the key concepts of this chapter. Problems are often formulated in words, which can be ambiguous. Some problems are amenable to mathematical and computational analysis, and a challenging chore is to translate imprecise words into precise computer code. The code to solve the posed problem is based upon a logical construct known as an *algorithm*. An algorithm is much like a recipe for cooking. An important element of most good scientific computing task involves what is known as *object-oriented programming*. Such an approach relies upon identifying repetitive tasks and writing a single algorithm for a task that is commonly executed as part of solving a larger problem. These building blocks, known as objects can be highly refined and optimized since they will be repeated many times in the course of a calculation. One might imagine developing a sub-algorithm for performing matrix inversion within a larger problem that requires inversion of many matrices of different types. Rather than write separate code for each matrix inversion, one writes one subroutine to invert matrices of arbitrary dimension, provided of course they are invertible. Then that subroutine may be called upon many times within the more general algorithm.

Chapter 3

Introduction to programming languages

Read CES, Chapter 3.

One can consult the text for a fine summary of the history of programming languages.

Chapter 4

Introduction to programming

Read C&S, Chapter 4.

Here, we summarize some important introductory comments regarding programming in Fortran. We adopt many notions and language directly from C&S.

4.1 Statement types

Our language will have a variety of statement types including

1. *data description statements*: These set the type of data, for example, `real`, `integer`, `character`, each of which have different features.
2. *control structures*: These are instructions for how the program is to proceed, for example `do`, `if`.
3. *data processing statements*: These instructions actually process the data. If the data are numbers, a data processing operation might include addition, `x=x+1`. This is a typical data processing statement, which is mathematical, but not ordinary mathematics! It says to take the number `x`, add one to it, and replace the value of `x`. We think of it as an *assignment* statement rather than a mathematical equation. So, if we start with `x` with a value of 5 the processing statement `x=x+1` maps 5 into 6.
4. *input and output (I/O) statements*: These statements allow the program to communicate with the outside world by reading in data and writing out data. An input statement will often have a `read` and an output statement will often have a `write` or `print`.

4.2 Example program

Let us examine C&S's program `ch0401.f90`:

```
program ch0401
!
! This program reads in and prints out a name
!
implicit none
character*20 :: first_name
!
    print *, ' type in your first name.'
    print *, ' up to 20 characters'
    read *, first_name
    print *, first_name
!
end program ch0401
```

Some notes adopted directly from C&S:

- Lower case characters are used. In old versions of **Fortran**, upper case was often the only option. Today, **Fortran** is simply case-insensitive, so it does not matter. Many programmers find it aesthetically pleasing to exclusively use lower case. Our text sporadically reverts to the older convention of upper case. The important thing for you to recognize is that as far as a **Fortran** compiler is concerned the variables are case insensitive; e.g. a variable named **ValuE** is seen by the **Fortran** compiler as identical to **value**.
- Each line is a distinct statement.
- The order of the statements is important. First things are done first. Last things are done last.
- The opening statement is the program name and must appear. It is useful to give the program a meaningful name. Here, we mean the first program of Chapter 4.
- The character “!” denotes a comment statement. They are ignored by the compiler, but are very useful to the program author and readers who are trying to understand your program. It is a good practice for students to insert a surplus of comment statements in their programs.
- The statement **implicit none** is a highly useful statement that should be included in all of your programs. It forces you to declare the type of all variables. And if you make a typographical error in your actual program, you will likely have mis-typed a variable name, and the compiler will identify this error for you. Otherwise, the compiler may simply assign a value of 0 to your mis-typed variable, and this can cause your program to yield results which are difficult to interpret.

- The statement `character*20 :: first_name` declares the variable `first_name` to be a *character* variable. Such a variable is one whose value is an ASCII character. It is not a number. The character here is defined to be up to 20 characters in length. Note that `character*20` works in most older Fortran compilers, but generates a compiler WARNING when the Fortran 2003 standard is imposed with the `ifort` compiler. Note that compiler warnings are not compiler errors; a program which compiles with warnings will still execute. To meet the 2003 standard, one can use the command `character (len=20)`.
- The `print` statement sends output to the screen. It is an I/O statement; specifically, it is an O statement. The `*` indicates the output is unformatted. Later, we shall see how to format output. The `print` statement must be followed by a comma. Items within single right-leaning apostrophes are those which are actually printed to the screen.
- The `read` statement is an I/O statement also; specifically it is an I statement. That is, it requests input from the screen. The `*` indicates the input data is unformatted.
- After the data has been read, it is then outputted in an unformatted fashion to the screen.
- The `end program` statement indicates the end of the program.

4.3 Data types

Some important data types in Fortran are listed next:

- **integer**: e.g. -461; depending on the computer architecture, e.g. 32 or 64-bit machines, the upper and lower bounds of the accepted integers can be extended. For a 64-bit machine, we might say `integer (kind=8)`.
- **real**, a seven significant digit real number, e.g. 1.234567. This is equivalent to the statement `real (kind=4)`.
- **complex**, a seven significant digit complex number, e.g. $1.234567 + 7.654321i$. This is equivalent to the statement `complex (kind=4)`.
- `real (kind=8)`, a fifteen significant digit real number, e.g. 1.23456789012345.
- `complex (kind=8)`, a fifteen significant digit complex number, e.g. $1.23456789012345 + 5.43210987654321i$.
- `real (kind=16)`, a thirty-three significant digit real number, e.g. 1.23456789012345678901234567890123.
This does not work on all compilers. It does work for `ifort`.

- `complex (kind=16)`, a thirty-three significant digit complex number, e.g.
 $1.23456789012345678901234567890123 + 1.23456789012345678901234567890123i$.
 This does not work on all compilers. It does work for `ifort`.
- `character (len=10)`, a ten character string, e.g. `abcdefghij`.
- `logical`, a logical variable which has value of either `.true.` or `.false.`

4.4 Another example

Let's look at a modification of another program from C&S. We name this file `ch0402m.f90`, with the `m` denoting a modification from C&S.

```
[powers@darrow1-p ame.20214]$ cat ch0402m.f90
program ch0402m
!
! This program reads in three numbers and sums
! and averages them.
!
implicit none
integer, parameter:: p=4
real (kind=8) :: n1,n2,n3
real (kind=p) :: average1 , total1
real (kind=2*p) :: average2 , total2
complex :: average3, total3
integer :: n=3

print*, ' type in three numbers.'
print*, ' separated by spaces or commas'
read*,n1,n2,n3
total1 = n1+n2+n3
total2 = n1+n2+n3
total3 = n1+n2+n3
average1 = total1/n
average2 = total2/n
average3 = total3/n
print*, 'single precision total of numbers is ', total1
print*, 'double precision total of numbers is ', total2
print*, 'complex total of numbers is ', total3
print*, 'single precision average of numbers is ',average1
print*, 'double precision average of numbers is ',average2
print*, 'complex average of numbers is ',average3
```

```

end program ch0402m
[powers@darrow1-p ame.20214]$ ifort ch0402m.f90
[powers@darrow1-p ame.20214]$ a.out
  type in three numbers.
  separated by spaces or commas
1.2 10.0 20.122
single precision total of numbers is    31.32200
double precision total of numbers is    31.3220000000000
complex total of numbers is (31.32200,0.0000000E+00)
single precision average of numbers is   10.44067
double precision average of numbers is   10.4406666666667
complex average of numbers is (10.44067,0.0000000E+00)
[powers@darrow1-p ame.20214]$

```

Let us examine some features of this program and its output:

- We have type declarations of `real (kind=4)`, `real (kind=8)`, `complex`, and `integer`. If we do not specify a `kind`, default values are assumed. The type declarations must be at the beginning of the program, before one does any processing of the data.
- One may choose whether or not to initialize variables within the data assignment statement. Here, we chose to initialize `n`, `average1`, `total1`, `average2`, and `total2`. We did not initialize `average3` or `total3`.
- We often choose exponential notation in Fortran. For example

$$1.23 \times 10^4 = 1.23\text{e}4. \quad (4.1)$$

- The first statement which actually processes data is `total1 = n1+n2+n3`, which assigns a value to `total1`.
- Order must be
 - `program` statement,
 - type declarations, e.g. `real`, `integer`,
 - processing and I/O statements,
 - `end program` statement.
- Comments may appear any where after `program` and before `end program`.
- Variable names may be up to 31 characters.
- Line length can be up to 132 characters
- To continue a line, use the `&` character.
- If `implicit none` is *not* used, then variables beginning with the characters `a – h` and `o – z` are by default `real`, and variables beginning with `i – n` are by default `integer`.

Chapter 5

Arithmetic

Read CES, Chapter 5.

Here, we will reinforce some notions introduced earlier regarding arithmetic on a digital computer using **Fortran**. There are several important issues which arise due to precision limits of the machine as well as specific syntax. Many notions are consistent with standard mathematics, but some idiosyncrasies arise because of finite precision as well as the need to have different data types for integers and real numbers.

5.1 Operations

There are five key operations available: To avoid surprises, one should use parentheses liber-

addition	+
subtraction	-
division	/
multiplication	*
exponentiation	**

Table 5.1: **Fortran** arithmetic operators

ally to avoid confusion when applying operations. If parentheses are not used, **Fortran** has a complicated set of priorities for which operation takes precedence. For example, consider the mathematical expression

$$x = \frac{5.(1. + 3.)}{4.} \quad (5.1)$$

The following is the preferred way in **Fortran** to perform this operation:

```
x = (5.*(1.+3.))/4.
```

The following is acceptable as well

```
x = 5*(1.+3.)/4.
```

Turning this around, one might ask what is the mathematical meaning of the Fortran assignment

```
x = 1.+2./3.-4.*5.
```

In Fortran, multiplication and division take priority over addition and subtraction, so the expression can be interpreted mathematically as

$$x = 1. + \frac{2.}{3.} - (4.)(5.) = -\frac{55.}{3.} = -18.33333..... \quad (5.2)$$

5.2 Integer division

Some unusual problems can arise when dividing by integers as shown by the following code, ch0502.f90:

```
[powers@darrow2-p 2e]$ cat ch0502.f90
program ch0502
implicit none
real :: a,b,c
integer :: i
    a = 1.5
    b = 2.0
    c = a / b
    i = a / b
print *,a,b
print *,c
print *,i
end program ch0502
[powers@darrow2-p 2e]$ ifort ch0502.f90
[powers@darrow2-p 2e]$ a.out
    1.500000    2.000000
    0.7500000
           0
[powers@darrow2-p 2e]$
```

Ordinary mathematics would hold that `c` should have the same value as `i`. Both are found by forming `a/b`. But their values are different. This is because we have declared `c` to be `real` and `i` to be `integer`. The number `i` must be an integer, and the rule applied was to round down to the nearest integer, which here was 0.

Another example is seen next in ch0503.f90:

```
[powers@darrows2-p 2e]$ cat ch0503.f90
program ch0503
implicit none
integer :: I,J,K
real :: Answer
  I = 5
  J = 2
  K = 4
  Answer = I / J * K
  print *,I
  print *,J
  print *,K
  print *,Answer
end program ch0503
[powers@darrows2-p 2e]$ ifort ch0503.f90
[powers@darrows2-p 2e]$ a.out
      5
      2
      4
 8.000000
[powers@darrows2-p 2e]$
```

Here, ordinary mathematics would tell us that

$$Answer = \frac{5}{2}4 = 10. \quad (5.3)$$

Note that *Answer* should in fact be an integer. But our **Fortran** code returned the nearby value of *Answer* = 8. This is because the code first divided 5 by 2, rounding to 2. It then multiplied by 4 to get 8. Generally, this is undesirable. In such cases, one should take care to see that all such calculations are done with **REAL** variable specifications.

Why then use an **integer** specification at all? There are several reasons:

- The integer representation uses less data storage.
- The integer representation is exact, which is often an advantage.
- The integer representation can lead some operations to execute more efficiently, e.g. x^2 when x is real is executed faster and with more accuracy as `x**2` than `x**2.0`.

Let us look at one final example, found in `ch5a.f90`:

```
[powers@darrows2-p 2e]$ cat ch5a.f90
program ch5a
integer::i,j
! beginning of the program
```


5.4 Overflow and underflow limits

With a 32-bit computer architecture, we have $q = 31$, and the largest integer that can be represented is

$$2^{31} - 1 = 2147483647 = 2.147483647 \times 10^9$$

With the 64-bit architecture more common in modern computers, we have $q = 63$, and the largest integer that can be represented in principle is

$$2^{63} - 1 = 9223372036854775807 = 9.223372036854775807 \times 10^{18}$$

Testing the `ifort` compiler on `darrow.cc.nd.edu` however shows that the largest integer in default operating mode is actually 2147483647, when using default compilation. However, one can invoke 64 bit integers via the compilation command `ifort -integer-size 64`. For example, we find for the program `ch5d.f90`

```
[powers@darrow1-p 2e]$ cat ch5d.f90
program test
implicit none
integer :: i
print*,huge(i)
end program test
[powers@darrow1-p 2e]$ ifort -integer-size 64 ch5d.f90
[powers@darrow1-p 2e]$ a.out
9223372036854775807
```

that we recover the maximum integer limit for a 64 bit architecture. There is an equivalent syntax for `gfortran` to invoke 64 bit integers:

```
[powers@darrow1-p 2e]$ cat ch5d.f90
program test
implicit none
integer :: i
print*,huge(i)
end program test
[powers@darrow1-p 2e]$ gfortran -fdefault-integer-8 ch5d.f90
[powers@darrow1-p 2e]$ a.out
9223372036854775807
```

For real numbers, there are similar limits. While actual values can vary, C&S report typical real numbers in a 32-bit architecture machine must be such that

$$|x| \in [0.3 \times 10^{-38}, 1.7 \times 10^{38}].$$

5.5 Assignment of undefined variables

Lastly, we note that variables can be defined and undefined in **Fortran**. The compiler will not view the existence of undefined variables as a problem. However, if one attempts to use an undefined variable, the program may respond in an unusual manner. Let us examine how the program `ch5c.f90` behaves under the `ifort` compiler.

A listing of the program, its compilation with `ifort` and execution are given next:

```
[powers@darrow2-p 2e]$ cat ch5c.f90
program ch5c
implicit none
integer (kind=8) :: n=1, m      !n is initialized, m is not
real (kind=8) :: x=1., y        !x is initialized, y is not
open (unit=10,file='out.txt') !open an output file
print*, 'n= ', n
print*, 'm= ', m
print*, 'n*m= ', n*m
write(10,*) n,m,n*m             !write some output to the output file
print*, 'x= ', x
print*, 'y= ', y
print*, 'x*y= ', x*y
write(10,*) x,y,x*y             !write some output to the output file
end program ch5c
[powers@darrow2-p 2e]$ ifort ch5c.f90
[powers@darrow2-p 2e]$ a.out
n=          1
m=          0
n*m=        0
x=  1.0000000000000000
y=  0.0000000000000000E+000
x*y=  0.0000000000000000E+000
[powers@darrow2-p 2e]$ cat out.txt
          1          0          0
 1.0000000000000000  0.0000000000000000E+000  0.0000000000000000E+000
[powers@darrow2-p 2e]$
```

Note we did not initialize the integer `m` or the real `y`. In both cases, `ifort` assigned them a value of zero. This can cause surprises if it is not intended for the variables to take on a value of zero.

As an aside, we also wrote the output to a file. To do so we had to open the file, using the command

```
open (unit=10,file='out.txt')
```

This opens a file which we have named `out.txt`. It will be a plain ASCII text file which we could edit with any text editor, like `gedit`. To populate this file, we must write to it. We have chosen to write twice to the file via the `write` commands

```
write(10,*) n,m,n*m  
write(10,*) x,y,x*y
```

The number 10 is linked via the `open` statement to the file `out.txt`. After execution, we find that `out.txt` has been created and populated with output.

Chapter 6

Arrays 1

Read CES, Chapter 6.

6.1 Dimension of an array

We often write lists. Often those lists have structure. Sometimes the structure is one-dimensional; other times it is two-dimensional. It can also be higher dimensional. These lists can be structured into *arrays*. One-dimensional arrays are known as *vectors*. Note that these vectors will usually not have our typical geometrical interpretation from ordinary Euclidean geometry and Gibbs vector mathematics; however, modern mathematics shows that the traditional concepts of Euclid and Gibbs can be extended. In short, a vector Gibbs would be comfortable with might be

$$\mathbf{x} = \begin{pmatrix} 1 \\ 5 \\ 9 \end{pmatrix}, \quad (6.1)$$

which could be interpreted as $\mathbf{x} = 1\mathbf{i} + 5\mathbf{j} + 9\mathbf{k}$, a vector in three-dimensional space. In matrix notation, it has dimension 3×1 in the space of two-dimensional arrays. Gibbs would call it a vector in a three-dimensional space because the first index is 3. We instead take the array dimension to be the number of non-unity dimensions. That is, we might also think of the vector of Eq. (6.1) as of dimension $3 \times 1 \times 1 \times 1$ in the space of four-dimensional arrays. But since only one of the dimensions is non-unity, it is really a one-dimensional array, or a vector. Thus, our so-called “one-dimensional” vectors could take the form

$$\mathbf{x} = \begin{pmatrix} 1 \\ 5 \\ 9 \\ -4 \\ 7 \\ 18 \\ -74 \end{pmatrix}. \quad (6.2)$$

The vector of Eq. (6.2) is of dimension 7×1 and is also a one-dimensional array. Note that a *scalar* ϕ might be thought of as a 1×1 array (or for that matter a $1 \times 1 \times 1$ three-dimensional array):

$$\phi = (9). \quad (6.3)$$

A scalar can be thought of as a zero-dimensional array, since it has no non-unity dimensions.

Two-dimensional arrays are known as *matrices*. An example might be

$$\mathbf{A} = \begin{pmatrix} 1 & 6 & 8 & 9 \\ 2 & -4 & 2 & 10 \\ -9 & 0 & 1 & 5 \end{pmatrix}. \quad (6.4)$$

This array has dimension 3×4 . In four-dimensional space, it could be represented as $3 \times 4 \times 1 \times 1$ array. Because only two of the dimensions are non-unity, we take $\mathbf{A}$ to be a two-dimensional array.

6.2 One-dimensional arrays from time series

Let us consider an important kind of one-dimensional array in engineering mathematics, a time series obtained by sampling an equation. This will be useful for, among other things, generating plots. Let us imagine for example that the temperature in South Bend on a typical September day is known to be well modeled by the following equation:

$$T(t) = a - b \sin\left(\frac{2\pi t}{c}\right), \quad (6.5)$$

where T is the mean temperature in K, b is the amplitude of the temperature fluctuation in K, c is the period of the oscillation in hours, and t is the time in hours. We could imagine that $a = 270$ K, $b = 10$ K, $c = 24$ hr are reasonable values. Thus, we have

$$T(t) = (270 \text{ K}) - (10 \text{ K}) \sin\left(\frac{2\pi t}{24 \text{ hr}}\right). \quad (6.6)$$

6.2.1 Fixed array size

Now we may want to plot this data. Nearly all plotting routines require a finite number of sample points. It is easily verified that if we sample the temperature every two hours, we will generate the set of data points, given in Table 6.1.

Now we can imagine a one-dimensional array, or vector, of values for both T and for t . Here, we have 13 values of both t and T . We might call the discrete values t_i and T_i , with $i = 1, \dots, 13$. So for $i = 5$, we have

$$i = 5, \quad (6.7)$$

$$t_5 = 8 \text{ hr} \quad (6.8)$$

$$T_5 = 261.34 \text{ K}. \quad (6.9)$$

i	t (hr)	T (K)
1	0	270.00
2	2	265.00
3	4	234.00
4	6	260.00
5	8	261.34
6	10	265.00
7	12	270.00
8	14	275.00
9	16	278.66
10	18	280.00
11	20	278.66
12	22	275.00
13	24	270.00

Table 6.1: Temperature versus time at discrete times

We can write a Fortran program to generate this table, embodied in `ch6a.f90`:

```
[powers@darrow2-p ame.20214]$ cat ch6a.f90
program temperature
implicit none
integer (kind=8) :: i
real (kind=8), dimension(1:13):: t,temp
real (kind=8), parameter :: a=270., b=10., c = 24.
real (kind=8) :: pi
pi = 4.*atan(1.)
print*, 'pi = ', pi
do i=1,13
  t(i) = 2.*(i-1)
  temp(i) = a - b*sin(2.*pi*t(i)/c)
  print*,i,t(i),temp(i)
end do
end program temperature
[powers@darrow2-p ame.20214]$ ifort ch6a.f90
[powers@darrow2-p ame.20214]$ a.out
pi =      3.14159274101257
      1  0.000000000000000E+000    270.000000000000
      2  2.000000000000000    264.99999873816
      3  4.000000000000000    261.339745816451
      4  6.000000000000000    260.000000000000
      5  8.000000000000000    261.339746253565
      6 10.000000000000000    265.00000630920
      7 12.000000000000000    270.00000874228
      8 14.000000000000000    275.00000883287
      9 16.000000000000000    278.660254620663
     10 18.000000000000000    280.000000000000
     11 20.000000000000000    278.660253309321
     12 22.000000000000000    274.99998611977
     13 24.000000000000000    269.99998251544
[powers@darrow2-p ame.20214]$
```

Let us note several features of this program:

- We are attempting to do double precision calculations, with `kind = 8` for real numbers.
- We are tempted to use `t` for time and `T` for temperature. That would be a mistake, since any Fortran compiler would interpret them as identical variables. So we use `t` for time and `temp` for temperature.
- The `dimension` statement tells us that both `t` and `temp` are one-dimensional arrays, whose first index is 1 and last index is 13. This highly general statement is useful

because it reminds us of our full range of options. Alternatively, we could have used the simpler

```
real (kind=8), dimension(13):: t,temp
```

where the first index is by default 1. Or we could have used the simpler still

```
real (kind=8) :: t(13),temp(13)
```

The second form is common, and tells us both `t` and `temp` are one-dimensional arrays of length 13.

- We set parameter values for `a`, `b`, and `c`. It is good practice to leave as much generality as possible in your program.
- While we could have defined π , we chose to let the computer do it, since we know from trigonometry that

$$\frac{\pi}{4} = \tan^{-1} 1. \quad (6.10)$$

Note that in **Fortran**, the function $\tan^{-1}$ is given by `atan`. We printed π to the screen to make sure our formula was correct.

- We now have an all-important `do` loop to initialize values for $t_i = t(i)$ and $T_i = temp(i)$.
- It is good practice to indent items within the `do` loop so that it is clear to the reader what is in and outside of the loop.
- We print values of `t(i)` and `temp(i)` to the screen.
- We note that we seem to have only achieved single precision results! Note that at $t = 2$, we should have

$$T = 270 - 10 \sin \left(\frac{2\pi(2)}{24} \right), \quad (6.11)$$

$$= 270 - 10 \sin \left(\frac{\pi}{6} \right), \quad (6.12)$$

$$= 270 - 10 \left(\frac{1}{2} \right), \quad (6.13)$$

$$= 265. \quad (6.14)$$

The result should be nearly exactly 265, but we see precision errors in the last six digits, namely `temp(2) = 264.999999873816`. Why? If we truly want double precision accuracy, we must take care that every constant is truly double precision.

The following modification, `ch6b.f90`, allows for single, double, and quad precision, which we exercise here for double precision:

```
[powers@darrow1-p ame.20214]$ cat ch6b.f90
program temperature
implicit none
integer, parameter :: p=8
integer (kind=8) :: i
real (kind=p), dimension(1:13):: t,temp
real (kind=p), parameter :: a=270._p, b=10._p, c = 24._p
real (kind=p) :: pi
pi = 4._p*atan(1._p)
print*, 'pi = ', pi
do i=1,13
    t(i) = 2._p*(i-1)
    temp(i) = a - b*sin(2._p*pi*t(i)/c)
    print*,i,t(i),temp(i)
end do
end program temperature
[powers@darrow1-p ame.20214]$ ifort ch6b.f90
[powers@darrow1-p ame.20214]$ a.out
pi =      3.14159265358979
      1  0.000000000000000E+000    270.000000000000
      2  2.000000000000000        265.000000000000
      3  4.000000000000000        261.339745962156
      4  6.000000000000000        260.000000000000
      5  8.000000000000000        261.339745962156
      6 10.000000000000000        265.000000000000
      7 12.000000000000000        270.000000000000
      8 14.000000000000000        275.000000000000
      9 16.000000000000000        278.660254037844
     10 18.000000000000000        280.000000000000
     11 20.000000000000000        278.660254037844
     12 22.000000000000000        275.000000000000
     13 24.000000000000000        270.000000000000
[powers@darrow1-p ame.20214]$
```

Note:

- The parameter `p` sets the precision; here, we have `p = 8`, for double precision.
- Every numerical constant is defined in terms of its precision, e.g.
`4._p`
- The results now have true double precision accuracy.

6.2.2 Variable array size via parameter declaration

Lastly, let us generalize the program for a variable number of sample points. This is actually the preferred way to write the program. Here, we choose to stay in single precision and give the program listing of `ch6c.f90`, compilation, and execution as

```
[powers@darrows2-p ame.20214]$ cat ch6c.f90
program temperature
implicit none
integer, parameter :: imax=7
integer (kind=4) :: i
real (kind=4), dimension(1:imax):: t,temp
real (kind=4), parameter :: a=270., b=10., c = 24., tmax = 24.
real (kind=4) :: pi, dt
pi = 4.*atan(1.)
print*, 'pi = ', pi
dt = tmax/(real(imax)-1.)
do i=1,imax
    t(i) = dt*(i-1)
    temp(i) = a - b*sin(2.*pi*t(i)/c)
    print*,i,t(i),temp(i)
end do
end program temperature
[powers@darrows2-p ame.20214]$ ifort ch6c.f90
[powers@darrows2-p ame.20214]$ a.out
pi =      3.141593
      1  0.0000000E+00    270.0000
      2   4.000000      261.3398
      3   8.000000      261.3398
      4  12.00000      270.0000
      5  16.00000      278.6602
      6  20.00000      278.6602
      7  24.00000      270.0000
[powers@darrows2-p ame.20214]$
```

Note

- The program structure is similar to the two previous.
- We now have `imax` as an adjustable parameter. Here, we chose a value of `imax = 7`.
- The array size is allowed to vary when the parameter statement gives the actual size of the array.

- We now have a variable time step `dt` which will have to be calculated. Mathematically, the formula is

$$\Delta t = \frac{t_{max}}{i_{max} - 1}. \quad (6.15)$$

So, if we just have one time step, $i_{max} = 2$ and $\Delta t = t_{max}$. If we have two times steps, $i_{max} = 3$, and $\Delta t = t_{max}/2$. For our $i_{max} = 7$, we have six time steps with $\Delta t = t_{max}/6 = (24 \text{ hr})/6 = 4 \text{ hr}$.

- We calculated `dt`, taking care to convert integers to reals when dividing. The command `real(imax)` simply converts `imax` to its single precision real number equivalent. Recall if we had wanted a double precision real, we would have used the command `real(imax,8)`.
- The parameter `i` in the do loop now ranges from `i = 1` to `i = imax`.

6.3 Simple writing to files and MATLAB plotting

We often want to plot our data. An easy way to do this is to `write` our output data to a file, and then import and plot that data with another application. We shall demonstrate this with `MATLAB`. We first modify our program slightly so that it opens the file `temperature.out` and writes to it. This is achieved in `ch6d.f90`:

```
[powers@darrow2-p ame.20214]$ cat ch6d.f90
program temperature
implicit none
integer, parameter :: imax=13
integer (kind=4) :: i
real (kind=4), dimension(1:imax):: t,temp
real (kind=4), parameter :: a=270., b=10., c = 24., tmax = 24.
real (kind=4) :: pi, dt
open(unit=20,file='temperature.out')
pi = 4.*atan(1.)
print*, 'pi = ', pi
dt = tmax/(real(imax)-1.)
do i=1,imax
  t(i) = dt*(i-1)
  temp(i) = a - b*sin(2.*pi*t(i)/c)
  print*,i,t(i),temp(i)
  write(20,*)t(i),temp(i)
end do
end program temperature
[powers@darrow2-p ame.20214]$ ifort ch6d.f90
[powers@darrow2-p ame.20214]$ a.out
```

```

pi =    3.141593
      1  0.0000000E+00    270.0000
      2  2.000000      265.0000
      3  4.000000      261.3398
      4  6.000000      260.0000
      5  8.000000      261.3398
      6 10.00000      265.0000
      7 12.00000      270.0000
      8 14.00000      275.0000
      9 16.00000      278.6602
     10 18.00000      280.0000
     11 20.00000      278.6602
     12 22.00000      275.0000
     13 24.00000      270.0000
[powers@darrow2-p ame.20214]$

```

Now we next write and execute a simple MATLAB script, taking care to see it resides in the same directory as our file `temperature.out`. The MATLAB script is given in `ptemperature.m`:

```

clear;
load('temperature.out')
t = temperature(:,1);
temp = temperature(:,2);
plot(t,temp,'-o'),...
    xlabel('t (hr)','FontSize',24),...
    ylabel('T (K)','FontSize',24);...
    set(gca,'FontSize',20)

```

The contents of `temperature.out` are loaded into MATLAB via the `load` command. The first column is assigned to time `t` and the second column is assigned to temperature `temp`. We then make a plot, taking care to include larger than default font sizes, as well as small circles for the individual points, and axes labels with units. The command

```
'FontSize', 24
```

within `xlabel` and `ylabel`, changed the font size for the axes labels. The command

```
set(gca,'FontSize',20)
```

changed the font size for the numbers on the axes. The figure is shown in Fig. 6.1. While the MATLAB-generated plot is not bad, it has some minor flaws. The exact solution is not shown, only a straight line interpolation between points. It is possible to reset MATLAB defaults to avoid these flaws. Alternatively, we can import the `.eps` file into another application, such as Adobe Illustrator, and edit the plot with this highly potent tool. We show in Fig. 6.2 a plot of the continuous function of Eq. (6.6) and the discrete values at the sample points, after further processing with Adobe Illustrator.

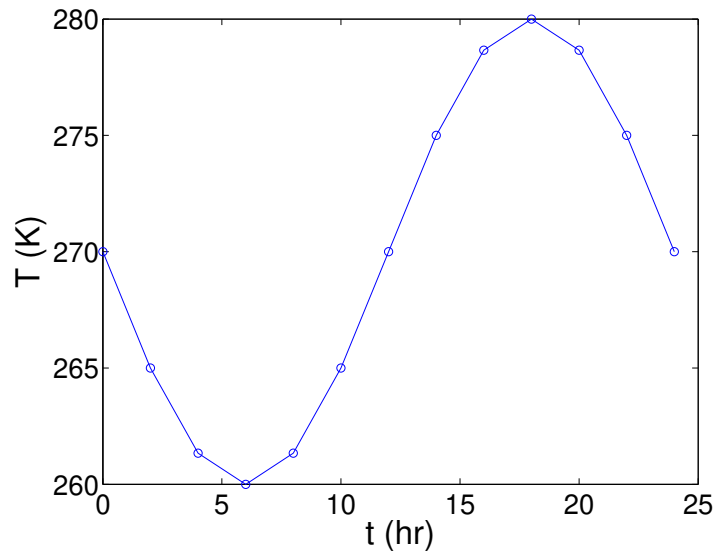


Figure 6.1: Temperature versus time; figure directly from MATLAB, saved in .eps format.

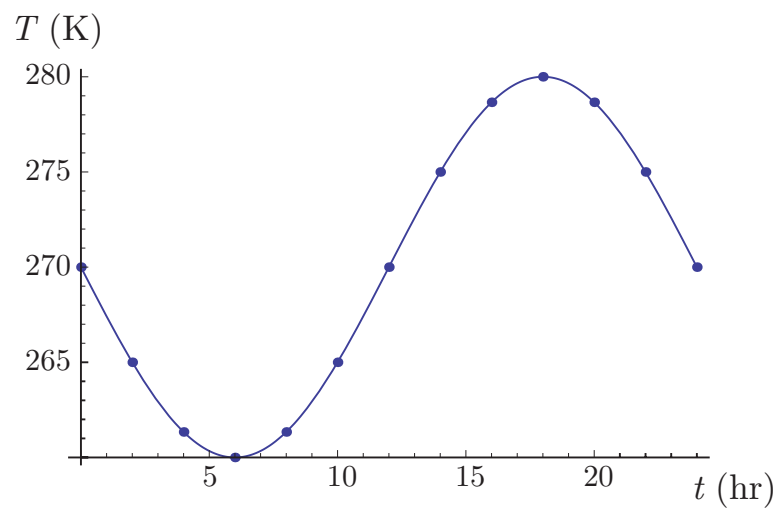


Figure 6.2: Continuous and discrete values of $T(t)$; plot improved with Adobe Illustrator.

Chapter 7

Arrays 2

Read CES, Chapter 7.

7.1 Varying the array size at run time

A significant advantage of modern Fortran is its ability to have memory sizes which can be set outside of the program. The key to this is the `allocatable` attribute in a declaration statement. For example, let us modify a program from the previous chapter, `ch6c.f90`, so that the user can input the value of the number of time steps from outside of the program. Here is the structure of the modified program, named `ch7a.f90`.

```
[powers@darrows1-p ame.20214]$ cat ch7a.f90
program temperature
implicit none
integer :: imax
integer (kind=4) :: i
real (kind=4), dimension(:), allocatable :: t,temp          ! modified
real (kind=4), parameter :: a=270., b=10., c = 24., tmax = 24.
real (kind=4) :: pi, dt
print*, ' Enter number of data sampling points' ! new data entry prompt
read*,imax                                     ! new data entry
allocate(t(1:imax))                           ! new statement for array size
allocate(temp(1:imax))                        ! new statement for array size
pi = 4.*atan(1.)
print*, 'pi = ', pi
dt = tmax/(real(imax)-1.)
do i=1,imax
    t(i) = dt*(i-1)
    temp(i) = a - b*sin(2.*pi*t(i)/c)
    print*,i,t(i),temp(i)
```

```

    end do
end program temperature
[powers@darrow1-p ame.20214]$ ifort ch7a.f90
[powers@darrow1-p ame.20214]$ a.out
Enter number of data sampling points
7
pi =    3.141593
      1  0.0000000E+00    270.0000
      2   4.000000      261.3398
      3   8.000000      261.3398
      4  12.00000      270.0000
      5  16.00000      278.6602
      6  20.00000      278.6602
      7  24.00000      270.0000
[powers@darrow1-p ame.20214]$ a.out
Enter number of data sampling points
13
pi =    3.141593
      1  0.0000000E+00    270.0000
      2   2.000000      265.0000
      3   4.000000      261.3398
      4   6.000000      260.0000
      5   8.000000      261.3398
      6  10.00000      265.0000
      7  12.00000      270.0000
      8  14.00000      275.0000
      9  16.00000      278.6602
     10  18.00000      280.0000
     11  20.00000      278.6602
     12  22.00000      275.0000
     13  24.00000      270.0000
[powers@darrow1-p ame.20214]$

```

Note the following:

- The variables `t` and `temp` have not been dimensioned, but have been declared as arrays with no dimensions as of yet.
- The dimensions of the arrays, `imax` is read in from the screen and given as direct input by the user.
- Once read in, the array dimensions of `t` and `temp` are assigned with the `allocate` function.

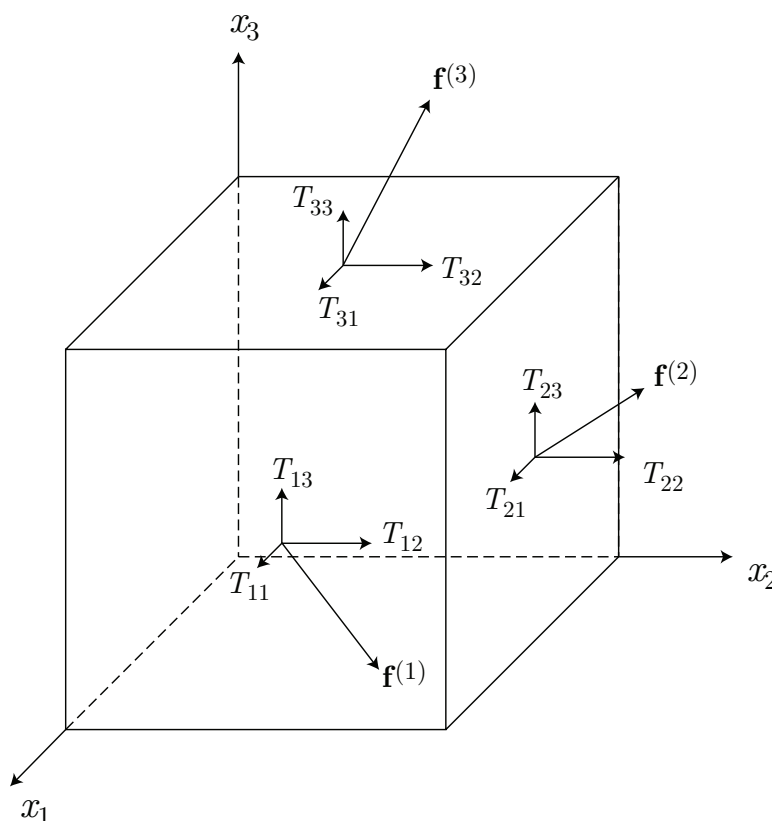


Figure 7.1: Sketch of the components of a stress tensor on a material element.

- The program is compiled once and executed twice, each time for a different value of `imax`. The first value is 7 and the second value is 13; no recompilation is necessary.

7.2 Higher dimension arrays

We often have need to consider arrays of higher dimension than unity. Most common is the use of two-dimensional arrays, or matrices, which in general are of dimension $n \times m$, where n and m are integers. If the two-dimensional array is of dimension $n \times n$, it is said to be *square*.

Let us consider the mathematical operation of matrix-vector multiplication to illustrate the use of two-dimensional arrays in **Fortran**. We can in fact consider a physically motivated problem where matrices arise. In solid mechanics, we are often given the *stress tensor*, really a 3×3 matrix to help describe the forces on a body.

We show in Fig. 7.1 a sketch of a cube in three space where the ordinary (x, y, z) coordinates are replaced by (x_1, x_2, x_3) . On each face of the cube is a vector $\mathbf{f}$, which represents a so-called surface traction force. The surface traction force has units of N/m^2 and rep-

resents a force per area. Each of the surface tractions can be represented in terms of its three-dimensional Cartesian components, e.g.

$$\mathbf{f}^{(1)} = T_{11}\mathbf{e}_1 + T_{12}\mathbf{e}_2 + T_{13}\mathbf{e}_3, \quad (7.1)$$

$$\mathbf{f}^{(2)} = T_{21}\mathbf{e}_1 + T_{22}\mathbf{e}_2 + T_{23}\mathbf{e}_3, \quad (7.2)$$

$$\mathbf{f}^{(3)} = T_{31}\mathbf{e}_1 + T_{32}\mathbf{e}_2 + T_{33}\mathbf{e}_3. \quad (7.3)$$

Here $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ are unit vectors in the 1, 2, and 3 directions, respectively. We see a structure in these equations, and in fact are inclined to organize the values of T_{ij} into a two-dimensional array, defined as the *stress tensor* $\mathbf{T}$:

$$\mathbf{T} = T_{ij} = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{pmatrix} \quad (7.4)$$

They are equivalent to the familiar $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$. Now, detailed analysis reveals that the surface traction force $\mathbf{f}$ on an *arbitrarily* inclined surface, with unit surface normal $\mathbf{n}$ is given by the following:

$$\mathbf{f} = \mathbf{T}^T \cdot \mathbf{n}. \quad (7.5)$$

Here the superscript T denotes a matrix transpose. In component form, we could say

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} T_{11} & T_{21} & T_{31} \\ T_{12} & T_{22} & T_{32} \\ T_{13} & T_{23} & T_{33} \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} \quad (7.6)$$

In summation form, this is equivalent to

$$f_i = \sum_{j=1}^3 n_j T_{ji}. \quad (7.7)$$

For example, for $i = 1$, we have

$$f_1 = n_1 T_{11} + n_2 T_{21} + n_3 T_{31}. \quad (7.8)$$

Recalling that any vector $\mathbf{v}$ can be normalized by dividing by its magnitude, we can say

$$\mathbf{n} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + v_3\mathbf{e}_3}{\sqrt{v_1^2 + v_2^2 + v_3^2}}. \quad (7.9)$$

Let us build a program which assuming a known value of a stress tensor, takes an arbitrary vector $\mathbf{v}$ as input, converts that vector to a unit vector $\mathbf{n}$, and then finds the associated surface traction vector $\mathbf{f}$. This will be aided greatly by the use of two-dimensional arrays. However, before entering into the actual program, let us outline our algorithm:

- initialize the two-dimensional array for the stress tensor $\mathbf{T}$.

- enter three real numbers for the components of $\mathbf{v}$.
- convert $\mathbf{v}$ into a unit normal vector $\mathbf{n}$.
- perform the matrix-vector multiplication of Eq. (7.7).
- output the results.

For our program, we will choose the stress tensor to be

$$\mathbf{T} = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{pmatrix} = \begin{pmatrix} 1 & 5 & 3 \\ -2 & 4 & 0 \\ -1 & 2 & 3 \end{pmatrix}. \quad (7.10)$$

Note that our stress tensor is asymmetric. Usually, stress tensors are symmetric, but if so-called “body couples,” which can be induced by magnetic forces, are present, the stress tensor may be asymmetric.

Note that if we select a unit vector aligned with one of the coordinate axes, say $\mathbf{n} = (1, 0, 0)^T$, we find

$$\mathbf{f}^{(1)} = \mathbf{T}^T \cdot \mathbf{n} = \begin{pmatrix} T_{11} & T_{21} & T_{31} \\ T_{12} & T_{22} & T_{32} \\ T_{13} & T_{23} & T_{33} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} T_{11} \\ T_{12} \\ T_{13} \end{pmatrix}, \quad (7.11)$$

$$= \begin{pmatrix} 1 & -2 & -1 \\ 5 & 4 & 2 \\ 3 & 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 3 \end{pmatrix}. \quad (7.12)$$

This unit normal vector in the 1 direction selected the components associated with the 1 face of the unit cube.

If we select a direction, say $\mathbf{v} = (1, 2, 3)^T$, we then would find

$$|\mathbf{v}| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}, \quad (7.13)$$

and

$$\mathbf{n} = \begin{pmatrix} \frac{1}{\sqrt{14}} \\ \frac{2}{\sqrt{14}} \\ \frac{3}{\sqrt{14}} \end{pmatrix} = \begin{pmatrix} 0.2673 \\ 0.5345 \\ 0.8018 \end{pmatrix}. \quad (7.14)$$

The surface traction force associated with the surface whose unit normal is that just given is

$$\mathbf{f} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} 1 & -2 & -1 \\ 5 & 4 & 2 \\ 3 & 0 & 3 \end{pmatrix} \begin{pmatrix} 0.2673 \\ 0.5345 \\ 0.8018 \end{pmatrix} = \begin{pmatrix} -1.6036 \\ 5.0780 \\ 3.2071 \end{pmatrix}. \quad (7.15)$$

Here is a Fortran program, `ch7b.f90`, and its execution for the two values just shown:

```

[powers@remote101 ame.20214]$ cat ch7b.f90
program stress
! program to calculate the surface
! traction force, f, given the stress
! tensor, T, and a user-input
! direction vector, v

implicit none
integer :: i,j
real, dimension(1:3,1:3) :: T
real, dimension(1:3) :: v,n,f
real :: magv
T(1,1) = 1.
T(1,2) = 5.
T(1,3) = 3.
T(2,1) = -2.
T(2,2) = 4.
T(2,3) = 0.
T(3,1) = -1.
T(3,2) = 2.
T(3,3) = 3.
! set values of stress tensor T

print*, 'enter v(1), v(2), v(3)'
! enter values for the directions
read*,v(1),v(2),v(3)
magv = sqrt(v(1)**2 + v(2)**2 + v(3)**2)
! calculate the magnitude of v
do i=1,3
! calculate the normal vector n
    n(i) = v(i)/magv
end do
print*, 'n= ',n
! print the normal vector, n
do i=1,3
! perform the matrix multiplication
    f(i)=0.
! initialize f(i) to zero
    do j=1,3
        f(i) = f(i) + n(j)*T(j,i)
! compute the sum
    end do
end do
print*, 'f = ',f
! print the surface traction, f
end program stress

[powers@remote101 ame.20214]$ ifort ch7b.f90
[powers@remote101 ame.20214]$ a.out
enter v(1), v(2), v(3)
1 0 0
n= 1.000000 0.000000E+00 0.000000E+00
f = 1.000000 5.000000 3.000000
[powers@remote101 ame.20214]$ a.out
enter v(1), v(2), v(3)

```

```

1 2 3
n=   0.2672612      0.5345225      0.8017837
f =  -1.603567      5.077964      3.207135
[powers@remote101 ame.20214]$

```

Note the following:

- The two-dimensional array for the stress tensor is T. Its declaration statement is

```
real, dimension(1:3,1:3) :: T
```

Two completely equivalent statements are

```
real, dimension(3,3) :: T
real :: T(3,3)
```

- We used `do` loops to organize two important sets of repetitive calculations: 1) the minorly repetitive calculation of the normal vector $\mathbf{n}$, and 2) the more repetitive calculation of the matrix-vector product $\mathbf{T}^T \cdot \mathbf{n}$. The second calculation employed *two nested do loops*. Note that the critical component of the calculation,

```
f(i) = f(i) + n(j)*T(j,i)
```

is the Fortran equivalent of the mathematical operation of Eq. (7.7), $f_i = \sum_{j=1}^3 n_j T_{ji}$.

7.3 Two-dimensional arrays of variable size

We can have two-dimensional arrays of variable size. It is also possible to let that size be read outside of the program. An example is given in `ch7d.f90`:

```

[powers@remote105 ame.20214]$ cat ch7d.f90
program twod
implicit none
integer :: imax,jmax
integer (kind=4) :: i,j
integer (kind=4), dimension(:,:), allocatable :: x !dimension variable array
print*, ' Enter the two lengths of the array'      !variable array size
read*,imax,jmax
allocate(x(1:imax,1:jmax))
do i=1,imax
  do j=1,jmax
    x(i,j) = i*j
  enddo

```

```

        enddo
do i=1,imax
    print*,(x(i,j),j=1,jmax)
enddo
end program twod
[powers@remote105 ame.20214]$ ifort ch7d.f90
[powers@remote105 ame.20214]$ a.out
    Enter the two lengths of the array
3 5
           1           2           3           4           5
           2           4           6           8          10
           3           6           9          12          15
[powers@remote105 ame.20214]$

```

Note

- Here, we form the array of integers whose elements are given by $x_{ij} = ij$.
- The variable lengths of the two-dimensional array is indicated by `dimension(:, :)`.
- We show results for a 3×5 array.

7.4 Non-standard do loop increment syntax

We last note that a do loop may start at an integer not equal to 1 and it may increment by integers not equal to one. An example is given in `ch7c.f90`:

```

[powers@darrow2-p 2e]$ cat ch7c.f90
program do
implicit none
integer :: i,imin=-4,imax=10,inc=2
do i=imin,imax,inc
    print*, 'i = ', i
end do
end program do
[powers@darrow2-p 2e]$ ifort ch7c.f90
[powers@darrow2-p 2e]$ a.out
i =          -4
i =          -2
i =           0
i =           2
i =           4
i =           6

```

```
i =          8  
i =          10  
[powers@darrow2-p 2e]$
```


Chapter 8

Whole array features

Read CES, Chapter 8.

The **Fortran** language has some limited features in manipulating vectors and arrays with simple global commands, known here as *whole array features*. These features are fairly characterized as clumsy but somewhat useful. Some of the whole array features are inspired by traditional linear algebra. In brief, there are **Fortran** short cuts for the

- vector dot product, `dot_product`
- matrix transpose, `transpose`
- matrix-vector product, `matmul` and
- matrix-matrix product, `matmul`.

Note, there is no shorthand for the matrix inverse, or for eigenvalues or eigenvectors, among other things. These important operations are often available as part of libraries external to the **Fortran** language, but easily woven into a program.

The reader will soon learn that, relative to some competing languages, especially **MATLAB**, **Fortran** has only limited effectiveness employing compact matrix notation for linear algebra problems. Worse still, the **Fortran** implementation does not always follow common linear algebra conventions. Nevertheless, those features that exist can be useful in writing compact code. Moreover, whatever method is used for matrix operations, **Fortran** will provide a very fast execution time, which is in fact what often matters most to the practicing engineer.

8.1 Dot product

8.1.1 Real vectors

We often wish to take the dot product of two vectors. Mathematically, we might say for vectors of length 3

$$\mathbf{u}^T \cdot \mathbf{v} = \begin{pmatrix} u_1 & u_2 & u_3 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = u_1 v_1 + u_2 v_2 + u_3 v_3 = \sum_{i=1}^3 u_i v_i. \quad (8.1)$$

In **Fortran**, this is achieved by the function `dot_product` via the command

```
dot_product(u,v)
```

As long as $\mathbf{u}$ and $\mathbf{v}$ are real, the dot product operation commutes:

$$\mathbf{u}^T \cdot \mathbf{v} = \mathbf{v}^T \cdot \mathbf{u}, \quad \mathbf{u}, \mathbf{v} \in \mathbb{R}. \quad (8.2)$$

An example is shown next of the dot product of two vectors $\mathbf{u} = (1, 2, 3)^T$ and $\mathbf{v} = (4, 5, 6)^T$. It is easily seen that

$$\mathbf{u}^T \cdot \mathbf{v} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = (1)(4) + (2)(5) + (3)(6) = 32. \quad (8.3)$$

In **Fortran**, this is embodied in the program `ch8a.f90`:

```
[powers@remote101 ame.20214]$ cat ch8a.f90
program dotproduct
implicit none
real :: u(3),v(3)
u(1) = 1.
u(2) = 2.
u(3) = 3.
v(1) = 4.
v(2) = 5.
v(3) = 6.
print*, 'u = ', u
print*, 'v = ', v
print*, 'u.v = ', dot_product(u,v)
print*, 'v.u = ', dot_product(v,u)
end program dotproduct
[powers@remote101 ame.20214]$ ifort ch8a.f90
[powers@remote101 ame.20214]$ a.out
```

```

u =      1.000000      2.000000      3.000000
v =      4.000000      5.000000      6.000000
u.v =     32.00000
v.u =     32.00000
[powers@remote101 ame.20214]$

```

We note

- We could have achieved the same result with a `do` loop and extra coding.
- The dot product commutes, as expected for real variables.

8.1.2 Complex vectors

If $\mathbf{u}$ and $\mathbf{v}$ are complex, it can be shown that

$$\overline{\mathbf{u}}^T \cdot \mathbf{v} = \overline{\mathbf{v}^T \cdot \mathbf{u}}, \quad \mathbf{u}, \mathbf{v} \in \mathbb{C}, \quad (8.4)$$

where the overline indicates the complex conjugate, $\overline{x + iy} = x - iy$. This unusual definition is to guarantee that $\overline{\mathbf{u}}^T \cdot \mathbf{u} = |\mathbf{u}|^2$ is a real positive scalar. Among other things, such a property is necessary for any viable theory of quantum physics.

Consider if

$$\mathbf{u} = \begin{pmatrix} 1 + i \\ 1 + 0i \\ 0 + i \end{pmatrix}. \quad (8.5)$$

Then, we see the square of the magnitude of $\mathbf{u}$ is

$$\overline{\mathbf{u}}^T \cdot \mathbf{u} = |\mathbf{u}|^2 = \sum_{j=1}^3 \overline{u_j} u_j = \begin{pmatrix} 1 - i & 1 - 0i & 0 - i \end{pmatrix} \begin{pmatrix} 1 + i \\ 1 + 0i \\ 0 + i \end{pmatrix} = 2 + 1 + 1 = 4. \quad (8.6)$$

In Fortran, this is embodied in the program `ch8b.f90`:

```

[powers@remote201 ame.20214]$ cat ch8b.f90
program dotproduct                                !calculate the square of the magnitude of
                                                !a complex vector via the dot product

implicit none
complex :: u(3)                                  !declare the complex vector
u(1) = (1,1)                                     !u(1) = 1+1i
u(2) = (1,0)                                     !u(2) = 1+0i
u(3) = (0,1)                                     !u(3) = 0+1i
print*, 'u = ', u
print*, 'u.u = ', dot_product(u,u)              !find the dot product

```

```

end program dotproduct
[powers@remote201 ame.20214]$ ifort ch8b.f90
[powers@remote201 ame.20214]$ a.out
u = (1.000000,1.000000) (1.000000,0.000000E+00) (0.000000E+00,1.000000)
u.u = (4.000000,0.000000E+00)
[powers@remote201 ame.20214]$

```

We note

- The complex numbers are expressed as ordered pairs, e.g. $u(1) = (1,1)$.
- The complex numbers were entered as integers, but were converted automatically to floating point numbers. We probably should have used decimal points to be careful, but it turned out not to matter in this case.
- The dot product of a complex scalar into itself indeed gives a positive real number.
- In this degenerate case, the dot product in fact commutes, only because we are dotting a vector into itself.
- The reader can easily verify that the dot product does not commute for two different complex vectors.

8.2 Matrix transpose

The matrix transpose simply trades elements across the diagonal. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \quad (8.7)$$

Its transpose is

$$\mathbf{A}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}. \quad (8.8)$$

In index notation, we would say

$$A_{ij}^T = A_{ji}. \quad (8.9)$$

We implement this in the code `ch8g.f90`:

```

[powers@darrow1-p ame.20214]$ cat ch8g.f90
program tpose      !This program finds the transpose of a matrix.
implicit none
integer :: A(2,3),At(3,2),i,j

```

```

A(1,1) = 1           !Initialize A
A(1,2) = 2
A(1,3) = 3
A(2,1) = 4
A(2,2) = 5
A(2,3) = 6

print*, 'raw form, A = ', A
print*, ' '
print*, 'structured form, A = '
do i=1,2
    print*, (A(i,j), j=1,3)
enddo

At=transpose(A)
print*, ' '
print*, 'raw form, At = ', At
print*, ' '
print*, 'structured form, At = '
do i=1,3
    print*, (At(i,j), j=1,2)
enddo
end program tpose
[powers@darrow1-p ame.20214]$ ifort ch8g.f90
[powers@darrow1-p ame.20214]$ a.out
raw form, A =           1           4           2           5           3
                   6

structured form, A =
                   1           2           3
                   4           5           6

raw form, At =           1           2           3           4           5
                   6

structured form, At =
                   1           4
                   2           5
                   3           6
[powers@darrow1-p ame.20214]$

```

Note:

- For variety, we chose integer arrays.
- $\mathbf{A}$ has dimensions 2×3 , and $\mathbf{A}^T$ has dimensions 3×2 .
- The command `transpose` works just as it is defined in linear algebra.
- The default `print` command prints each element of the matrix as a simple one-dimensional list of numbers, ordered column by column.
- It probably would have been better for **Fortran** to have been designed so the default `print` command was row by row, which would have allowed more consistency with standard matrix algebra.
- With a special structuring, we were able to print the matrices in a more standard form. This involved commands like `print*, (A(i,j), j=1,3)`, as well as the `do` loop in which it was embedded.
- We declared $\mathbf{A}$ and $\mathbf{A}^T$ to be integers, entered them as integers, and they print as integers.

8.3 Matrix-vector multiplication

We can form the product of a matrix and a vector, as we have seen in Ch. 7. Such an operation does not commute. When $\mathbf{T}$ is a matrix and $\mathbf{n}$ is vector,

$$\mathbf{T}^T \cdot \mathbf{n} \neq \mathbf{n}^T \cdot \mathbf{T}, \quad (8.10)$$

unless we have the special case in which $\mathbf{T}$ is symmetric. Much like the intrinsic function `dot_product` simplified the coding, the **Fortran** intrinsic function `matmul` simplifies matrix-vector multiplication. This is illustrated with the code `ch8c.f90`, where we compute

$$\mathbf{f} = \mathbf{T}^T \cdot \mathbf{n}, \quad f_i = \sum_{j=1}^3 T_{ij}^T n_j, \quad (8.11)$$

$$\mathbf{f}^T = \mathbf{n}^T \cdot \mathbf{T}, \quad f_i = \sum_{j=1}^3 n_j T_{ji}. \quad (8.12)$$

```
[powers@remote101 ame.20214]$ cat ch8c.f90
program stress
! program to calculate the surface
! traction force, f, given the stress
! tensor, T, and a user-input
! direction vector, v

implicit none
integer :: i,j
```

```

real, dimension(1:3,1:3) :: T,Tt      ! declaration for 2d array (matrix)
real, dimension(1:3) :: v,n,f         ! declaration for 1d arrays (vectors)
real :: magv
T(1,1) = 1.                          ! set values of stress tensor T
T(1,2) = 5.
T(1,3) = 3.
T(2,1) = -2.
T(2,2) = 4.
T(2,3) = 0.
T(3,1) = -1.
T(3,2) = 2.
T(3,3) = 3.
Tt = transpose(T)                    ! define the transpose of T
print*, 'enter v(1), v(2), v(3)'     ! enter values for the directions
read*,v(1),v(2),v(3)
magv = sqrt(v(1)**2 + v(2)**2 + v(3)**2) ! calculate the magnitude of v
do i=1,3                              ! calculate the normal vector n
    n(i) = v(i)/magv
end do
print*, 'n= ',n                      ! print the normal vector, n
do i=1,3                              ! perform the matrix multiplication
    f(i)=0.                          ! initialize f(i) to zero
    do j=1,3
        f(i) = f(i) + n(j)*T(j,i)    ! compute the sum
    end do
end do
print*, 'f = ',f                    ! print the surface traction, f
print*, 'f = transpose(n).T = ', matmul(n,T)
print*, 'f = transpose(T).n = ', matmul(Tt,n)
print*, 'T.n = ', matmul(T,n)
print*, 'n.n = ', dot_product(n,n)
end program stress
[powers@remote101 ame.20214]$ ifort ch8c.f90
[powers@remote101 ame.20214]$ a.out
enter v(1), v(2), v(3)
1 2 3
n=  0.2672612      0.5345225      0.8017837
f =  -1.603567      5.077964      3.207135
f = transpose(n).T =  -1.603567      5.077964      3.207135
f = transpose(T).n =  -1.603567      5.077964      3.207135
T.n =   5.345225      1.603567      3.207135
n.n =   0.9999999

```

[powers@remote101 ame.20214]\$

We note

- We used the useful intrinsic operation `transpose` to form the transpose of several matrices. Note that both $\mathbf{T}$ and $\mathbf{T}^T$ have dimension 3×3 .
- The function `matmul(n,T)` yielded the same result as our more complex program for $\mathbf{f}$. This really was the equivalent of forming $\mathbf{f}^T = \mathbf{n}^T \cdot \mathbf{T}$.
- We performed the commutation `matmul(T,n)` and found it yielded a different result, confirming that the matrix vector multiplication may not commute.
- We transposed $\mathbf{T}$ and formed `matmul(Ttranspose,n)` and recovered the correct $\mathbf{f}$.
- We checked using `dot_product(n,n)` that $\mathbf{n}^T \cdot \mathbf{n} = 1$, as expected.
- While doable and efficient to code, whole array matrix operations in **Fortran** are complicated relative to some other codes such as **MATLAB**.
- **Fortran** does not recognize the difference between row vectors and column vectors; each is simply a one-dimensional list. For this problem $\mathbf{n}$ and $\mathbf{n}^T$ as well as $\mathbf{f}$ and $\mathbf{f}^T$ are treated identically. This is not the case in **MATLAB**.

8.4 Matrix-matrix multiplication

We can also use the **Fortran** intrinsic `matmul` to multiply two matrices. They may be square or non-square. Let us consider $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, where

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 2 \\ 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 \end{pmatrix} \quad (8.13)$$

Note if we write in the dimensions of the arrays as subscripts, we see what the matrix product will yield,

$$\mathbf{A}_{2 \times 3} \cdot \mathbf{B}_{3 \times 4} = \mathbf{C}_{2 \times 4}. \quad (8.14)$$

The adjacent 3's cancel, leaving us with a 2×4 result. Standard matrix multiplication then reveals that

$$\underbrace{\begin{pmatrix} 1 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} 1 & 1 & 1 & 2 \\ 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 \end{pmatrix}}_{\mathbf{B}} = \underbrace{\begin{pmatrix} 7 & 13 & 17 & 22 \\ 4 & 7 & 9 & 12 \end{pmatrix}}_{\mathbf{C}} \quad (8.15)$$

In index notation, we could say

$$C_{ij} = \sum_{k=1}^3 A_{ik} B_{kj}. \quad (8.16)$$

This is embodied in the Fortran program `ch8d.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch8d.f90
program multiply          !This program multiplies two rectangular matrices.
implicit none
integer :: A(2,3),B(3,4),C(2,4),i,j
A(1,1) = 1
A(1,2) = 2
A(1,3) = 2
A(2,1) = 1
A(2,2) = 1
A(2,3) = 1
B(1,1) = 1
B(1,2) = 1
B(1,3) = 1
B(1,4) = 2
B(2,1) = 2
B(2,2) = 3
B(2,3) = 4
B(2,4) = 5
B(3,1) = 1
B(3,2) = 3
B(3,3) = 4
B(3,4) = 5
C=matmul(A,B)
print*, 'C = ', C
print*, ' '
print*, 'C = '
do i=1,2
  print*, (C(i,j), j=1,4)
enddo
end program multiply
[powers@darrow1-p ame.20214]$ ifort ch8d.f90
ch8d.f90(22): (col. 3) remark: LOOP WAS VECTORIZED.
[powers@darrow1-p ame.20214]$ a.out
C =          7          4          13          7          17          9
      22          12
```

```

C =
      7      13      17      22
      4       7       9      12
[powers@darrow1-p ame.20214]$

```

Note:

- We used the simplest possible declaration statement.
- We had to initialize each of the elements of A and B.
- The compiler noted the loop was vectorized. At present, this is not important.
- The output C was formed via `matmul`. The numbers have the correct value, but one must take care to interpret them as to how they actually fit into the array C. The unformatted output of C is given as a column-by-column list.
- We also printed C in a traditionally structured way via a `do` loop.

We illustrate an alternate and more compact way to initialize matrices and do the same task as the previous program in the code of `ch8e.f90`:

```

[powers@darrow1-p ame.20214]$ cat ch8e.f90
program multiply                                !This program multiplies two rectangular matrices.
implicit none
integer :: A(2,3) = &                          !Note this ordering convention is backwards from
      (/1, 1, &                                !that of standard matrix algebra...
      2, 1, &
      2, 1 /)
integer :: B(3,4) = &                          !Once again a backwards ordering convention
      (/1, 2, 1, &
      1, 3, 3, &
      1, 4, 4, &
      2, 5, 5/)
integer :: C(2,4)
C=matmul(A,B)
print*,C
end program multiply
[powers@darrow1-p ame.20214]$ ifort ch8e.f90
ch8e.f90(13): (col. 3) remark: LOOP WAS VECTORIZED.
[powers@darrow1-p ame.20214]$ a.out
      7       4      13       7      17       9
      22      12
[powers@darrow1-p ame.20214]$

```

We note that

- We achieved precisely the same result as the previous version of the algorithm.
- The ordering convention is precisely backwards from conventional mathematics! We certainly have **A** as a 2×3 matrix. But it is entered in as data in a way which appears to be appropriate for a 3×2 matrix. This is unfortunate, and due to the adoption of an alternate approach to rows and columns in **Fortran**. Thus one needs to be careful when performing matrix operations.
- **Fortran** really only keeps track of one-dimensional lists, and we entered the data for **A** and **B** as one-dimensional lists, *but we used the continuation line character &* to aid in the visual display of the matrix structure on input.
- This technique for initializing two-dimensional arrays also works for one-dimensional arrays.

We can also achieve the same result of a traditional matrix-matrix multiplication from a detailed triply nested **do** loop, found in `ch8f.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch8f.f90
program multiply          !This program multiplies two rectangular matrices.
implicit none
integer :: A(2,3),B(3,4),C(2,4),i,j,k
A(1,1) = 1
A(1,2) = 2
A(1,3) = 2
A(2,1) = 1
A(2,2) = 1
A(2,3) = 1
B(1,1) = 1
B(1,2) = 1
B(1,3) = 1
B(1,4) = 2
B(2,1) = 2
B(2,2) = 3
B(2,3) = 4
B(2,4) = 5
B(3,1) = 1
B(3,2) = 3
B(3,3) = 4
B(3,4) = 5
do i=1,2
  do j=1,4
    C(i,j) = 0
    do k=1,3
```

```

        C(i,j) = C(i,j)+A(i,k)*B(k,j)
    enddo
enddo
enddo
print*,C
end program multiply
[powers@darrow1-p ame.20214]$ ifort ch8f.f90
[powers@darrow1-p ame.20214]$ a.out
         7         4        13         7        17         9
        22        12
[powers@darrow1-p ame.20214]$

```

Note that

- Triply nested do loops pose no particular problem.
- We had to insure that *i* ran from 1 to 2 and *j* from 1 to 4, consistent with the dimensions of *C*. Then *k* ran from 1 to 3, consistent with the dimensions of *A* and *B*.

8.5 Addition

Two arrays of the same dimension can be added. As an example, we take

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}. \quad (8.17)$$

Then

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}. \quad (8.18)$$

In index notation this simply says

$$C_{ij} = A_{ij} + B_{ij}. \quad (8.19)$$

This works in Fortran in an obvious way, as seen in `ch8h.f90`:

```

[powers@darrow1-p ame.20214]$ cat ch8h.f90
program add      !This program adds two matrices.
implicit none
integer :: A(2,2)=(/1,3,2,4/),B(2,2)=(/5,7,6,8/),C(2,2),i,j

print*, ' A = '
do i=1,2
    print*,(A(i,j), j=1,2)

```

```

        enddo

print*, ' B = '
do i=1,2
    print*,(B(i,j), j=1,2)
enddo

C = A + B

print*, ' C = A+B = '
do i=1,2
    print*,(C(i,j), j=1,2)
enddo

end program add
[powers@darrow1-p ame.20214]$ ifort ch8h.f90
ch8h.f90(15): (col. 1) remark: LOOP WAS VECTORIZED.
[powers@darrow1-p ame.20214]$ a.out
A =
      1      2
      3      4
B =
      5      6
      7      8
C = A+B =
      6      8
     10     12
[powers@darrow1-p ame.20214]$

```

Note:

- We used the compact, but non-intuitive initialization of **A** and **B** as a one-dimensional list, column by column, starting with column 1, of each matrix.
- We printed each matrix using the nested `do` statements which allow the natural matrix form and structure to be displayed.
- The matrix **C** is indeed composed of the sum of each element of **A** and **B**. The compact statement `C = A+ B` was used instead of a set of instructions involving two `do` loops to sum the matrices.

8.6 Element-by-element matrix “multiplication”

The command `*` in `Fortran` when applied to matrices does not imply ordinary matrix multiplication. It simply invokes a term-by-term multiplication, which is occasionally of value.

Consider

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}. \quad (8.20)$$

Note that the formal matrix multiplication does not exist because of incompatible dimensions of $\mathbf{A}$ and $\mathbf{B}$ for this operation. However, the element-by-element “multiplication” does exist and is given by

$$\mathbf{C} = \begin{pmatrix} (1)(7) & (2)(8) & (3)(9) \\ (4)(10) & (5)(11) & (6)(12) \end{pmatrix} = \begin{pmatrix} 7 & 16 & 27 \\ 40 & 55 & 72 \end{pmatrix}. \quad (8.21)$$

This is embodied in the program `ch8i.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch8i.f90
program multiply          !This program "multiplies" two matrices.
                        !It is not ordinary matrix multiplication
implicit none
integer :: A(2,3)=(/1,4,2,5,3,6/),B(2,3)=(/7,10,8,11,9,12/),C(2,3),i,j

print*, ' A = '
do i=1,2
  print*,(A(i,j), j=1,3)
enddo

print*, ' B = '
do i=1,2
  print*,(B(i,j), j=1,3)
enddo

C = A*B

print*, ' C = A*B = '
do i=1,2
  print*,(C(i,j), j=1,3)
enddo

end program multiply
[powers@darrow1-p ame.20214]$ ifort ch8i.f90
[powers@darrow1-p ame.20214]$ a.out
```

```
A =  
      1      2      3  
      4      5      6  
B =  
      7      8      9  
     10     11     12  
C = A*B =  
      7      16     27  
     40     55     72  
[powers@darrow1-p ame.20214]$
```

Similarly the Fortran statement `sqrt(A)` takes the square root of every element of the matrix A. This is not a traditional matrix function.

Chapter 9

Output of results

Read CES, Chapter 9.

The `print*` statement we have used thus far sends raw output to the screen, and is a good way. But it can be aesthetically unwieldy to the extent that the reader has unnecessary difficulty interpreting the results. To alleviate this difficulty, one can use a variety of tools to produced output with better format. And the key to this is the use of what is known as a `format` statement.

9.1 Unformatted printing

Let us first consider some clumsy unformatted output, generated by the program `ch9a.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch9a.f90
program unformatoutput
implicit none
integer :: i
real :: x
do i = 1,8
    x = 10.**i
    print*,i,x
end do
end program unformatoutput
[powers@darrow1-p ame.20214]$ ifort ch9a.f90
[powers@darrow1-p ame.20214]$ a.out
      1    10.00000
      2   100.0000
      3  1000.000
      4 10000.00
      5 100000.0
      6 1000000.
```

```

      7  1.0000000E+07
      8  1.0000000E+08
[powers@darrow1-p ame.20214]$

```

Note:

- The decimal points are not aligned for the real numbers.
- The output changes suddenly to exponential notation from ordinary decimal notation.

9.2 Formatted printing

We gain more control over the output with use of the `format` command, as shown in `ch9b.f90`:

```

[powers@darrow1-p ame.20214]$ cat ch9b.f90
program formatoutput
implicit none
integer :: i
real :: x
do i = 1,8
  x = 10.**i
  print 100,i,x
  100 format(i3,3x,f12.2)
end do
end program formatoutput
[powers@darrow1-p ame.20214]$ ifort ch9b.f90
[powers@darrow1-p ame.20214]$ a.out
 1          10.00
 2          100.00
 3         1000.00
 4        10000.00
 5       100000.00
 6      1000000.00
 7     10000000.00
 8    100000000.00
[powers@darrow1-p ame.20214]$

```

Note:

- The decimal places are now aligned.
- There are only two digits to the right of the decimal point in all real numbers.
- There is no change to exponential notation.

- This is realized due to the `format` statement `100 format(i3,3x,f12.2)`. Let us break this down.
 - `100`: This is a label, and is called upon by the formatted `print 100`. We often place the `format` statement near the `print` statement, but this is not required. It is also possible to have many `print` statements draw upon one `format` statement.
 - `i3`: This says to print the number as a three digit integer. If the integer has in fact four or more entries, the output will be problematic, but the program will execute.
 - `3x`: This says to skip three spaces.
 - `f12.2`: The `f` format says to print a real number in non-exponential notation. The `12` assigns twelve spaces to the number. These spaces include the `+/-` signs, the decimal point, and all digits. The `.2` says to include exactly two digits to the right of the decimal point.

Let us see what happens when we only allow ten instead of twelve slots for the real number, and simultaneously ask for three digits right of the decimal point. To do that, we change `f12.2` to `f10.3`. Let's also change `x` to `-x`. Let's also change the `i3` format to `i1`, so that we only print one digit of the integer. We will also let 50 values be computed. Results are shown in `ch9c.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch9c.f90
program formatoutput2
implicit none
integer :: i
real :: x
do i = 1,50
  x = -10.**i
  print 100,i,x
  100 format(i1,3x,f10.3)
end do
end program formatoutput2
[powers@darrow1-p ame.20214]$ ifort ch9c.f90
[powers@darrow1-p ame.20214]$ a.out
1      -10.000
2     -100.000
3    -1000.000
4   -10000.000
5    *****
6    *****
7    *****
8    *****
```

CC BY-NC-ND. 29 April 2021, J. M. Powers.

```
[powers@darrow1-p ame.20214]$
```

Note:

- The last value of x that is actually displayed is -10000.000 . This number has entries in 10 slots:
 - the minus sign: 1
 - digits to the left of the decimal point: 5
 - the decimal point: 1
 - digits to the right of the decimal point: 3
 - Total slots= 10
- This is consistent with the `f10.3` format.
- Numbers that cannot fit into this format are displayed as `*****`. Note there are ten stars.
- The last integer that displays is the last single digit integer, 9.
- At a certain value, x is not just unprintable, it has encountered an overflow error, indicated by `-Infinity` in the output.

We can change the format to print many large numbers. In `ch9h.f90`, we change to an `f50.3` format and get

```
[powers@darrow1-p ame.20214]$ cat ch9h.f90
program formatoutput2
implicit none
integer :: i
real :: x
do i = 1,50
  x = -10.**i
  print 100,i,x
  100 format(i3,3x,f50.3)
end do
end program formatoutput2
[powers@darrow1-p ame.20214]$ ifort ch9h.f90
[powers@darrow1-p ame.20214]$ a.out
1                -10.000
2                -100.000
3                -1000.000
4                -10000.000
5                -100000.000
```

6	-1000000.000
7	-10000000.000
8	-100000000.000
9	-1000000000.000
10	-10000000000.000
11	-9999997952.000
12	-999999995904.000
13	-9999999827968.000
14	-100000000376832.000
15	-999999986991104.000
16	-10000000272564224.000
17	-99999998430674944.000
18	-999999984306749440.000
19	-9999999980506447872.000
20	-100000002004087734272.000
21	-1000000020040877342720.000
22	-9999999778196308361216.000
23	-99999997781963083612160.000
24	-1000000013848427855085568.000
25	-9999999562023526247432192.000
26	-100000002537764290115403776.000
27	-999999988484154753734934528.000
28	-9999999442119689768320106496.000
29	-100000001504746621987668885504.000
30	-1000000015047466219876688855040.000
31	-9999999848243207295109594873856.000
32	-100000003318135351409612647563264.000
33	-999999994495727286427992885035008.000
34	-9999999790214767953607394487959552.000
35	-100000004091847875962975319375216640.000
36	-999999961690316245365415600208216064.000
37	-9999999933815812510711506376257961984.000
38	-99999996802856924650656260769173209090.000
39	-Infinity
40	-Infinity
41	-Infinity
42	-Infinity
43	-Infinity
44	-Infinity
45	-Infinity
46	-Infinity
47	-Infinity

```

48                                     -Infinity
49                                     -Infinity
50                                     -Infinity
[powers@darrow1-p ame.20214]$

```

Note that we only achieve seven digits of accuracy and that for large numbers, the output values have a strong pseudo-random component in many digits. This is because we only requested single precision accuracy.

Let us do the same chore in exponential notation, using the so-called **e** format. So, let us replace **f10.3** by **e10.3** and examine the result in **ch9d.f90**:

```

[powers@darrow1-p ame.20214]$ cat ch9d.f90
program formatoutput3
implicit none
integer :: i
real :: x
do i = 1,8
    x = -10.**i
    print 100,i,x
    100 format(i3,3x,e10.3)
end do
end program formatoutput3
[powers@darrow1-p ame.20214]$ ifort ch9d.f90
[powers@darrow1-p ame.20214]$ a.out
 1  -0.100E+02
 2  -0.100E+03
 3  -0.100E+04
 4  -0.100E+05
 5  -0.100E+06
 6  -0.100E+07
 7  -0.100E+08
 8  -0.100E+09
[powers@darrow1-p ame.20214]$

```

Note:

- There is no longer a problem with data that cannot be displayed. This is an advantage of the **e** format.
- The alignment is somewhat better.

Character variables can be printed to the screen in a formatted form using the **a** format. An example is given in **ch9e.f90**:

```
[powers@darrow1-p ame.20214]$ cat ch9e.f90
program formatoutput4
implicit none
character (12) :: a,b,c,d,e
a = 'I'
b = 'am'
c = 'an'
d = 'engineering'
e = 'enthusiast.'
print*,a,b,c,d,e
print 100, a,b,c,d,e
100 format(a1,1x,a2,1x,a2,1x,a11,1x,a11)
end program formatoutput4
[powers@darrow1-p ame.20214]$ ifort ch9e.f90
[powers@darrow1-p ame.20214]$ a.out
I          am          an          engineering enthusiast.
I am an engineering enthusiast.
[powers@darrow1-p ame.20214]$
```

Note:

- The variables `a, b, c, d, e` are declared character variables which are allowed up to 12 letters in length. They are assigned values as ASCII strings, and their assignment must be enclosed within single quotation marks.
- The unformatted `print *` statement gives an unevenly formatted result.
- The formatted `print 100` statement calls upon statement 100 to dictate its output format to the screen.

9.3 Formatted writing to files

Often we will want to write our results to a file. There are many variants on this discussed in the text. Here we give a simple example. To write to a file, we first must **open** that file, where its name is declared. Then we are allowed to write to it. To write in a formatted fashion is nearly identical to printing in a formatted fashion. Here we modify an earlier program to implement the writing of formatted output to the file `numbers`; see `ch9f.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch9f.f90
program formatoutput5
implicit none
integer :: i
real :: x
```

```

open(unit=20,file='numbers')
do i = 1,8
    x = -10.**i
    print 100,i,x
    write(20,100) i,x
    100 format(i3,3x,e10.3)
end do
end program formatoutput5
[powers@darrow1-p ame.20214]$ ifort ch9f.f90
[powers@darrow1-p ame.20214]$ a.out
 1  -0.100E+02
 2  -0.100E+03
 3  -0.100E+04
 4  -0.100E+05
 5  -0.100E+06
 6  -0.100E+07
 7  -0.100E+08
 8  -0.100E+09
[powers@darrow1-p ame.20214]$ cat numbers
 1  -0.100E+02
 2  -0.100E+03
 3  -0.100E+04
 4  -0.100E+05
 5  -0.100E+06
 6  -0.100E+07
 7  -0.100E+08
 8  -0.100E+09
[powers@darrow1-p ame.20214]$

```

Note:

- We associated the arbitrary integer 20 with the output file we chose to name **numbers**.
- We did not put any identifier like **.txt** or **.dat** on the file **numbers**. This is fine, but some applications will not recognize it as a text file, while others will.
- We used the same format statement to **print** and to **write** the same data; consequently, the output to the screen is identical to that in the file.

If we neglect to open a file, but write to “20” anyway, most compilers will be accommodating. The **ifort** compiler will obligingly open a file and give it the name **fort.20**. It will then write to this file, if requested. This is shown in **ch9g.f90**:

```

[powers@darrow1-p ame.20214]$ cat ch9g.f90

```

```
program formatoutput6
implicit none
integer :: i
real :: x
do i = 1,8
    x = -10.**i
    write(20,100) i,x
    100 format(i3,3x,e10.3)
end do
end program formatoutput6
[powers@darrow1-p ame.20214]$ ifort ch9g.f90
[powers@darrow1-p ame.20214]$ a.out
[powers@darrow1-p ame.20214]$ cat fort.20
 1  -0.100E+02
 2  -0.100E+03
 3  -0.100E+04
 4  -0.100E+05
 5  -0.100E+06
 6  -0.100E+07
 7  -0.100E+08
 8  -0.100E+09
[powers@darrow1-p ame.20214]$
```

Chapter 10

Reading data

Read CES, Chapter 10.

The `read*` statement we have used thus far reads raw input from the screen, and is a good way to read limited amounts of data. But it can be problematic if there is a large amount of input data that could change from case to case. In such a case, we are well advised to read the data in from a file. And there are two ways to read the data, unformatted and formatted. Unformatted is the simplest, but is difficult for the user to comprehend. Formatting input data requires more effort, but is easier for the user to understand.

10.1 Unformatted reading from a file

Let us say we have available a list of numbers in a file named `ch10a.in`:

```
[powers@darrow1-p ame.20214]$ cat ch10a.in
0. 0.
1.1232 2.23
2. 4.1
3.51 7.9
5. 10.
[powers@darrow1-p ame.20214]$
```

There are two columns of data, but the file lacks a clean structure. Let us consider a **Fortran** program to read this data and print it to the screen in a formatted fashion. We will associate the first column with a vector `x(i)` and the second column with a vector `y(i)`. The program and its execution are found in `ch10a.f90`:

```
[powers@darrow1-p ame.20214]$ cat ch10a.f90
program readdata
implicit none
real :: x(5),y(5)
```

```

integer :: i
open(10,file='ch10a.in')
do i=1,5
    read(10,*) x(i),y(i)
    print 199, i,x(i),y(i)
end do
199 format(i4,2x,e12.6,2x,e12.6)
end program readdata
[powers@darrow1-p ame.20214]$ ifort ch10a.f90
[powers@darrow1-p ame.20214]$ a.out
  1  0.000000E+00  0.000000E+00
  2  0.112320E+01  0.223000E+01
  3  0.200000E+01  0.410000E+01
  4  0.351000E+01  0.790000E+01
  5  0.500000E+01  0.100000E+02
[powers@darrow1-p ame.20214]$

```

Note:

- The number 10 is associated with the input file `ch10a.in`.
- We read from unit 10, which is now our input file.
- We print the data to the screen in a formatted fashion and see that it is in fact the same as our input data.
- We needed to know that there were only five lines in the input file. This can pose a problem for input data files of arbitrary length.

Let us say we try to consider 10 values of `x` and `y`, but we only have 5 values in the input file. This induces a problem, as is evident in `ch10b.f90`:

```

[powers@remote202 ame.20214]$ cat ch10b.f90
program readdata
implicit none
real :: x(10),y(10)
integer :: i
open(10,file='ch10a.in')
do i=1,10
    read(10,*) x(i),y(i)
    print 199, i,x(i),y(i)
end do
199 format(i4,2x,e12.6,2x,e12.6)
end program readdata
[powers@remote202 ame.20214]$ ifort ch10b.f90

```

```
[powers@remote202 ame.20214]$ a.out
 1  0.000000E+00  0.000000E+00
 2  0.112320E+01  0.223000E+01
 3  0.200000E+01  0.410000E+01
 4  0.351000E+01  0.790000E+01
 5  0.500000E+01  0.100000E+02
forrtl: severe (24): end-of-file during read, unit 10,
      file /afs/nd.edu/users/powers/www/ame.20214/2e/ch10a.in
```

Image	PC	Routine	Line	Source
a.out	0000000000456C9E	Unknown	Unknown	Unknown
a.out	0000000000455E9A	Unknown	Unknown	Unknown
a.out	000000000043D936	Unknown	Unknown	Unknown
a.out	000000000042FAA2	Unknown	Unknown	Unknown
a.out	000000000042F0BE	Unknown	Unknown	Unknown
a.out	0000000000410160	Unknown	Unknown	Unknown
a.out	0000000000402DE2	Unknown	Unknown	Unknown
a.out	000000000040290E	Unknown	Unknown	Unknown
libc.so.6	00000032CAE1ECDD	Unknown	Unknown	Unknown
a.out	0000000000402829	Unknown	Unknown	Unknown

```
[powers@remote202 ame.20214]$
```

We changed the array dimensions to 10 and tried to read 10 values. Clearly 5 values were read and printed to the screen, But an execution error arose when the unavailable sixth value was attempted to be read.

One common, albeit clumsy, way around this issue is to guarantee that the input data file has a last line of data which is an unusual value. And when the main code reads this unusual value, it knows that it is at the end of the input data file. Here we have created `ch10b.in`:

```
[powers@remote202 ame.20214]$ cat ch10b.in
0. 0.
1.1232 2.23
2. 4.1
3.51 7.9
5. 10.
-9999. -9999.
[powers@remote202 ame.20214]$
```

The last line has two unusual values `-9999.`, twice. This number should be designed to be an unrealistic value for the user's application. Our modified code then has arrays dimensioned to a large value so that we can handle potentially large input files. And it has a condition to quit reading the data file. All this is seen in the listing and execution of `ch10c.f90`:

```
[powers@remote202 ame.20214]$ cat ch10c.f90
```

```

program readdata
implicit none
integer, parameter :: imax=1000
real :: x(imax),y(imax)
integer :: i
open(10,file='ch10b.in')
do
    read(10,*) x(i),y(i)
    if(x(i).lt.-9998.)exit
    print 199, i,x(i),y(i)
end do
199 format(i4,2x,e12.6,2x,e12.6)
end program readdata
[powers@remote202 ame.20214]$ ifort ch10c.f90
[powers@remote202 ame.20214]$ a.out
    0  0.000000E+00  0.000000E+00
    0  0.112320E+01  0.223000E+01
    0  0.200000E+01  0.410000E+01
    0  0.351000E+01  0.790000E+01
    0  0.500000E+01  0.100000E+02
[powers@remote202 ame.20214]$

```

Note:

- We are prepared for as many as `imax = 1000` data points, having dimensioned each vector to this value.
- Our `do` loop is now unbounded by a counter.
- We exit the `do` loop if we find a data point with a value of `x(i)` less than `-9998`.
- If our input data file does not have the appropriate end of file statement, we will likely find an execution error.

10.2 Formatted reading from a file

Often it is helpful for the user for input data in a file to be formatted in a way that the user can recognize the data. Let us rewrite the program of Sec. 7.1 so that all possible input data is read from file `ch10d.in`.

```

[powers@darrows2-p ame.20214]$ cat ch10d.in
a (Kelvin)      =      270.
b (Kelvin)      =       10.
c (hour)        =       24.

```

```
tmax (hour)      =      24.
number of samples =      10
123456789*123456789*123456789*
[powers@darrow2-p ame.20214]$
```

We see

- There is more room for descriptive words in the input file.
- The last line is a convenience for counting spaces. It will not be read and not influence the program.

Use of this input file will allow us to compile the program *once*, and if we want to change the input file, we can simply edit it, and re-run the binary executable `a.out`. The program itself, its compilation, and execution are given as

```
[[powers@darrow2-p ame.20214]$ cat ch10d.f90
program temperature
implicit none
integer (kind=4) :: i,imax
real (kind=4), dimension(:), allocatable :: t,temp
real (kind=4) :: a,b,c,tmax,pi,dt
open(12,file='ch10d.in')      !open the input file
read(12,100) a                 !read from the input file
read(12,100) b                 !read from the input file
read(12,100) c                 !read from the input file
read(12,100) tmax              !read from the input file
read(12,101) imax              !read from the input file
100 format(20x,f10.5)          !format of the input data for reals
101 format(20x,i10)            !format of the input data for integers
allocate(t(1:imax))
allocate(temp(1:imax))
pi = 4.*atan(1.)
print*, 'pi = ', pi
dt = tmax/(real(imax)-1.)
do i=1,imax
    t(i) = dt*(i-1)
    temp(i) = a - b*sin(2.*pi*t(i)/c)
    print*,i,t(i),temp(i)
end do
end program temperature
[powers@darrow2-p ame.20214]$ ifort ch10d.f90
[powers@darrow2-p ame.20214]$ a.out
pi =      3.141593
```

1	0.0000000E+00	270.0000
2	2.666667	263.5721
3	5.333333	260.1519
4	8.000000	261.3398
5	10.66667	266.5798
6	13.33333	273.4202
7	16.00000	278.6602
8	18.66667	279.8481
9	21.33333	276.4279
10	24.00000	270.0000

[powers@darrow2-p ame.20214]\$

Note:

- We read in values for `a`, `b`, `c`, `tmax`, and `imax` from file `ch10d.in`
- We chose to read one value for each read; we could have restructured the reading so that one read statement read several variables.
- The 100 format statement said to ignore the first 20 spaces in the line, and read as a real variable that which occupies the next 10 spaces, with up to five decimal points. This employed the `f10.5` format. For exponential notation data, we could have chosen `e10.5`, for example.
- The 101 format statement said to ignore the first 20 spaces in the line, and read as an integer variable that which occupies the next 10 spaces.

Chapter 11

I/O Concepts

Read CES, Chapter 11.

There is some additional information on Input/Output concepts in the text. The interested student should review the appropriate material.

Chapter 12

Functions

Read C&S, Chapter 12.

Functions are a useful aspect of the **Fortran** language. Many intrinsic functions are built into the language and simplify many mathematical operations. These are pre-defined functions. It is also possible to build user-defined functions. Let us examine both types.

12.1 Pre-defined functions

Fortran has a large number of pre-defined functions whose general structure is of the form

```
y = function(x)
```

Here **x** is the input, **y** is the output, and **function** is the intrinsic **Fortran** function. One example is the exponential function. The **Fortran** version of

$$y = e^x, \tag{12.1}$$

is

```
y = exp(x)
```

Here **exp** is the intrinsic function.

We list some of the more common **Fortran** functions in Table 12.1, similar to that given in C&S. The text of C&S has a more complete discussion.

As might be expected, these functions may have limitations on their domain. Consider the square root function applied to a real number and to a complex number in **ch12a.f90**:

```
[powers@darrows2-p ame.20214]$ cat ch12a.f90
program sqroot
real :: x = -1.
print*,sqrt(x)
```

Function	Action	Example
<code>int</code>	conversion to integer	<code>j = int(x)</code>
<code>real</code>	conversion to real	<code>x = real(j)</code>
<code>cmplx</code>	conversion to complex	<code>z = cmplx(x)</code>
<code>abs</code>	absolute value	<code>x = abs(x)</code>
<code>sqrt</code>	square root (positive)	<code>x = sqrt(y)</code>
<code>exp</code>	exponential	<code>y = exp(x)</code>
<code>log</code>	natural logarithm	<code>x = log(y)</code>
<code>log10</code>	common logarithm	<code>x = log10(y)</code>
<code>sin</code>	sine	<code>x = sin(y)</code>
<code>cos</code>	cosine	<code>x = cos(y)</code>
<code>tan</code>	tangent	<code>x = tan(y)</code>
<code>asin</code>	arcsine	<code>y = asin(x)</code>
<code>acos</code>	arccosine	<code>y = acos(x)</code>
<code>atan</code>	arctangent	<code>y = atan(x)</code>
<code>sinh</code>	hyperbolic sine	<code>x = sinh(y)</code>
<code>cosh</code>	hyperbolic cosine	<code>x = cosh(y)</code>
<code>tanh</code>	hyperbolic tangent	<code>x = tanh(y)</code>
<code>asinh</code>	inverse hyperbolic sine	<code>y = asinh(x)</code>
<code>acosh</code>	inverse hyperbolic cosine	<code>y = acosh(x)</code>
<code>atanh</code>	inverse hyperbolic tangent	<code>y = atanh(x)</code>

Table 12.1: List of some common **Fortran** pre-defined functions.

```

end program sqroot
[powers@darrows2-p ame.20214]$ ifort ch12a.f90
[powers@darrows2-p ame.20214]$ a.out
NaN
[powers@darrows2-p ame.20214]$

```

Note:

- We set `x` to be real and assigned it a negative value.
- We then tried to take the square root of a negative real number. Because the square root of a negative real number is not a real number, the program refuses to perform the calculation and returns the value `NaN`, which stands for *Not a Number*.

Let us declare `x` to be complex and take its square root; see `ch12b.f90`:

```

[powers@darrows2-p ame.20214]$ cat ch12b.f90

```

```

program sqroot
complex :: x = (-1.,0.)
print*,sqrt(x)
end program sqroot
[powers@darrow2-p ame.20214]$ ifort ch12b.f90
[powers@darrow2-p ame.20214]$ a.out
(0.0000000E+00,1.0000000)
[powers@darrow2-p ame.20214]$

```

- A correct value of $0 + i$ is returned.
- The actual value of $\sqrt{-1}$ is $0 \pm i$; we must recognize that only one of the roots is returned by `sqrt`.

In a similar manner, functions such as $\sin^{-1}$, $\cos^{-1}$, or $\tan^{-1}$ are multi-valued, but **Fortran** will only return one value. For example, while we know that

$$\sin^{-1}(1) = \frac{\pi}{2} + 2n\pi, \quad n = \dots, -2, -1, 0, 1, 2, \dots \quad (12.2)$$

Fortran returns only

```
asin(1.) = 1.570796
```

which we see is a single precision representation of $\pi/2$. In fact, **Fortran** defines `asin` so that its range is

$$\sin^{-1}(x) = \text{asin}(x) \in [-\pi/2, \pi/2]. \quad (12.3)$$

Note also that the domain of `asin` must be such that

$$x \in [-1, 1]. \quad (12.4)$$

The user must take care that the appropriate value of the multi-valued inverse functions is being used.

12.2 User-defined functions

We may use a type of subroutine, a *user-defined function*, to simplify repeated calculations. Such user-defined functions are a part of a so-called *object-oriented program*. Object-oriented programming is often closely associated with languages such as **C++** or **Java**, which employ the so-called *class* to serve a similar role as a subroutine. In **Fortran** the `module` is roughly equivalent to the class of **C++** or **Java** and brings much of the same functionality. Certainly it is possible to write **Fortran** user-defined functions without the use of `module`, but the modern user is advised to employ this feature for stronger compatibility across different

computing platforms, it is possible and useful to have **Fortran**-generated code interact with **C++** or **Java** code.

Let us say we have a generic function $f(x)$ and wish to have a program to build a list of values of x and $f(x)$ for $x \in [0, 1]$. We want the main program to be general, and to have the ability to plug in a function from somewhere else. We will take

$$y = f(x) = \sin(2\pi x), \quad x \in [0, 1]. \quad (12.5)$$

We can explore this in the following set of example programs, each of which employs a different program structure to achieve the same end.

12.2.1 Simple structure

The following gives a simple structure for evaluating a user-defined function, based on a module. It is found in `ch12c.f90`:

```
[powers@remote102 ame.20214]$ cat ch12c.f90
!-----
module myfunction                                !This is a module declaration
contains
real function f(x)                               !This is the function subroutine
implicit none
real, intent(in) :: x                           !Declare x as a real input variable
real :: pi
pi = 4.*atan(1.)
f = sin(2.*pi*x)
end function f
end module myfunction
!
!-----
!
program fofx                                     !This is the main program--it comes last
use myfunction                                  !The main program draws upon myfunction
implicit none
real :: x,y,xmin=0.,xmax=1.,dx
integer :: i,n
print*,'enter number of points, n>1'
read*,n
dx = (xmax-xmin)/real(n-1.)
do i=1,n
    x = xmin + dx*(i-1.)
    y = f(x)                                     !Here we call on the function subroutine
    print 100,x,y
```

```

    100 format(f12.6,2x,f12.6)
    end do
end program fofx
!-----
[powers@remote102 ame.20214]$ ifort ch12c.f90
[powers@remote102 ame.20214]$ a.out
  enter number of points, n>1
13
    0.000000      0.000000
    0.083333      0.500000
    0.166667      0.866025
    0.250000      1.000000
    0.333333      0.866025
    0.416667      0.500000
    0.500000      0.000000
    0.583333     -0.500000
    0.666667     -0.866025
    0.750000     -1.000000
    0.833333     -0.866025
    0.916667     -0.500000
    1.000000      0.000000
[powers@remote102 ame.20214]$

```

Note:

- The function subroutine `f(x)` is embedded within the module named `myfunction`, which appears *first* in the program.
- The module syntax requires that we identify that which it contains.
- For function subroutines, we exercise the option of reserving some variables for input. Here our input variable is `x`, and it is reserved as input by the statement `real, intent(in) :: x`.
- The function must have a type, here `real`.
- The action of the function is to take a value of `x` and map it into `f` and return that value to the main program.
- The main program appears *last*. It calls upon the function subroutine with the statement `f(x)`.
- We must inform the main program that a function subroutine module here `myfunction`, is to be employed. That is achieved by `use myfunction`, which appears *before* variable declaration statements.

- The user must have defined $f(\mathbf{x})$, since it is not a pre-defined Fortran function.
- We can extend this syntax to evaluate functions of N variables of the form $f(x_1, x_2, x_3, \dots, x_N)$. That is to say, functions can take several inputs and map them to a single output.

12.2.2 Use of dummy variables

The previous example used $\mathbf{x}$ in the main program and in the function subroutine. The function subroutine allows more flexibility. Let us consider our general function to be of the form

$$f(t) = \sin(2\pi t). \quad (12.6)$$

Let us further define

$$y = f(x), \quad (12.7)$$

$$z = f\left(\frac{x}{2}\right). \quad (12.8)$$

Thus, with our general form for f , we expect

$$y = \sin(2\pi x), \quad (12.9)$$

$$z = \sin(\pi x). \quad (12.10)$$

This is illustrated with the program `ch12f.f90`

```
[powers@remote102 ame.20214]$ cat ch12f.f90
```

```
!-----
module myfunction                                !This is a module declaration
contains
real function f(t)                               !This is the function subroutine
implicit none
real, intent(in) :: t                           !Declare t as a real input variable
real :: pi
pi = 4.*atan(1.)
f = sin(2.*pi*t)
end function f
end module myfunction
!
!-----
!
program fofx                                     !This is the main program--it comes last
use myfunction                                  !The main program draws upon myfunction
implicit none
real :: x,y,z,xmin=0.,xmax=1.,dx
```

```

integer :: i,n
print*, 'enter number of points, n>1'
read*,n
dx = (xmax-xmin)/real(n-1.)
do i=1,n
    x = xmin + dx*(i-1.)
    y = f(x)                                !Here we call on the function subroutine
    z = f(x/2.)                             !Here we again call on the function subroutine
    print 100,x,y,z
    100 format(f12.6,2x,f12.6,2x,f12.6)
end do
end program fofx
!-----
[powers@remote102 ame.20214]$ ifort ch12f.f90
[powers@remote102 ame.20214]$ a.out
enter number of points, n>1
13
    0.000000      0.000000      0.000000
    0.083333      0.500000      0.258819
    0.166667      0.866025      0.500000
    0.250000      1.000000      0.707107
    0.333333      0.866025      0.866025
    0.416667      0.500000      0.965926
    0.500000      0.000000      1.000000
    0.583333     -0.500000      0.965926
    0.666667     -0.866025      0.866025
    0.750000     -1.000000      0.707107
    0.833333     -0.866025      0.500000
    0.916667     -0.500000      0.258819
    1.000000      0.000000      0.000000
[powers@remote102 ame.20214]$

```

12.2.3 Sharing data using module

Sometimes we wish to share data between the main program and the function subroutine. There are many ways to do this; a good way is to use the `module` option in a slightly different fashion. Let us say that in our example program for this chapter that we would like both the main program and the function to know what the value of π is. And we only want to compute it once. We can introduce a `module` at the beginning of a program to build shared data, and then import that data into the elements of the program that require it. An example is shown in `ch12g.f90`:

```
[powers@remote102 ame.20214]$ cat ch12g.f90
```

```

!-----
module commondata                                !This sets up data which will be shared.
implicit none
real :: pi=4.*atan(1.)
end module commondata
!
!-----
!
module myfunction                                !This is a module declaration
contains
real function f(t)                               !This is the function subroutine
use commondata                                  !Import the common data
implicit none
real, intent(in) :: t                           !Declare t as a real input variable
f = sin(2.*pi*t)                                !Employ the common data, pi
end function f
end module myfunction
!
!-----
!
program fofx                                     !This is the main program--it comes last
use commondata                                  !Import the common data
use myfunction                                  !The main program draws upon myfunction
implicit none
real :: x,y,z,xmin=0.,xmax=1.,dx
integer :: i,n
print*, pi                                     !print the common data
print*, 'enter number of points, n>1'
read*,n
dx = (xmax-xmin)/real(n-1.)
do i=1,n
  x = xmin + dx*(i-1.)
  y = f(x)                                     !Here we call on the function subroutine
  z = f(x/2.)                                 !Here we again call on the function subroutine
  print 100,x,y,z
  100 format(f12.6,2x,f12.6,2x,f12.6)
end do
end program fofx
!-----
[powers@remote102 ame.20214]$ ifort ch12g.f90
[powers@remote102 ame.20214]$ a.out
  3.141593

```

```

enter number of points, n>1
13
    0.000000      0.000000      0.000000
    0.083333      0.500000      0.258819
    0.166667      0.866025      0.500000
    0.250000      1.000000      0.707107
    0.333333      0.866025      0.866025
    0.416667      0.500000      0.965926
    0.500000      0.000000      1.000000
    0.583333     -0.500000      0.965926
    0.666667     -0.866025      0.866025
    0.750000     -1.000000      0.707107
    0.833333     -0.866025      0.500000
    0.916667     -0.500000      0.258819
    1.000000      0.000000      0.000000
[powers@remote102 ame.20214]$

```

Note that neither the main program nor the function subroutine constructed `pi`; moreover, `pi` was not declared in either. Instead, `pi` was declared and initialized in module `commondata`, and `commondata` was imported into the main program and the function via the command `use`. The notion of the module is discussed further in Ch. 21.

12.2.4 Splitting functions into separate files

It is sometimes convenient to split the main program and function subroutines into *separate files*. Then we can compile them one at a time if we want and combine them later. Let us take the main program to be in the file `ch12d.f90`, given next:

```

[powers@remote102 ame.20214]$ cat ch12d.f90
program fofx                                !This is the main program--it comes last
use myfunction                              !The main program draws upon myfunction
implicit none
real :: x,y,xmin=0.,xmax=1.,dx
integer :: i,n
print*,'enter number of points, n>1'
read*,n
dx = (xmax-xmin)/real(n-1.)
do i=1,n
    x = xmin + dx*(i-1.)
    y = f(x)                                !Here we call on the function subroutine
    print 100,x,y
    100 format(f12.6,2x,f12.6)
end do

```

```
end program fofx
[powers@remote102 ame.20214]$
```

Then let us take the function subroutine to reside in `ch12e.f90`:

```
[powers@remote102 ame.20214]$ cat ch12e.f90
module myfunction                                !This is a module declaration
contains
real function f(x)                               !This is the function subroutine
implicit none
real, intent(in) :: x                           !Declare x as a real input variable
real :: pi
pi = 4.*atan(1.)
f = sin(2.*pi*x)
end function f
end module myfunction
[powers@remote102 ame.20214]$
```

Now, we can do a different kind of compilation of the function subroutine module first via the command

```
[powers@remote102 ame.20214]$ ifort -c ch12e.f90
```

This `-c` option generates the binary file `ch12e.o`. It is binary, but it is not executable on its own. To work, it must be linked to a driver program. We can achieve the linkage to the driver program `ch12d.f90` by compiling `ch12d.f90` and informing it that it should be linked to `ch12e.o` via the compilation command

```
[powers@darrows2-p ame.20214]$ ifort ch12d.f90 ch12e.o
```

This command generates the binary executable `a.out`. Note that if we did not link to `ch12e.o`, the main program `ch12d.f90` would not compile, because it was missing `f(x)`. Here is the execution of `a.out` for 7 points:

```
[powers@darrows2-p ame.20214]$ a.out
enter number of points, n>1
7
    0.000000    0.000000
    0.166667    0.866025
    0.333333    0.866025
    0.500000    0.000000
    0.666667   -0.866025
    0.833333   -0.866025
    1.000000    0.000000
[powers@darrows2-p ame.20214]$
```

Another option is to use the `-c` option on both `.f90` files first and create two `.o` files, then use the compiler to merge them to create `a.out`:

```
[powers@remote102 ame.20214]$ ifort -c ch12d.f90
[powers@remote102 ame.20214]$ ifort -c ch12e.f90
[powers@remote102 ame.20214]$ ifort ch12d.o ch12e.o
[powers@remote102 ame.20214]$ a.out
  enter number of points, n>1
7
    0.000000      0.000000
    0.166667      0.866025
    0.333333      0.866025
    0.500000      0.000000
    0.666667     -0.866025
    0.833333     -0.866025
    1.000000      0.000000
[powers@remote102 ame.20214]$
```

12.2.5 Creation of executable script files

One advantage of the UNIX system is that one can create files which actually contain a list of commands and execute them at once. In the previous example, we had to compile `ch12e.f90` with the `-c` option, then link and compile this with `ch12d.f90`. Then we had to execute the binary executable with `a.out`. Let us create a single file `runch12de` which performs all these tasks:

```
[powers@remote102 ame.20214]$ cat runch12de
ifort -c ch12d.f90
ifort -c ch12e.f90
ifort ch12d.o ch12e.o
a.out
[powers@remote102 ame.20214]$
```

The file has our chores combined into a single list. We next have to *one time only* declare this file to be executable. In UNIX this is achieved via

```
[powers@darrow2-p ame.20214]$ chmod u+x runch12de
```

The so-called “change mode” command, `chmod`, changes permissions on files. We have changed the permission for the user, `u`, so as to add, `+`, the ability to execute, `x`, the file. So entering the new UNIX command `runch12de` compiles and executes, generating

```
[powers@darrow2-p ame.20214]$ runch12de
  enter number of points, n>1
```

8

0.000000	0.000000
0.142857	0.781832
0.285714	0.974928
0.428571	0.433883
0.571429	-0.433884
0.714286	-0.974928
0.857143	-0.781831
1.000000	0.000000

`[powers@darrow2-p ame.20214]$`

A more powerful way to build executable programs is through the UNIX utility **Make**, which we will not discuss further here, but offers many advantages to the interested user.

Chapter 13

Control structures

Read C&S, Chapter 13.

There are an important set of logical structures to control **Fortran** programs. We have seen some of these before. The most critical are based on either

- `if, then, endif`: a binary choice, or
- `do, enddo`: an iteration.

For binary choices, we also have the constructs `if, then, else if, endif`. For iterations, we have several constructs:

- `do i=1,n,`
- `do while,`
- `do ...if, then exit, enddo,`
- others.

A typical `if` statement will require a syntax for the condition to be met. For example, the English sentence, “If today is Thursday, then attend AME 20214,” has a **Fortran**-like structure

```
if (today is Thursday) then
attend AME 20214.
```

We need a syntax for English words like “is.” That syntax and others are listed in Table 13.1, adapted from C&S: An example of an `if`-based structure is obtained if we seek the real roots of the quadratic equation

$$ax^2 + bx + c = 0, \quad a, b, c \in \mathbb{R}. \quad (13.1)$$

Operator	Meaning	Type
<code>==</code>	equal to	relational
<code>/=</code>	not equal to	relational
<code>>=</code>	greater than or equal to	relational
<code><=</code>	less than or equal to	relational
<code><</code>	less than	relational
<code>></code>	greater than	relational
<code>.and.</code>	and	logical
<code>.or.</code>	or	logical
<code>.not.</code>	not	logical

Table 13.1: Control structures in **Fortran**.

We know the solution is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (13.2)$$

Moreover, we know there are two real roots if

$$b^2 - 4ac > 0, \quad (13.3)$$

there are two repeated real roots if

$$b^2 - 4ac = 0, \quad (13.4)$$

and there are no real roots if

$$b^2 - 4ac < 0. \quad (13.5)$$

Given a , b , and c , we should be able to construct a **Fortran** program to identify the real roots, if they exist, and to also tell us if no real roots exist. This is achieved in a small modification of a program given by C&S, found in `ch13a.f90`:

```
[powers@remote202 ame.20214]$ cat ch13a.f90
PROGRAM ch13a
IMPLICIT NONE
REAL :: a , b , c , term , root1 , root2
!
!   a b and c are the coefficients of the terms
!   a*x**2+b*x+c
!   find the roots of the quadratic, root1 and root2
!
```

```

PRINT*, ' GIVE THE COEFFICIENTS a, b AND c'
READ*,a,b,c
term = b*b - 4.*a*c
! if term < 0, roots are complex
! if term = 0, roots are equal
! if term > 0, roots are real and different
IF(term < 0.0)THEN
  PRINT*, ' ROOTS ARE COMPLEX'
ELSEIF(term > 0.0)THEN
  term = SQRT(term)
  root1 = (-b+term)/2./a
  root2 = (-b-term)/2./a
  PRINT*, ' ROOTS ARE ',root1,' AND ',root2
ELSE
  root1 = -b/2./a
  PRINT*, ' TWO EQUAL ROOTS, EACH IS ',root1
ENDIF
END PROGRAM ch13a
[powers@remote202 ame.20214]$ ifort ch13a.f90
[powers@remote202 ame.20214]$ a.out
  GIVE THE COEFFICIENTS a, b AND c
1 -3 2
  ROOTS ARE      2.000000      AND      1.000000
[powers@remote202 ame.20214]$

```

Note

- conditions lie inside the parentheses, e.g. `term > 0.0`.
- there is a logical structure to the `if`, `then`, `elseif`, `then`, `else`, `endif`.

We can also use logical constructs, such as `.and.` to effect. Let us say we want to plot a so-called “top hat” function:

$$f(x) = \begin{cases} 0, & x \in (-\infty, 1), \\ 1, & x \in [1, 2], \\ 0, & x \in [2, \infty). \end{cases} \quad (13.6)$$

The nature of the function $f(x)$ is that it has explicit conditional dependency on x which is appropriate for an `if`, `then` implementation in Fortran. We give such an implementation in `ch13b.f90`:

```

[powers@remote102 ame.20214]$ cat ch13b.f90

```

```

!-----
module myfunction                                !This is a module declaration

```

```

contains
real function f(t)                                !This is the function subroutine
implicit none
real, intent(in) :: t                             !Declare t as a real input variable
if(t<1.0.or.t>2.0) then
    f = 0.0
else
    f = 1.0
endif
end function f
end module myfunction
!
!-----
!
program fofx                                       !This is the main program--it comes last
use myfunction                                   !The main program draws upon myfunction
implicit none
real :: x,y,xmin=-3.,xmax=3.,dx
integer :: i,n
print*, 'enter number of points, n>1'
read*,n
dx = (xmax-xmin)/real(n-1.)
do i=1,n
    x = xmin + dx*(i-1.)
    y = f(x)                                     !Here we call on the function subroutine
    print 100,x,y
    100 format(f12.6,2x,f12.6)
end do
end program fofx
!-----
[powers@remote102 ame.20214]$ ifort ch13b.f90
[powers@remote102 ame.20214]$ a.out
enter number of points, n>1
11
-3.000000      0.000000
-2.400000      0.000000
-1.800000      0.000000
-1.200000      0.000000
-0.600000      0.000000
 0.000000      0.000000
 0.600000      0.000000
 1.200000      1.000000

```

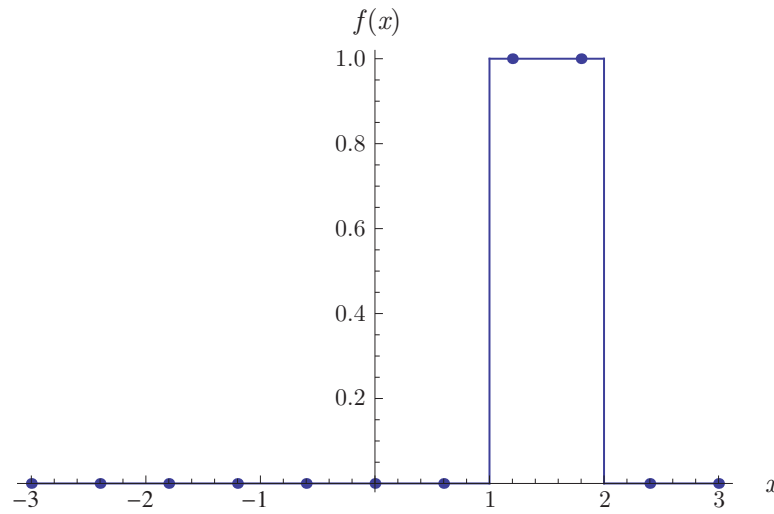


Figure 13.1: Plot of top hat function $f(x)$ for $x \in [-3, 3]$, discrete and exact values.

```

1.800000      1.000000
2.400000      0.000000
3.000000      0.000000
[powers@remote102 ame.20214]$

```

Note:

- We elected to use the function subroutine to evaluate f , here expressed generically as $f(t)$.
- Within the function subroutine, we used the conditional `if`, `then`, `else`, `endif` structure.
- The conditional for the `if` statement involved the Boolean logical operator `.or`.

We give a plot of the exact representation of $f(x)$ and the eleven discrete points we chose to evaluate in Fig. 13.1.

We can illustrate a `do while` structure with the following problem. Use the Euler method to solve the ordinary differential equation

$$\frac{dy}{dt} = \lambda y, \quad y(0) = y_o, \quad t \in [0, t_{stop}], \quad (13.7)$$

for $\lambda = -1$, $y_o = 1$, $t_{stop} = 1$. Euler's method has us approximate the first derivative as

$$\frac{dy}{dt} \sim \frac{y_{n+1} - y_n}{\Delta t}, \quad (13.8)$$

and evaluate the right hand side of the differential equation at the n time step. This gives rise to

$$\frac{y_{n+1} - y_n}{\Delta t} = \lambda y_n, \quad (13.9)$$

$$y_{n+1} = y_n + \lambda \Delta t y_n. \quad (13.10)$$

We choose $\Delta t = 0.1$. A Fortran implementation of this is given in `ch13c.f90`:

```
[powers@remote202 ame.20214]$ cat ch13c.f90
program euler
implicit none
integer :: n
real, parameter :: tstop=1., yo = 1., lambda = -1., dt=1.e-1
real :: y,ynew,t
t = 0.
y = yo
n=0
print*,t,y,yo*exp(lambda*t)
do while (t<tstop)                                !do while structure for loop
  ynew = y+lambda*dt*y
  n = n+1
  t = dt*n
  y = ynew
  print*,t,y,yo*exp(lambda*t)
enddo
print*, 'error ',y-yo*exp(lambda*t)
end program euler
[powers@remote202 ame.20214]$ ifort ch13c.f90
[powers@remote202 ame.20214]$ a.out
0.000000E+00    1.000000    1.000000
0.1000000    0.9000000    0.9048374
0.2000000    0.8100000    0.8187308
0.3000000    0.7290000    0.7408182
0.4000000    0.6561000    0.6703200
0.5000000    0.5904900    0.6065307
0.6000000    0.5314410    0.5488116
0.7000000    0.4782969    0.4965853
0.8000000    0.4304672    0.4493290
0.9000000    0.3874205    0.4065697
1.000000    0.3486784    0.3678795
error -1.9201010E-02
[powers@remote202 ame.20214]$
```

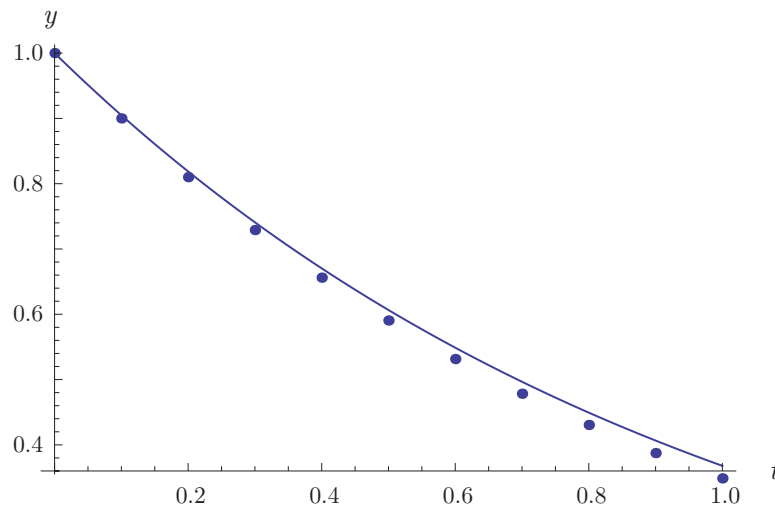


Figure 13.2: Plot of the exact solution to $dy/dt = -y$, $y(0) = 1$ ($y = \exp(-t)$), along with its approximation from the Euler method with $\Delta t = 0.1$.

We note that the `do while` structure has been implemented, and it is an obvious and convenient program control structure for this problem. We give a plot of the exact representation of $y(x)$ and the eleven discrete points we chose to evaluate with the Euler method in Fig. 13.2.

Chapter 14

Characters

Read C&S, Chapter 14.

One should read the text of C&S to learn further about characters within **Fortran**.

Chapter 15

Complex

Read CES, Chapter 15.

Complex mathematics is a rich and varied subject. At its foundation lays the number i where

$$i \equiv \sqrt{-1}. \quad (15.1)$$

If we consider numbers z to be complex, then we can think of them as

$$z = x + iy. \quad (15.2)$$

Here x and y are real numbers. We say mathematically,

$$z \in \mathbb{C}^1, \quad x \in \mathbb{R}^1, \quad y \in \mathbb{R}^1. \quad (15.3)$$

where $\mathbb{C}$ and $\mathbb{R}$ denote the complex and real numbers, respectively, and the superscript 1 denotes scalars. The real part of z is defined as

$$\Re(z) = \Re(x + iy) = x. \quad (15.4)$$

The **Fortran** equivalent is the function **real**:

$$\Re(z) = \text{real}(z). \quad (15.5)$$

The imaginary part of z is defined as

$$\Im(z) = \Im(x + iy) = y. \quad (15.6)$$

The **Fortran** equivalent is the function **aimag**:

$$\Im(z) = \text{aimag}(z). \quad (15.7)$$

Many compilers also accept **imag**.

If $z = x + iy$, then the so-called *complex conjugate*, $\bar{z}$, is defined as

$$\bar{z} \equiv x - iy. \quad (15.8)$$

The **Fortran** command for taking the complex conjugate of a complex variable **z** is

$$\bar{z} = \overline{(x + iy)} = \text{conjg}(z). \quad (15.9)$$

Note that

$$z\bar{z} = (x + iy)(x - iy) = x^2 + ixy - ixy - (i)^2y^2 = x^2 + y^2. \quad (15.10)$$

We call this the square of magnitude of the complex number:

$$|z|^2 = z\bar{z} = x^2 + y^2, \quad |z|^2 = \mathbb{R}^1. \quad (15.11)$$

The magnitude of a complex number is obviously a positive real number. We can also get the magnitude of a complex number by the command **abs(z)**.

For $z = 3 + 4i$, this is illustrated in **ch15a.f90**:

```
[powers@remote102 ame.20214]$ cat ch15a.f90
program cplex
complex :: z,zconj,zmag,zabs
real :: x,y
z = (3.,4.)
zconj = conjg(z)
zmag = sqrt(z*zconj)
zabs = abs(z)
x = real(z)
y = aimag(z)
print*, 'z = x+iy'      = ',z
print*, 'zbar'          = ',zconj
print*, 'sqrt(x^2+y^2)' = ',zmag
print*, '|z|'           = ',zabs
print*, 'Re(z) = x'     = ',x
print*, 'Im(z) = y'     = ',y
end program cplex
[powers@remote102 ame.20214]$ ifort ch15a.f90
[powers@remote102 ame.20214]$ a.out
z = x+iy      = (3.000000,4.000000)
zbar          = (3.000000,-4.000000)
sqrt(x^2+y^2) = (5.000000,0.0000000E+00)
|z|           = (5.000000,0.0000000E+00)
Re(z) = x     = 3.000000
Im(z) = y     = 4.000000
[powers@remote102 ame.20214]$
```

Note, we have given **Fortran** output for the mathematical operations

1. $z = 3 + 4i$,
2. $\bar{z} = 3 - 4i$,
3. $|z| = \sqrt{z\bar{z}} = \sqrt{(3 + 4i)(3 - 4i)} = \sqrt{3^2 + 4^2} = \sqrt{25} = 5 + 0i$.
4. $\Re(z) = \Re(3 + 4i) = 3$.
5. $\Im(z) = \Im(3 + 4i) = 4$.

Obviously operations which have restricted domains for real numbers have less restricted domains for complex numbers; as an example $\sqrt{z}$ is defined for all $z = x + iy$ with $x \in (-\infty, \infty)$, $y \in (-\infty, \infty)$. Moreover $\ln z$ is similarly defined for all z , as are all trigonometric functions. **Fortran** is well equipped to handle these operations when variables are declared to be complex. We give an example program in `ch15b.f90`:

```
[powers@remote202 ame.20214]$ cat ch15b.f90
program cplex
complex :: z1,z2,z3

z1 = (-2.,0.)
z2 = (0.,1.)
z3 = (1.,1.)

print*, 'sqrt(-2) = ', sqrt(z1)
print*, 'ln(-2)    = ', log(z1)
print*, 'sin(-2)   = ', sin(z1)

print*, 'sqrt(i)   = ', sqrt(z2)
print*, 'ln(i)     = ', log(z2)
print*, 'sin(i)    = ', sin(z2)

print*, 'sqrt(1+i) = ', sqrt(z3)
print*, 'ln(1+i)   = ', log(z3)
print*, 'sin(1+i)  = ', sin(z3)

end program cplex
[powers@remote202 ame.20214]$ ifort ch15b.f90
[powers@remote202 ame.20214]$ a.out
sqrt(-2) = (0.0000000E+00,1.414214)
ln(-2)    = (0.6931472,3.141593)
sin(-2)   = (-0.9092974,0.0000000E+00)
sqrt(i)   = (0.7071068,0.7071068)
```

```

ln(i)    = (0.0000000E+00,1.570796)
sin(i)   = (0.0000000E+00,1.175201)
sqrt(1+i) = (1.098684,0.4550899)
ln(1+i)  = (0.3465736,0.7853982)
sin(1+i) = (1.298458,0.6349639)
[powers@remote202 ame.20214]$

```

We see

1. for $z = -2 + 0i$ that

- (a) $\sqrt{z} = \sqrt{-2} = \sqrt{2}i$,
- (b) $\ln z = \ln(-2) = \ln(2) + i\pi$, and
- (c) $\sin(z) = \sin(-2) = -\sin(2)$.

2. for $z = 0 + i$ that

- (a) $\sqrt{z} = \sqrt{i} = \sqrt{2} + \sqrt{2}i$,
- (b) $\ln z = \ln(i) = i\frac{\pi}{2}$, and
- (c) $\sin(z) = \sin(i) = i\frac{e^1 - e^{-1}}{2}$.

3. for $z = 1 + i$ that

- (a) $\sqrt{z} = \sqrt{1+i} = 2^{1/4}(\cos(\pi/8) + i\sin(\pi/8)) = 1.09868 + 0.45509i$,
- (b) $\ln z = \ln(1+i) = \frac{1}{2}\ln(2) + i\frac{\pi}{4} = 0.346574 + 0.785398i$, and
- (c) $\sin(z) = \sin(1+i) = 1.29846 + 0.634964i$.

We lastly note that inverse trigonometric functions have a clean mathematical definition for complex numbers. For instance, it can be shown that

$$\sin^{-1} z = -i \ln \left(iz + \sqrt{1 - z^2} \right), \quad z \in \mathbb{C}^1, \quad (15.12)$$

so that, for example

$$\sin^{-1}(2) = -i \ln \left((2 + \sqrt{3})i \right) = \frac{\pi}{2} - \ln(2 + \sqrt{3})i = 1.5708 - 1.31696i. \quad (15.13)$$

Note also that the inverse sine is not unique as the following are all solutions:

$$\sin^{-1}(2) = 1.5708 + 2n\pi - 1.31696i, \quad n = \dots, -2, -1, 0, 1, 2, \dots \quad (15.14)$$

Note to achieve real values of $\sin^{-1} x$, we must restrict x so that $x \in \mathbb{R}^1$. When we go outside of the domain $x \in [-1, 1]$, the range is extended to the complex numbers.

Now one might suppose **Fortran**'s `asin` could handle complex numbers, but that this is only available in the **Fortran** 2008 standard, not yet invoked on our **ifort** compiler. This is not a serious problem as the interested user can always define a function which serves the same purpose. This is seen in `ch15c.f90`:

```

[powers@remote202 ame.20214]$ cat ch15c.f90
program arcsine
implicit none
complex :: z,arcsin
z = (2.,0.)
print*, 'arcsin(',z,')= ',arcsin(z)           !call upon our function arcsin
end program arcsine

complex function arcsin(t)
implicit none
complex :: t,i = (0.,1.)
arcsin = -i*log(t*i+sqrt(1-t**2))
end function arcsin
[powers@remote202 ame.20214]$ ifort ch15c.f90
[powers@remote202 ame.20214]$ a.out
arcsin( (2.000000,0.0000000E+00) )= (1.570796,-1.316958)
[powers@remote202 ame.20214]$

```

Note

- We simply defined our own function subroutine `arcsin` and gave it the proper definition, consistent with that of mathematics.
- When exercised on $z = 2 + 0i$, our program returned the value, $\sin^{-1}(2) = 1.570796 - 1.316958i$. This value is acceptable, though we must realize that in general others could be as well.

Chapter 16

Logical

Read C&S, Chapter 16.

The interested reader can consult the text of C&S.

Chapter 17

Introduction to derived types

Read C&S, Chapter 17.

The interested reader can consult C&S's Chapter 17 for a discussion of derived types.

Chapter 18

An introduction to pointers

Read C&S, Chapter 18.

The interested reader can consult C&S for a discussion of pointers.

Chapter 19

Introduction to subroutines

Read CES, Chapter 19.

Many times we have a task that is composed of a sequence of steps that need to be repeated many times. For an efficient code, it is to our advantage to write a general subroutine that can handle tasks which must be repeated.

19.1 Simple subroutine

Let us begin with a problem that is probably too simple to justify a subroutine, but which will illustrate the syntax. Let us say that we wish to convert from Cartesian to polar coordinates via

$$r = +\sqrt{x^2 + y^2}, \quad (19.1)$$

$$\theta = \arctan\left(\frac{y}{x}\right). \quad (19.2)$$

and that we need to calculate r and θ for arbitrary x and y . Because there are two outputs, r and θ , a single user-defined function subroutine is inappropriate. While we could employ two user-defined functions for this, let us instead employ a single user-defined subroutine. We show such a subroutine and its use in the program `ch19a.f90`.

```
[powers@darrows1-p ame.20214]$ cat ch19a.f90
```

```
!-----  
module xyrt                                !beginning of subroutine module  
contains  
subroutine crt(xx,yy,rr,ttheta) !beginning of actual subroutine  
implicit none  
real, intent(out) :: rr          !useful optional statement reserving rr for output  
real, intent(out) :: ttheta      !useful optional statement reserving qq for output  
real, intent(in)  :: xx          !useful optional statement reserving xx for input
```

```

real, intent(in)  :: yy      !useful optional statement reserving yy for input
rr = sqrt(xx**2+yy**2)
ttheta = atan(yy/xx)
end subroutine crt
end module xyrt
!
!-----
!
program ch19a
use xyrt                !import the module with the subroutine
implicit none
real:: x=3., y=4., r, theta
call crt(x,y,r,theta)   !call the actual subroutine
print*, 'x = ', x
print*, 'y = ', y
print*, 'r = ', r
print*, 'theta = ', theta
end program ch19a
!-----
[powers@darrow1-p ame.20214]$ ifort ch19a.f90
[powers@darrow1-p ame.20214]$ a.out
x =    3.000000
y =    4.000000
r =    5.000000
theta =  0.9272952
[powers@darrow1-p ame.20214]$

```

Note:

- The module containing the subroutine is placed first in the program structure.
- The subroutine uses a set of dummy variable names, **xx**, **yy**, **rr**, **ttheta**, which we choose to be similar to those of the actual variables. We could have chosen them to be identical. No matter which choice we made, those variables are *local* variables to the subroutine. They are passed back to the main program where they take on the value of the names assigned in the main program. So **rr** gets assigned in the subroutine, and it is passed back to the main program, where it is assigned to the variable **r**. The same holds for θ , represented as **ttheta** and **theta**.
- It is good, but optional, practice to define variables as either input or output variables within the subroutine. Here, input variables were defined with **intent(in)** as input variables only. This then requires that the subroutine not change their value. Output variables were defined with **intent(out)** and were not given initial values.

- Though not used here, variables which are allowed values on input and can be changed for output can be assigned `intent(inout)`.
- The main program, which appears last, is executed first.
- The main program assigns values to `x` and `y`, but leaves `r` and `theta` unassigned initially.
- The main program refers to the subroutine module at its beginning.
- The first real executable line of the main program is to call upon the subroutine, where `r` gets its value.
- After execution of the subroutine, control returns to the main program.
- The main program then prints all variables to the screen.

19.2 Subroutines for solution of a quadratic equation

We next consider a subroutine to aid in solving for the real roots of the quadratic equation

$$ax^2 + bx + c = 0. \quad (19.3)$$

We know the solution to be

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (19.4)$$

We give a program which is an adaptation of one found in C&S to solve this problem with the aid of two subroutines, `ch19b.f90`:

```
[powers@remote102 ame.20214]$ cat ch19b.f90
module interact_module
contains
  subroutine interact(a,b,c)
    implicit none
    real, intent (out) :: a
    real, intent (out) :: b
    real, intent (out) :: c

    print *, ' type in the coefficients a, b and c '
    read (unit=*,fmt=*) a, b, c
  end subroutine interact
end module interact_module

module solve_module
```

```

contains
  subroutine solve(e,f,g,root1,root2,ifail)
    implicit none
    real, intent (in) :: e
    real, intent (in) :: f
    real, intent (in) :: g
    real, intent (out) :: root1
    real, intent (out) :: root2
    integer, intent (inout) :: ifail
! local variables
    real :: term
    real :: a2

    term = f*f - 4.*e*g
    a2 = e*2.0
! if term < 0, roots are complex
    if (term<0.0) then
      ifail = 1
    else
      term = sqrt(term)
      root1 = (-f+term)/a2
      root2 = (-f-term)/a2
    end if
  end subroutine solve
end module solve_module

program roots
use interact_module
use solve_module
implicit none
! simple example of the use of a main program and two
! subroutines. one interacts with the user and the
! second solves a quadratic equation,
! based on the user input.
real :: p, q, r, root1, root2
integer :: ifail = 0

  call interact(p,q,r)
  call solve(p,q,r,root1,root2,ifail)
  if (ifail==1) then
    print *, ' complex roots'
    print *, ' calculation abandoned'
  end if
end program roots

```

```

        else
            print *, ' roots are ', root1, ' ', root2
        end if
end program roots
[powers@remote102 ame.20214]$ ifort ch19b.f90
[powers@remote102 ame.20214]$ a.out
    type in the coefficients a, b and c
1 -4 3
    roots are      3.000000      1.000000
[powers@remote102 ame.20214]$

```

Note:

- For the variables a , b and c , the main program uses `p`, `q`, and `r`, while subroutine `interact` uses `a`, `b`, and `c`, and subroutine `solve` uses `e`, `f`, and `g`.
- Data input is handled by subroutine `interact`.
- The output of subroutine `interact` is used as input for subroutine `solve`.
- The variable `ifail` is a flag which lets the main program know whether or not the calculation was such that real roots were found. The variable `ifail` is initialized to zero in the main program, and is an input to subroutine `solve`. If the discriminant is negative, the roots are complex, and the subroutine sets `ifail` to 1. Note that `ifail` is both an input and an output to the subroutine, and is declared as such with `intent(inout)`.

19.3 Subroutines for solving systems of ordinary differential equations

Let us develop a program to solve a general system of N non-linear ordinary differential equations using a first order Euler method. We briefly review the relevant mathematics.

We consider the following initial value problem:

$$\frac{dy_1}{dt} = f_1(y_1, y_2, \dots, y_N), \quad y_1(0) = y_{1o}, \quad (19.5)$$

$$\frac{dy_2}{dt} = f_2(y_1, y_2, \dots, y_N), \quad y_2(0) = y_{2o}, \quad (19.6)$$

$$\vdots \quad (19.7)$$

$$\frac{dy_N}{dt} = f_N(y_1, y_2, \dots, y_N), \quad y_N(0) = y_{No}. \quad (19.8)$$

Here $f_1, f_2, \dots, f_N$ are general non-linear algebraic functions. A simple finite difference approximation of these equations gives

$$\frac{y_1^{n+1} - y_1^n}{\Delta t} = f_1(y_1^n, y_2^n, \dots, y_N^n), \quad y_1^1 = y_{1o}, \quad (19.9)$$

$$\frac{y_2^{n+1} - y_2^n}{\Delta t} = f_2(y_1^n, y_2^n, \dots, y_N^n), \quad y_2^1 = y_{2o}, \quad (19.10)$$

$$\vdots \quad (19.11)$$

$$\frac{y_N^{n+1} - y_N^n}{\Delta t} = f_N(y_1^n, y_2^n, \dots, y_N^n), \quad y_N^1 = y_{No}. \quad (19.12)$$

Here, the superscripts are *not* exponents; instead, they are simply indices for counting. We can solve for the new values at $n + 1$ in terms of the old at n directly:

$$y_1^{n+1} = y_1^n + \Delta t f_1(y_1^n, y_2^n, \dots, y_N^n), \quad (19.13)$$

$$y_2^{n+1} = y_2^n + \Delta t f_2(y_1^n, y_2^n, \dots, y_N^n), \quad (19.14)$$

$$\vdots \quad (19.15)$$

$$y_N^{n+1} = y_N^n + \Delta t f_N(y_1^n, y_2^n, \dots, y_N^n). \quad (19.16)$$

If we repeat this process many times for sufficiently small values of Δt , we get a good approximation to the continuous behavior of $y_1(t), y_2(t), \dots, y_N(t)$.

Now we wish to have a program with two desirable features:

- It can handle arbitrary functions $f_1, f_2, \dots, f_N$.
- It can handle arbitrary integration algorithms, say Euler, or say the common Runge-Kutta method. For example, the two-step Runge-Kutta method is as follows:

$$y_i^{n+1/2} = y_i^n + \frac{\Delta t}{2} f_i(y_j^n), \quad i, j = 1, \dots, N, \quad (19.17)$$

$$y_i^{n+1} = y_i^n + \Delta t f_i(y_j^{n+1/2}), \quad i, j = 1, \dots, N. \quad (19.18)$$

19.3.1 One linear ordinary differential equation

Let us apply this strategy to a “system” comprised of $N = 1$ equations:

$$\frac{dy_1}{dt} = \lambda y_1, \quad y_1(0) = y_{1o}. \quad (19.19)$$

We will take $\lambda = -1$, $y_{1o} = 1$. The exact solution is given by

$$y_1(t) = e^{-t}. \quad (19.20)$$

Our general code for this problem is in `ch19c.f90`.

```

[powers@darrow2-p ame.20214]$ cat ch19c.f90
!-----
!
module commondata
integer, parameter :: neq=1           !set the number of equations
integer, parameter :: nt=10          !set the number of time steps
real, parameter :: tstop = 1.         !set the stop time
real, dimension(1:neq) :: yic=(/1./) !set the initial conditions
end module commondata
!
!-----
!
module rhs
! This subroutine in this module evaluates the right hand side of the
! system of neq ordinary differential equations.
contains
subroutine fcn(t,y,ydot)
use commondata
implicit none
real, intent(in) :: t
real, intent(in), dimension(1:neq) :: y
real, intent(out), dimension(1:neq) :: ydot
real, parameter :: lambda=-1.
ydot(1) = lambda*y(1)
end subroutine fcn
end module rhs
!
!-----
!
module ode1module
! The subroutine in this module is a generic algorithm for
! the first order explicit Euler method
! for solving the system of neq ordinary
! differential equations evaluated by
! subroutine fcn.
contains
subroutine ode1(y,t,tout)
use commondata
use rhs
implicit none
integer :: i
real, intent(inout), dimension(1:neq) :: y

```

```

real, intent(inout) :: t
real, intent(in) :: tout
real, dimension(1:neq) :: ydot
real :: dt
dt = tout-t
call fcn(t,y,ydot)          !estimate ydot at t
do i=1,neq
  y(i) = y(i)+dt*ydot(i)    !estimate y at t
enddo
t = tout                    !update t
end subroutine ode1
end module ode1module
!
!-----
!
module ode2module
! The subroutine in this module is a generic algorithm for
! the second order Runge-Kutta method for
! solving the system of neq ordinary differential
! equations evaluated by subroutine fcn.
contains
subroutine ode2(y,t,tout)
use commondata
use rhs
implicit none
integer :: i
real, intent(inout), dimension(1:neq) :: y
real, dimension(1:neq) :: yi,ydot
real, intent(inout) :: t
real, intent(in) :: tout
real :: dt
dt = tout-t
call fcn(t,y,ydot)          !estimate ydot at t
do i=1,neq
  yi(i) = y(i)+dt*ydot(i)/2. !estimate y at t+dt/2
enddo
call fcn(t+dt/2.,yi,ydot)   !estimate ydot at t+dt/2
do i=1,neq
  y(i) = y(i) + dt*ydot(i)  !estimate y at t
enddo
t = tout                    !update t
end subroutine ode2

```

```

end module ode2module
!
!-----
!
program solveode
! This is the main driver program for solving
! a generic set of neq differential equations.
use commondata
use ode1module
use ode2module
implicit none
integer :: i
real, dimension(1:neq) :: y
real :: t,tout,dt
open(10,file = 'output.dat')
t = 0.                                !initialize t
dt = tstop/real(nt)                   !determine dt
do i=1,neq
    y(i) = yic(i)                     !assign y to its initial condition
enddo
print*,t,y                            !output results
write(10,*)t,y
do i=1,nt                             !main loop in time
    tout = t + dt                     !set incremental end time
    call ode1(y,t,tout)               !update y and t
    print*,t,y                        !output the results
    write(10,*)t,y
enddo
print*, 'error = ',y(1) - exp(-t)
end program solveode
[powers@darrows2-p ame.20214]$ ifort ch19c.f90
[powers@darrows2-p ame.20214]$ a.out
0.0000000E+00    1.000000
0.1000000      0.9000000
0.2000000      0.8100000
0.3000000      0.7290000
0.4000000      0.6561000
0.5000000      0.5904900
0.6000000      0.5314410
0.7000000      0.4782969
0.8000001      0.4304672
0.9000001      0.3874205

```

```

1.000000      0.3486784
error = -1.9200981E-02
[powers@darrows2-p ame.20214]$

```

Note:

- There is a module for shared data, `commondata`.
- There are three subroutine modules:
 - `rhs` which evaluates the right hand sides of the differential equations,
 - `ode1module` which embodies the Euler method, and
 - `ode2module` which embodies the second order Runge-Kutta method.
- The three subroutine modules each `contain` a subroutine.
- Some subroutines `use` modules. Here `ode1` and `ode2` both need to call upon `fcn` to evaluate the right hand side of the function. Also the main driver program, `solveode`, may call on either `ode1` or `ode2`. It does not choose to call upon `fcn`.
- The main program calls upon either `ode1` or `ode2` to advance the solution from `t` to `tout`. Both `y` and `t` enter the solver with old values and return with new values.
- The particularities of the mathematics problem to be solved is confined to the `commondata` and `rhs` modules. The driver program as well as the Euler and Runge-Kutta solvers are general.
- When we execute with the Euler solver `ode1`, we get an error at $t = 1$ of $-1.9200981 \times 10^{-2}$.
- If we do nothing but change to `ode2`, the error changes to 6.6155195×10^{-4} . Thus, a simple change to our algorithm gave us two orders of magnitude improvement in the error.

We plot the exact solution along with the approximations of the Euler and two-step Runge-Kutta methods in Fig. 19.1.

19.3.2 Three non-linear ordinary differential equations

Let us apply this method to a problem with some interesting dynamics. Consider the non-linear forced mass-spring-damper system of

$$m \frac{d^2 y}{dt^2} + b \frac{dy}{dt} + k_1 y + k_2 y^3 = F_o \cos \nu t, \quad y(0) = y_o, \quad \frac{dy}{dt}(0) = 0. \quad (19.21)$$

This is also known as a forced, damped, Duffing equation. Here y is the position, y_o is the initial position, m is the mass, b is the damping coefficient, k_1 is the linear spring constant,

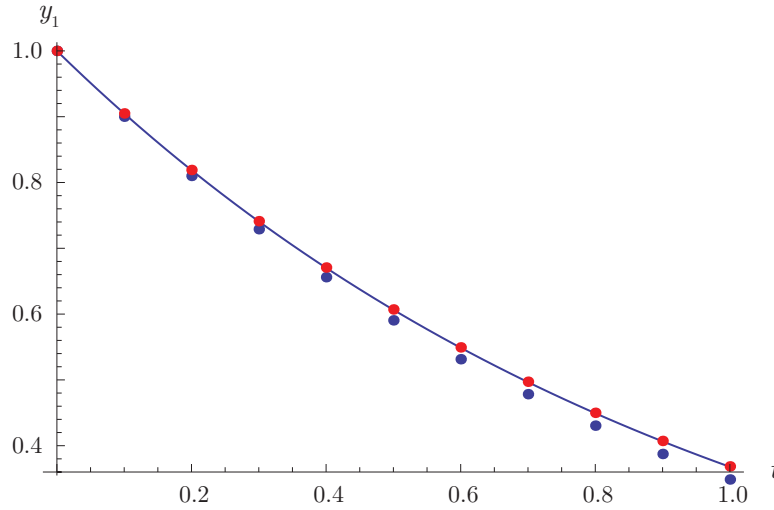


Figure 19.1: Plot of $y_1(t)$, solid line is the exact solution, blue dots give the finite difference solution with the first order Euler method, red dots give the finite difference solution with the second order Runge-Kutta method.

k_2 is the *non-linear* spring constant, F_o is the amplitude of the forcing function, and ν is the frequency of forcing. Let us do a useful scaling exercise, common in engineering. Our goal is to achieve more universality in our results. Let us define a dimensionless distance y^* by scaling y by its initial value y_o :

$$y^* = \frac{y}{y_o}. \quad (19.22)$$

Thus, we might expect y^* to have a value around 1; we say that $y^* = \mathcal{O}(1)$, or of “order one.” There are many choices of how to scale time. Let us make an obvious choice of

$$t^* = \nu t. \quad (19.23)$$

Since the frequency ν has units of $\text{Hz} = 1/\text{s}$ and time t has units of seconds, t^* will be dimensionless. We make these substitutions into our system of equations, Eqs. (19.21), and get

$$m y_o \nu^2 \frac{d^2 y^*}{dt^{*2}} + b y_o \nu \frac{dy^*}{dt^*} + k_1 y_o y^* + k_2 y_o^3 y^{*3} = F_o \cos t^*, \quad y_o y^*(0) = y_o, \quad y_o \nu \frac{dy}{dt}(0) = 0. \quad (19.24)$$

Now let us scale the second order differential equation by the constant $m y_o \nu^2$, and make obvious reductions to the initial conditions to get

$$\frac{d^2 y^*}{dt^{*2}} + \frac{b}{m \nu} \frac{dy^*}{dt^*} + \frac{k_1}{m \nu^2} y^* + \frac{k_2 y_o^2}{m \nu^2} y^{*3} = \frac{F_o}{m y_o \nu^2} \cos t^*, \quad y^*(0) = 1, \quad \frac{dy}{dt}(0) = 0. \quad (19.25)$$

Let us take

$$\delta \equiv \frac{b}{m\nu}, \quad \beta \equiv \frac{k_1}{m\nu^2}, \quad \alpha = \frac{k_2 y_o^2}{m\nu^2}, \quad f = \frac{F_o}{my_o\nu^2}. \quad (19.26)$$

Each of these system parameters, δ , β , α , and f are *dimensionless*. Physically, they represent the ratio of various forces. For example, δ is the ratio of the damping force to the so-called “inertia” of the system:

$$\delta = \frac{by_o\nu}{my_o\nu^2} = \frac{\text{damping force}}{\text{inertia}} = \frac{b}{m\nu}. \quad (19.27)$$

The term β is the ratio of the linear spring force to the inertial force; α is the ratio of the non-linear spring force to the inertial force, and f is the ratio of the driving force to the inertia. Thus our governing equation becomes

$$\frac{d^2 y^*}{dt^{*2}} + \delta \frac{dy^*}{dt^*} + \beta y^* + \alpha y^{*3} = f \cos t^*, \quad y^*(0) = 1, \quad \frac{dy^*}{dt^*}(0) = 0. \quad (19.28)$$

For convenience, let us drop the $*$ notation, and simply realize that all quantities are dimensionless. So we have

$$\frac{d^2 y}{dt^2} + \delta \frac{dy}{dt} + \beta y + \alpha y^3 = f \cos t, \quad y(0) = 1, \quad \frac{dy}{dt}(0) = 0. \quad (19.29)$$

Now let us take

$$y_1 \equiv y, \quad (19.30)$$

$$y_2 \equiv \frac{dy}{dt}. \quad (19.31)$$

Thus, Eq. (19.29) can be written as

$$\frac{dy_1}{dt} = y_2, \quad y_1(0) = 1, \quad (19.32)$$

$$\frac{dy_2}{dt} = -\beta y_1 - \delta y_2 - \alpha y_1^3 + f \cos t, \quad y_2(0) = 0. \quad (19.33)$$

Lastly, let us define $t = s = y_3$, so that $dt/ds = dy_3/ds = 1$ with $t(s=0) = y_3(s=0) = 0$. Then, replacing t by y_3 , and multiplying right and left sides of our system by $dt/ds = 1$, we get

$$\frac{dt}{ds} \frac{dy_1}{dt} = (1)(y_2), \quad (19.34)$$

$$\frac{dt}{ds} \frac{dy_2}{dt} = (1)(-\beta y_1 - \delta y_2 - \alpha y_1^3 + f \cos y_3), \quad (19.35)$$

$$\frac{dt}{ds} \frac{dy_3}{dt} = (1)1. \quad (19.36)$$

The chain rule then gives

$$\frac{dy_1}{ds} = y_2 \quad (19.37)$$

$$\frac{dy_2}{ds} = -\beta y_1 - \delta y_2 - \alpha y_1^3 + f \cos y_3, \quad (19.38)$$

$$\frac{dy_3}{ds} = 1. \quad (19.39)$$

Since $s = t$, we can also say

$$\frac{dy_1}{dt} = y_2, \quad (19.40)$$

$$\frac{dy_2}{dt} = -\beta y_1 - \delta y_2 - \alpha y_1^3 + f \cos y_3, \quad (19.41)$$

$$\frac{dy_3}{dt} = 1. \quad (19.42)$$

This system has some interesting mathematical properties when we take the parameters of $\alpha = 1$, $\beta = -1$, $\delta = 0.22$, $f = 0.3$. This is a highly unusual physical system, as the linear spring constant is in fact negative!

To simulate this function, we simply modify our `commondata rhs` modules so that they are as given in the complete `ch19d.f90`:

```
!-----
!  
module commondata  
  integer, parameter :: neq=3                !set the number of equations  
  integer, parameter :: nt=100000           !set the number of time steps  
  real, parameter :: tstop = 200.           !set the stop time  
  real, dimension(1:neq) :: yic=(/1.,0.,0./) !set the initial conditions  
end module commondata  
!  
!-----  
!  
module rhs  
  ! This subroutine in this module evaluates the right hand side of the  
  ! system of neq ordinary differential equations.  
contains  
  subroutine fcn(t,y,ydot)  
    use commondata  
    implicit none  
    real, intent(in) :: t  
    real, intent(in), dimension(1:neq) :: y  
    real, intent(out), dimension(1:neq) :: ydot
```

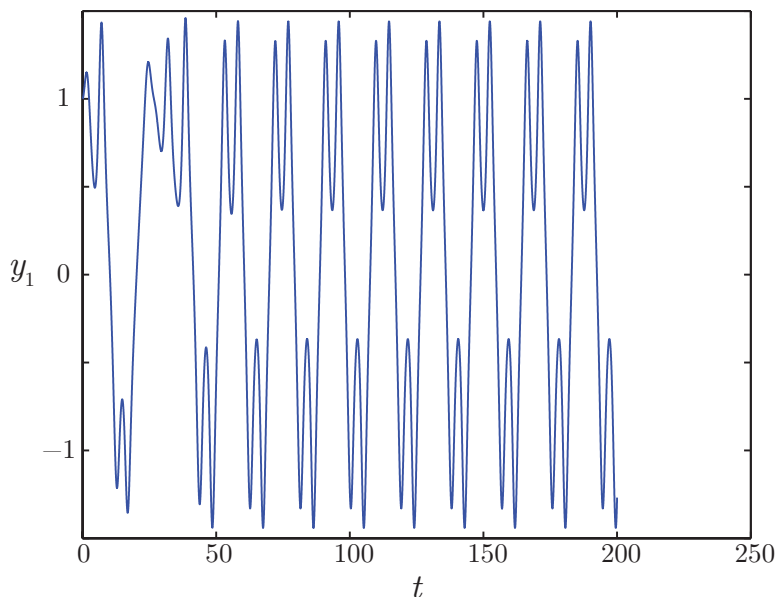


Figure 19.2: Plot of prediction of position versus time, $y_1(t)$, for the forced, damped Duffing equation for $t \in [0, 200]$; $\alpha = 1$, $\beta = -1$, $\delta = 0.22$, $f = 0.3$.

```

real, parameter :: beta=-1.,delta=0.22,alpha=1.,f=0.3
ydot(1) = y(2)
ydot(2) = -beta*y(1)-delta*y(2)-alpha*y(1)**3+f*cos(y(3))
ydot(3) = 1.
end subroutine fcn
end module rhs
!
!-----
!
```

We also select $y_1(0) = 1$, $y_2(0) = 0$, and $y_3(0) = 0$, and integrate for $t \in [0, 200]$ with $\Delta t = 0.02$. Now $y_3(t) = t$ is not very interesting. But $y_1(t) = y(t)$ is interesting. We plot the position (y_1) versus time ($y_3 = t$) in Fig. 19.2. We see there is an initial transient and then a relaxation to a cyclic behavior for $t > 100$. We then plot the velocity (y_2) versus time ($y_3 = t$) in Fig. 19.3. We again see there is an initial transient and then a relaxation to a cyclic behavior for $t > 100$. We plot the position (y_1) versus velocity (y_2) in Fig. 19.4. Figure 19.4 was for $t \in [0, 200]$. If we only plot the long time behavior, say $t \in [100, 200]$, we find an interesting result. For this late time domain, we plot the position versus velocity in Fig. 19.5. We see that the solution has become periodic; that is, it truly repeats itself.

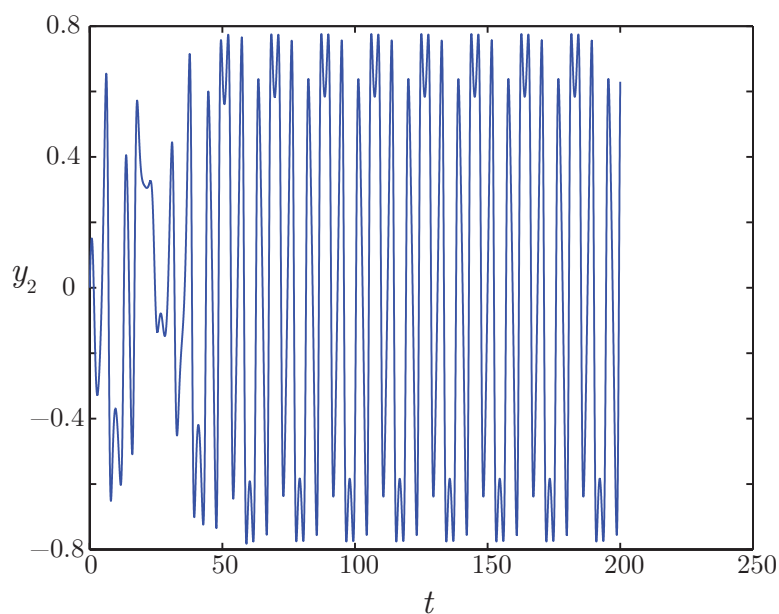


Figure 19.3: Plot of prediction of velocity versus time, $y_2(t)$, for the forced, damped Duffing equation for $t \in [0, 200]$; $\alpha = 1$, $\beta = -1$, $\delta = 0.22$, $f = 0.3$.

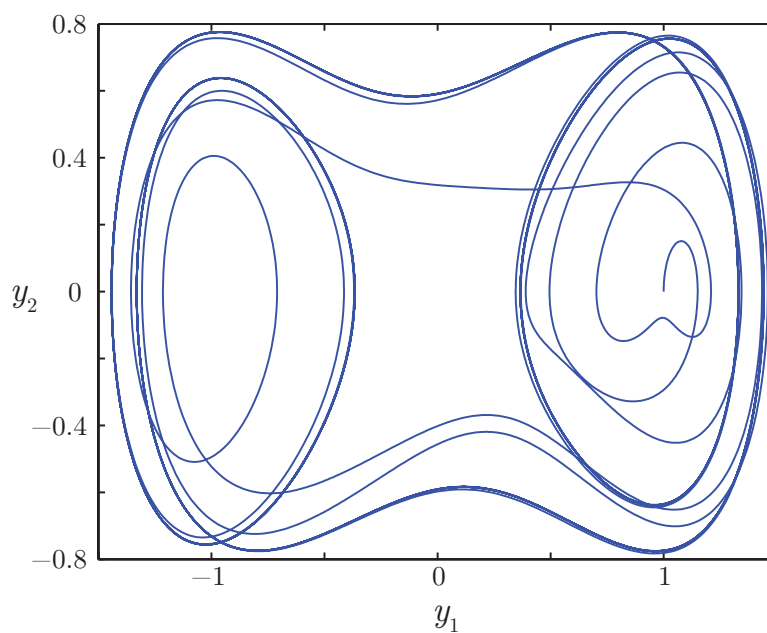


Figure 19.4: Plot of prediction of velocity versus position, $y_2(y_1)$, for the forced, damped Duffing equation, $t \in [0, 200]$; $\alpha = 1$, $\beta = -1$, $\delta = 0.22$, $f = 0.3$.

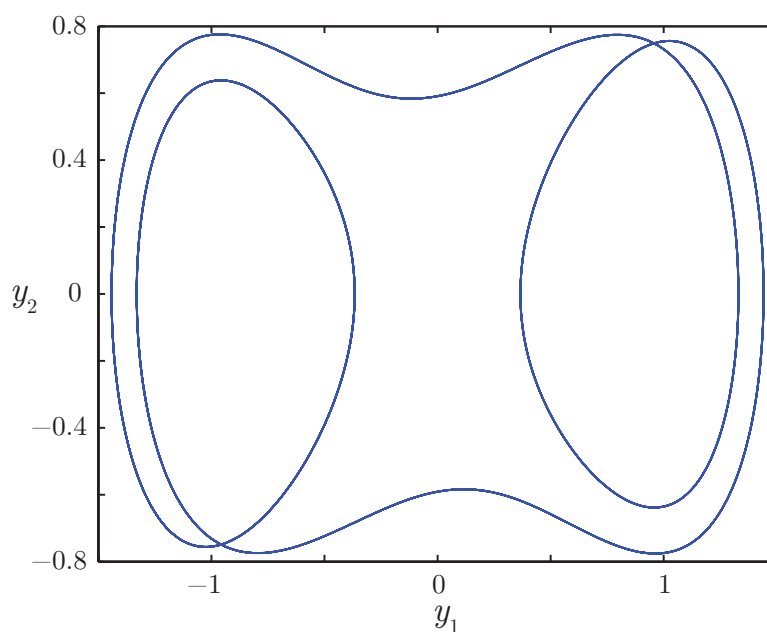


Figure 19.5: Plot of prediction of velocity versus position, $y_2(y_1)$, for the forced, damped Duffing equation, $t \in [100, 200]$; $\alpha = 1$, $\beta = -1$, $\delta = 0.22$, $f = 0.3$.

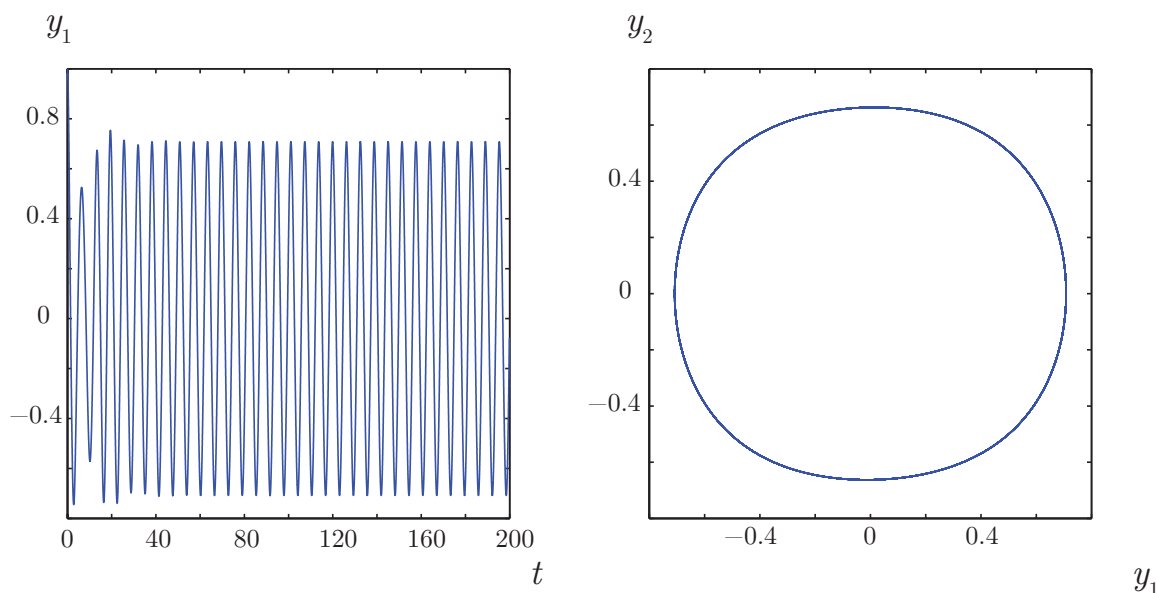


Figure 19.6: Plot of prediction of position versus time, $y_1(t)$, $t \in [0, 200]$ and velocity versus position, $y_2(y_1)$, $t \in [100, 200]$, for the forced, damped Duffing equation, $\alpha = 1$, $\beta = 1$, $\delta = 0.22$, $f = 0.3$.

Now the negative linear spring constant was highly unusual and might be difficult to actually build in nature. Let us return to a more physical situation where the linear spring constant is positive; we now take $\beta = 1$ and leave all other parameters the same. We give relevant plots in Fig. 19.6. It is noted the the phase plane plot has unfolded and now has a simpler ellipsoidal shape. It also appears there is a single frequency in play in the long time limit.

Let us make the problem simpler still by removing the effects of the non-linear spring by setting $\alpha = 0$. We give relevant plots in Fig. 19.7. It is noted the the phase plane plot has a circular shape. It also appears there is a single frequency in play in the long time limit. Even though there is damping, the amplitude of oscillation is $\mathcal{O}(1)$ in the long time limit. This is because we are forcing the system at a frequency near its resonant frequency of $\sim \sqrt{\beta} = 1$.

Let us change the natural frequency of the system by studying $\beta = 10$, thus inducing a resonant frequency of $\sim \sqrt{\beta} = \sqrt{10} \sim 3.16$. We are still forcing the system with a dimensionless frequency of unity. We give relevant plots in Fig. 19.8. It also appears there is a single frequency, that of the driver, in play in the long time limit. It is also noted that the phase plane plot retains a circular shape, but its amplitude is greatly reduced. This is because we are forcing the system at a frequency not near its resonant frequency.

Lastly, we can highlight the object-oriented nature of our approach by segregating our source code into separate files:

- `comdata.f90`,

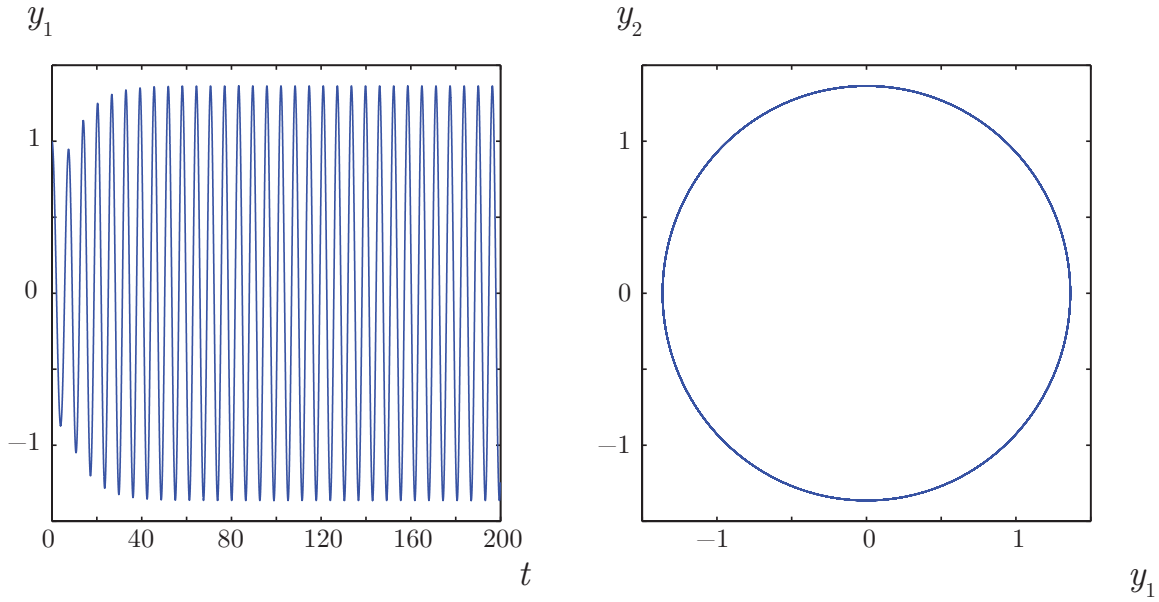


Figure 19.7: Plot of prediction of position versus time, $y_1(t)$, $t \in [0, 200]$ and velocity versus position, $y_2(y_1)$, $t \in [100, 200]$, for the forced, damped Duffing equation, $\alpha = 0$, $\beta = 1$, $\delta = 0.22$, $f = 0.3$.

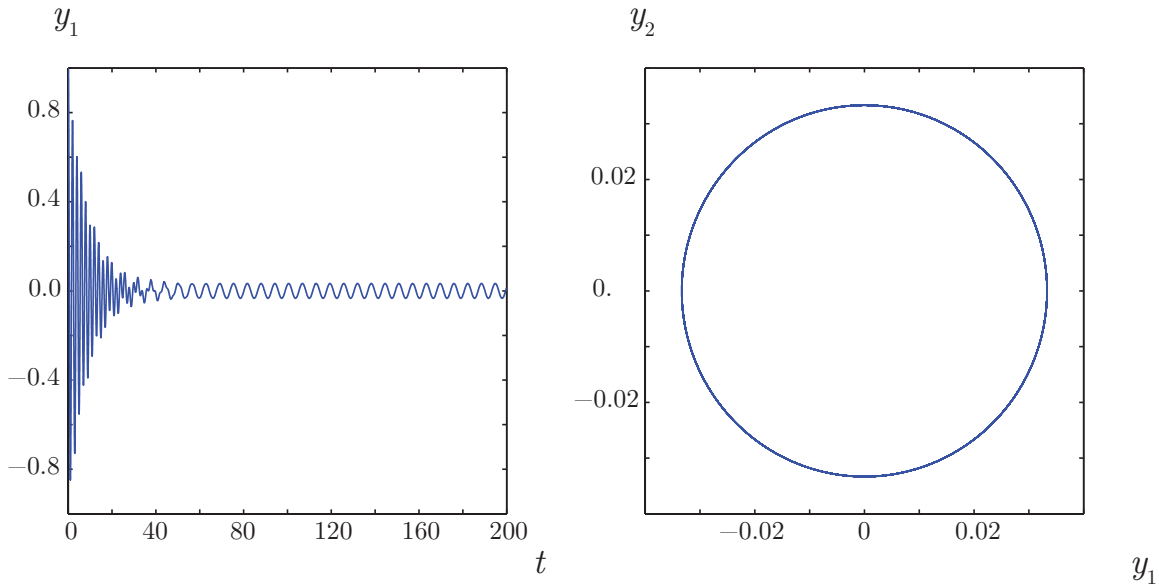


Figure 19.8: Plot of prediction of position versus time, $y_1(t)$, $t \in [0, 200]$ and velocity versus position, $y_2(y_1)$, $t \in [100, 200]$, for the forced, damped Duffing equation, $\alpha = 0$, $\beta = 10$, $\delta = 0.22$, $f = 0.3$.

- `rhs.f90`,
- `ode1.f90`
- `ode2.f90`, and
- `solveode.f90`.

Then we can build an executable file `runode`, to compile and execute our programs:

```
[powers@remote105 ame.20214]$ cat runode
ifort -c commondata.f90
ifort -c rhs.f90
ifort -c ode1.f90
ifort -c ode2.f90
ifort -c solveode.f90
ifort commondata.o rhs.o ode1.o ode2.o solveode.o
a.out
[powers@remote105 ame.20214]$
```

We omit the execution step because it generates lengthy output.

Chapter 20

Subroutines: 2

Read CES, Chapter 20.

The interested reader can consult the text of C&S.

Chapter 21

Modules

Read CES, Chapter 21.

We have already seen some use of the `module` feature. The interested reader can consult the text of C&S for more details.

Chapter 22

Converting from Fortran 77

Read CES, Chapter 36.

There is a large body of legacy **Fortran** code written under the **Fortran 77** standard. Code written under this standard often has noticeable differences from that written under the **Fortran 90**, **95**, **2003**, **2008** standards, along with many similarities. With a few exceptions, the user can expect

- Code which compiles and executes with a **Fortran 77** compiler will also compile and execute, generating the same output, under a **Fortran 90**, **95**, **2003**, **2008** standard.
- Code which was written under the enhanced standards of **Fortran 90**, **95**, **2003**, **2008** most likely will not be able to be compiled with a **Fortran 77** compiler.

As the **Fortran 77** standard has fewer features and thus is more streamlined, for certain applications, it will execute faster than programs written with the later standard, and for this reason, some users prefer it. However, for many modern scientific computing environments, especially those requiring dynamic memory allocation and parallelization, **Fortran 77** is simply unusable and thus not an option.

22.1 A hello world program

We give a “hello world” program for Fortran 77 in `helloworld.f`:

```
[powers@darrow2-p ame.20214]$ cat helloworld.f
    program test
    print*, 'Hello world.'
    end program test
[powers@darrow2-p ame.20214]$ f77 helloworld.f
[powers@darrow2-p ame.20214]$ a.out
Hello world.
[powers@darrow2-p ame.20214]$ ifort helloworld.f
```

```
[powers@darrows2-p ame.20214]$ a.out
Hello world.
[powers@darrows2-p ame.20214]$
```

Note:

- The file identifier for a Fortran 77 standard program is `.f`.
- *All Fortran 77 programs must have most text beginning in column 7 or greater.* Exceptions are the numbers for `format` statements and for a few other statements. This is a old feature motivated by the use of punchcards for early computer programs.
- The program compiles with the compiler `f77` and executed successfully.
- The program compiled with the modern compiler `ifort` and executed successfully.

22.2 A simple program using arrays

Here we revisit an earlier program. We recall the earlier presented `ch6c.f90`:

```
[powers@darrows2-p ame.20214]$ cat ch6c.f90
program temperature
implicit none
integer, parameter :: imax=7
integer (kind=4) :: i
real (kind=4), dimension(1:imax):: t,temp
real (kind=4), parameter :: a=270., b=10., c = 24., tmax = 24.
real (kind=4) :: pi, dt
pi = 4.*atan(1.)
print*, 'pi = ', pi
dt = tmax/(real(imax)-1.)
do i=1,imax
    t(i) = dt*(i-1)
    temp(i) = a - b*sin(2.*pi*t(i)/c)
    print*,i,t(i),temp(i)
end do
end program temperature
[powers@darrows2-p ame.20214]$ ifort ch6c.f90
[powers@darrows2-p ame.20214]$ a.out
pi =      3.141593
      1  0.0000000E+00    270.0000
      2   4.000000      261.3398
      3   8.000000      261.3398
      4  12.00000      270.0000
```

5	16.00000	278.6602
6	20.00000	278.6602
7	24.00000	270.0000

```
[powers@darrow2-p ame.20214]$
```

The Fortran 77 equivalent of this program, and its execution, is given in `ch22a.f`:

```
[powers@darrow2-p ame.20214]$ cat ch22a.f
```

```

program temperature
  implicit none
  integer*4 imax
  parameter (imax=7)
  integer*4 i
  real*4 t(1:imax),temp(1:imax)
  real*4 a,b,c,tmax
  parameter (a=270.,b=10.,c=24.,tmax=24.)
  real*4 pi, dt
  pi = 4.*atan(1.)
  print*, 'pi = ', pi
  dt = tmax/(real(imax)-1.)
  do i=1,imax
    t(i) = dt*(i-1)
    temp(i) = a - b*sin(2.*pi*t(i)/c)
    print*,i,t(i),temp(i)
  end do
end program temperature
```

```
[powers@darrow2-p ame.20214]$ f77 ch22a.f
```

```
[powers@darrow2-p ame.20214]$ a.out
```

```

pi =   3.14159274
1  0.  270.
2  4.  261.339752
3  8.  261.339752
4 12.  270.
5 16.  278.660248
6 20.  278.660248
7 24.  270.
```

```
[powers@darrow2-p ame.20214]$ ifort ch22a.f
```

```
[powers@darrow2-p ame.20214]$ a.out
```

```

pi =   3.141593
1  0.0000000E+00  270.0000
2  4.000000      261.3398
3  8.000000      261.3398
4 12.00000      270.0000
```

```

      5    16.00000      278.6602
      6    20.00000      278.6602
      7    24.00000      270.0000

```

```
[powers@darrow2-p ame.20214]$
```

We note the under the Fortran 77 standard

- Again, all lines start at column 7 or greater.
- We exercise the common option not to use `::` in the variable declaration sections. We could do this however.
- Variables must first be declared, then later assigned a value in a `parameter` statement.
- The `kind` statement is not used in Fortran 77. For (`kind=4`), we employ a `*4`. We also could have simply used `real` or `integer` as appropriate.
- We cannot use `dimension` to set array dimensions.
- The output to the screen has weaker formatting.

Here are a few other key differences.

- Fortran 77 often uses `double precision` for `kind=8`.
- Some Fortran 77 compilers require a double precision version of intrinsic functions, e.g. the double precision version of the exponential, `exp` is `dexp`. The double precision version of `sin` is `dsin`.
- Modules are not used in Fortran 77. To pass data between subroutines and the main program, `common` blocks are used. It is possible to use `common` blocks in Fortran 90, 95, 2003, 2008, but we will not focus on that here.
- Whole array functions such as `matmul` are not a part of Fortran 77.

22.3 A program using subroutines

Here we give the Fortran 77 version of an earlier program for solving generic systems of ordinary differential equations, `ch22c.f90`. The Fortran 77 version is found in `ch22b.f`. That program, its compilation, and execution are as follows:

```

[powers@remote105 ame.20214]$ cat ch22b.f
      program solveode
! This is the main driver program for solving
! a generic set of neq differential equations.
      implicit none

```

```

integer :: i,neq,nt
parameter (neq=1,nt=10)
real :: y(1:neq)
real :: t,tout,dt
real :: tstop,yic(1:neq)
parameter (tstop=1.)
data yic/1./                                !data statement for array

open(10,file = 'output.dat')
t = 0.                                       !initialize t
dt = tstop/real(nt)                         !determine dt
do i=1,neq
    y(i) = yic(i)                           !assign y to its initial condition
enddo
print*,t,y                                  !output results
write(10,*)t,y
do i=1,nt                                    !main loop in time
    tout = t + dt                           !set incremental end time
    call ode1(y,t,tout,neq)                 !update y and t
    print*,t,y                               !output the results
    write(10,*)t,y
enddo
print*, 'error = ',y(1) - exp(-t)
stop
end

!
!-----
!
! This subroutine evaluates the right hand side of the
! system of neq ordinary differential equations.
!

subroutine fcn(t,y,ydot,neq)
implicit none
integer neq
real :: t
real :: y(1:neq)
real :: ydot(1:neq)
real :: lambda
parameter (lambda=-1.)
ydot(1) = lambda*y(1)
return
end

```

```

!
!-----
!
! The subroutine is a generic algorithm for
! the first order explicit Euler method
! for solving the system of neq ordinary
! differential equations evaluated by
! subroutine fcn.
!
      subroutine ode1(y,t,tout,neq)
      implicit none
      integer :: i,neq
      real :: y(1:neq)
      real :: t
      real :: tout
      real :: ydot(1:neq)
      real :: dt
      dt = tout-t
      call fcn(t,y,ydot,neq)      !estimate ydot at t
      do i=1,neq
         y(i) = y(i)+dt*ydot(i)  !estimate y at t
      enddo
      t = tout                    !update t
      return
      end
!
!-----
!
! This subroutine is a generic algorithm for
! the second order Runge-Kutta method for
! solving the system of neq ordinary differential
! equations evaluated by subroutine fcn.
!
      subroutine ode2(y,t,tout,neq)
      implicit none
      integer :: i,neq
      real :: y(1:neq)
      real :: yi(1:neq),ydot(1:neq)
      real :: t
      real :: tout
      real :: dt
      dt = tout-t

```

```

      call fcn(t,y,ydot,neq)          !estimate ydot at t
      do i=1,neq
        yi(i) = y(i)+dt*ydot(i)/2.  !estimate y at t+dt/2
      enddo
      call fcn(t+dt/2.,yi,ydot,neq)  !estimate ydot at t+dt/2
      do i=1,neq
        y(i) = y(i) + dt*ydot(i)    !estimate y at t
      enddo
      t = tout                        !update t
      return
    end
!
!-----
!
[powers@remote105 ame.20214]$ f77 ch22b.f
[powers@remote105 ame.20214]$ a.out
 0.  1.
0.100000001  0.899999976
0.200000003  0.810000002
0.300000012  0.728999972
0.400000006  0.656099975
0.5  0.590489984
0.600000024  0.531440973
0.700000048  0.478296876
0.800000072  0.430467188
0.900000095  0.387420475
1.00000012  0.34867841
error = -0.0192009807
[powers@remote105 ame.20214]$

```

Note

- The module structure was not used.
- The main program appeared first.
- The `parameter` statement has a slightly different syntax.
- We used a `data` statement to initialize the array `yic`.
- We did not use a `commondata` structure for shared data. Sometimes this is useful, and one can employ a so-called `common` block, not done here, when appropriate.

- The number of equations, `neq`, was passed as a parameter to most subroutines because Fortran 77 is more restrictive with variable sized arrays.
- We could have separated our code into distinct codes for the main program and subroutines, compiled each under the `-c` option, and employed an executable script file.

Chapter 23

Python and Fortran

Python is a general open-source interpreted high-level programming language. It shares some functionality with MATLAB. It is possible to use Python to interact with Fortran routines. This section of the notes will give only the most rudimentary of introductions, focusing entirely on how Python can be implemented with Fortran on local Notre Dame machines.

At present software to allow Python to interact with Fortran is only loaded on the remote machines (e.g. `remote102.helios.nd.edu`).

Now let us consider a Fortran subroutine to give a six-term Taylor series approximation of e^x about $x = 0$. The exact Taylor series is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \quad (23.1)$$

The six-term approximation is

$$e^x \sim 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120}. \quad (23.2)$$

A function subroutine to accomplish this task is embodied in `ch23a.f90`:

```
[powers@remote102 ame.20214]$ cat ch23a.f90
real function expa(x)
implicit none
real, intent(in) :: x
expa = 1. + x + x**2/2. + x**3/6. + x**4/24. + x**5/120.
end function expa
[powers@remote102 ame.20214]$
```

The function takes an input `x` and returns an approximation of e^x in the term `expa`.

Now let us operate on `ch23a.f90` in such a fashion that it can be used by Python. This is achieved with the Fortran to Python interface generator `f2py`. The generation is achieved as follows:

```
[powers@remote102 ame.20214]$ f2py -c -m ch23a ch23a.f90 --fcompiler=gfortran
```

The generation relies on the freeware compiler `gfortran`. It generates a binary output file named `ch23a.so`. It also creates a large output to the screen, omitted here.

At this stage we can open `Python`, and use it to call upon our `Fortran` interface to approximate e^x . We will look at three approximations: e^0 , e^1 , and e^2 . The sequence of commands that achieves this is as follows:

```
[powers@remote102 ame.20214]$ python
Python 2.6.6 (r266:84292, May  1 2012, 13:52:17)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ch23a
>>> print ch23a.expa(0)
1.0
>>> print ch23a.expa(1)
2.71666693687
>>> print ch23a.expa(2)
7.26666688919
>>> exit()
[powers@remote102 ame.20214]$
```

Chapter 24

Introduction to C

The C language is a commonly used programming language for which we give brief examples here.

24.1 A hello world program

Here is a “hello world” program in C, `helloworld.c`

```
[powers@darrow1-p 2e]$ cat helloworld.c
#include <stdio.h>
int main()
{
    printf("hello world\n");
}
[powers@darrow1-p 2e]$ gcc helloworld.c
[powers@darrow1-p 2e]$ a.out
hello world
[powers@darrow1-p 2e]$
```

Note:

- The `.c` identifier is appropriate for a C program.
- The symbol `#` invokes a so-called “pre-processor”.
- The `#include` statement inserts the object file into the program at that line.
- The code needs to appeal to an external standard input output library, `stdio.h`.
- The main program is encased by brackets, `{` and `}`.
- The appendage `\n` instructs the program to begin a new line.
- The compiler is invoked by the command `gcc`.

24.2 A program using the Euler method

Let us now use C to solve one of our paradigm problems, $dy/dt = \lambda y$, with $y(0) = y_o$ via the Euler method. We take $\lambda = -1$ and $y_o = 1$. The program is found in `euler.c`. Its listing, compilation, and execution steps are given next.

```
[powers@darrow2-p ame.20214]$ cat euler.c
/*Name: euler.c
 *Description: Approximates  $dy/dt = \lambda y$ ;  $y(0) = y_0$ 
 *              with Euler's method. Exact solution is
 *               $y = y_0 \exp(\lambda t)$ 
 */
//Include I/O and math libraries
#include <stdio.h>
#include <math.h>
//Define problem parameters
#define tstop 1.0f
#define y0 1.0f
#define lambda -1.0f
#define dt 0.1f
int main ( void ){
    FILE *output;                                //output file syntax
    output = fopen("output.txt","w");             //open output file
    float y=y0, ynew, t = 0;                     //set initial time/value
    int n = 0;                                    //initialize counter
    printf("%.8f\t%.8f\t%.8f\n", t, y, y0*expf(lambda*t)); //print initial values
    fprintf(output, "%.8f\t%.8f\t%.8f\n", t, y, y0*expf(lambda*t)); //write initial values
    do{
        ynew = y+lambda*dt*y;                    //approximate new solution
        n = n+1;                                  //update counter
        t = dt*n;                                 //update time
        y = ynew;                                  //update y
        printf("%.8f\t%.8f\t%.8f\n", t, y, y0*expf(lambda*t)); //print results
        fprintf(output, "%.8f\t%.8f\t%.8f\n", t, y, y0*expf(lambda*t)); //write results
    } while(t < tstop);                          //loop end condition
    printf("error %.8e\n", y-y0*expf(lambda*t)); //print final error
    fclose(output);                              //close output file
    return 0;
}
[powers@darrow2-p ame.20214]$ gcc -lm euler.c
[powers@darrow2-p ame.20214]$ a.out
0.00000000 1.00000000 1.00000000
0.10000000 0.89999998 0.90483743
```

```

0.20000000 0.81000000 0.81873077
0.30000001 0.72899997 0.74081820
0.40000001 0.65609998 0.67032003
0.50000000 0.59048998 0.60653067
0.60000002 0.53144097 0.54881161
0.69999999 0.47829688 0.49658531
0.80000001 0.43046719 0.44932896
0.90000004 0.38742048 0.40656966
1.00000000 0.34867844 0.36787945
error -1.92010105e-02
[powers@darrow2-p ame.20214]$ cat output.txt
0.00000000 1.00000000 1.00000000
0.10000000 0.89999998 0.90483743
0.20000000 0.81000000 0.81873077
0.30000001 0.72899997 0.74081820
0.40000001 0.65609998 0.67032003
0.50000000 0.59048998 0.60653067
0.60000002 0.53144097 0.54881161
0.69999999 0.47829688 0.49658531
0.80000001 0.43046719 0.44932896
0.90000004 0.38742048 0.40656966
1.00000000 0.34867844 0.36787945
[powers@darrow2-p ame.20214]$

```

Note:

- The `.c` identifier is appropriate for a C program.
- A block of comment lines is bracketed by a pair of `/*`.
- The symbol `#` invokes a so-called “pre-processor”.
- The `#include` statement inserts the object file into the program at that line.
- The `#define` statement will substitute the second argument for the first in all occurrences of the first; it is equivalent to a Fortran `parameter` statement.
- An inline comment is initiated by the symbols `//`.
- The code needs to appeal to external libraries, a standard input output library, `stdio.h`, and a mathematics library, `math.h`.
- A floating point number is identified by a `f` at its trailing edge.
- Integers do not have a trailing `f`.

- Formatted printing to the screen is complicated.
- We have also opened, written to, and closed the file `output.txt`.
- The compiler is invoked by the command `gcc`.
- The compiler option `-lm` invokes the math library.

Chapter 25

Introduction to Microsoft Excel

The Microsoft Excel software is a common tool for some straightforward tasks in engineering computing. While it is most commonly used for business and finance tasks, with some effort, it can be used as a software tool to solve problems in scientific computing. It is generally not as efficient, both in programming ease as well as computational efficiency, as other tools that have been studied. But its near universal presence in the worldwide business environment gives it the advantage of being able to run on nearly every modern office computer. Occasionally, this advantage can outweigh its considerable disadvantages, especially for small-scale problems.

25.1 A hello world program

The “hello world” program in `helloworld.xlsx` is trivial. One simply enters “hello world” into a box. The actual display is shown in Fig. 25.1.

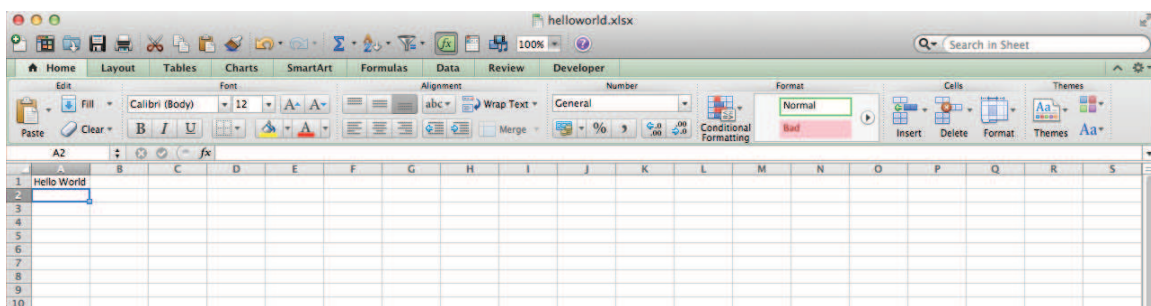


Figure 25.1: Microsoft Excel spreadsheet for “hello world.”

25.2 A program using the Euler method

We once again consider our model problem

$$\frac{dy}{dt} = -y, \quad y(0) = 1. \quad (25.1)$$

We use the Euler method with $\Delta t = 0.1$. The source code is in the file `euler.xlsx`. The formulæ are entered in the spreadsheet, displayed in Fig. 25.2. This is not the default display

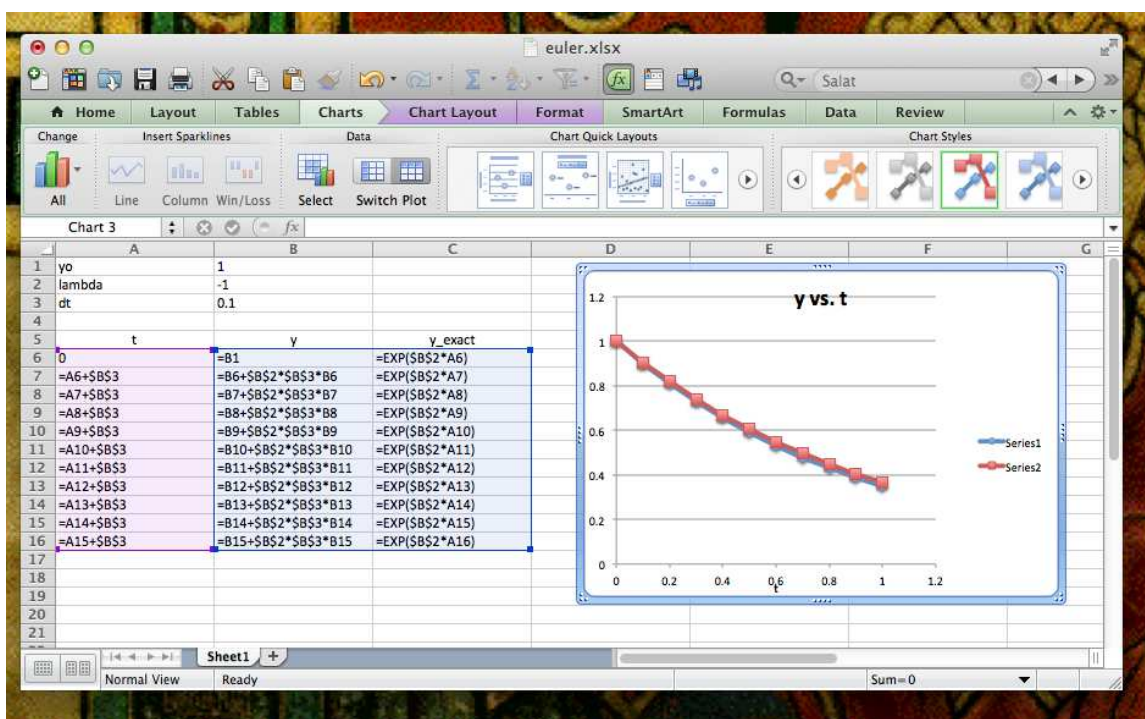


Figure 25.2: Microsoft Excel spreadsheet showing formulæ to solve $dy/dt = -y$, $y(0) = 1$ using the Euler method with $\Delta t = 0.1$.

format of Microsoft Excel, but can be realized by exercising some of the menu-driven options. The graph shown within Fig. 25.2 is a plot of $y(t)$ for both the Euler method and the exact solution. With considerable and non-obvious effort, the plot could be significantly improved. The rudimentary outline for forming this plot on the Macintosh version of Excel is as follows:

- Generate the appropriate numbers in columns A , B , and C with the appropriate formulæ.
- With the computer mouse, highlight and select all of the numbers within columns A , B , and C .

- From the **Charts** menu item, select **Scatter**, and from within **Scatter**, select **Smooth Marked Scatter**. This will generate a plot.
- From the **Charts** menu item, select **Chart Layout**, and from within this, select **Axis Titles**. This will give you options for entering text for axis titles.
- Explore many of the other options available for formatting the graphics, for example, under **Chart Layout**, one may define a **Chart Title**.

The actual display that is given by default upon entry of the formulæ is shown in Fig. 25.3.

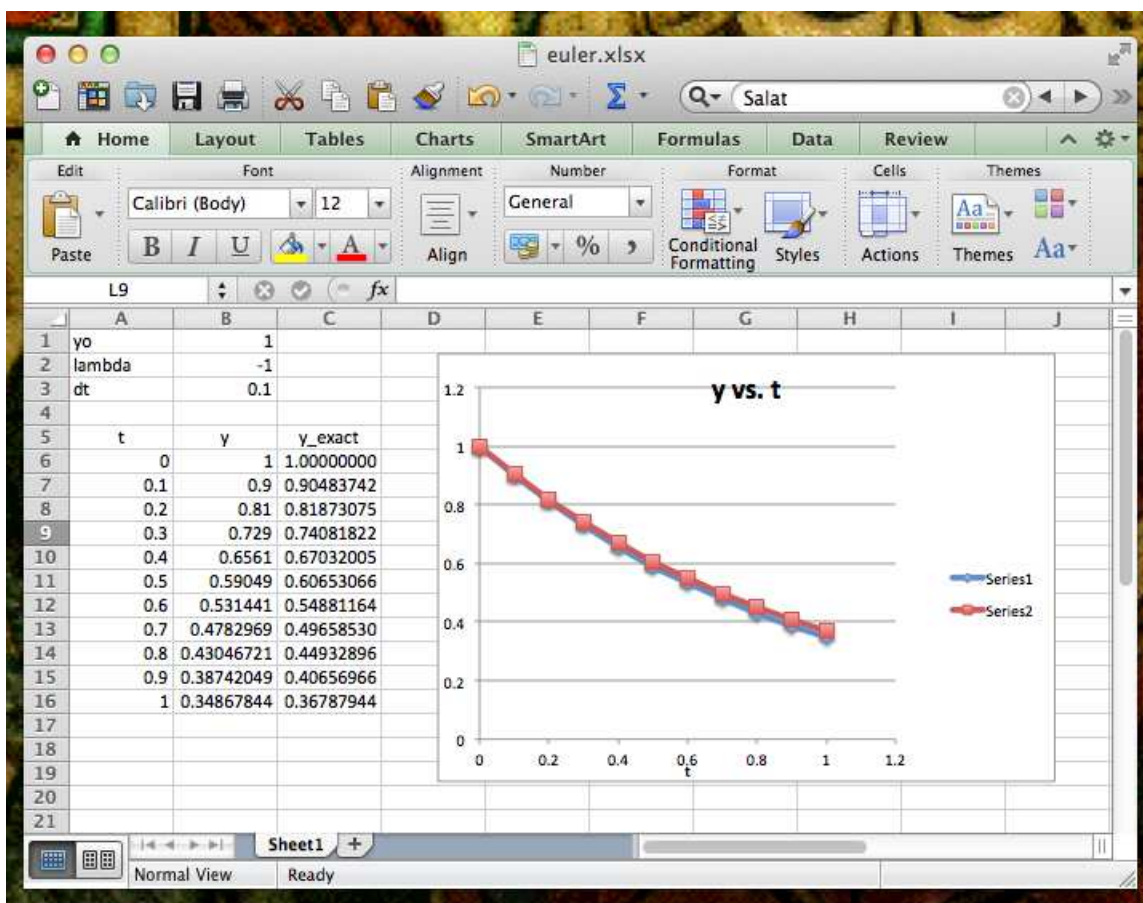


Figure 25.3: Microsoft Excel spreadsheet showing results of simulation of $dy/dt = -y$, $y(0) = 1$ using the Euler method with $\Delta t = 0.1$.

Chapter 26

Introduction to VBA

co-authored with Ms. Laura A. Paquin

The VBA language is a variant of the BASIC language specialized for execution within the spreadsheet application Microsoft Excel. It is in widespread use in many engineering environments.

Using a so-called “macro,” a program written or recorded within Microsoft Excel to automate a process, the user can link a code to a keyboard shortcut or a “button” in order to perform a task repeatedly. In order to enable macro-writing, one must first activate the Developer tab in Microsoft Excel. For Macintosh users, this can be achieved by clicking Preferences... in the Excel drop-down menu, then navigating to Ribbon and checking Developer in the list of tabs available. For Windows users, this can be achieved by going to Excel Options in the Office Button, navigating to Popular, and then checking Show Developer tab in the Ribbon. Additionally, security settings may have to be modified in order to allow the use of macros. These settings reside again in Excel Options within the Trust Center tab. From here, you can navigate into Trust Center Settings... and change the Macro Settings to enable all macros.

26.1 A hello world program

The “hello world” program in VBA_helloworld.xlsm calls upon a pop-up box to display a specified message. To build this from scratch, the user can first, in the Developer tab within Excel, place a button on a spreadsheet by selecting Button. Put the button in any location on your worksheet and then select New to begin writing the code in the Visual Basic editor. For this program, only three lines of code are required:

```
Sub Button1_Click()  
MsgBox ("hello world")  
End Sub
```

After closing the editor, one can activate the macro by clicking on the button in the spreadsheet. A depiction of the action of the “hello world” in the Excel environment is shown in Fig. 26.1.

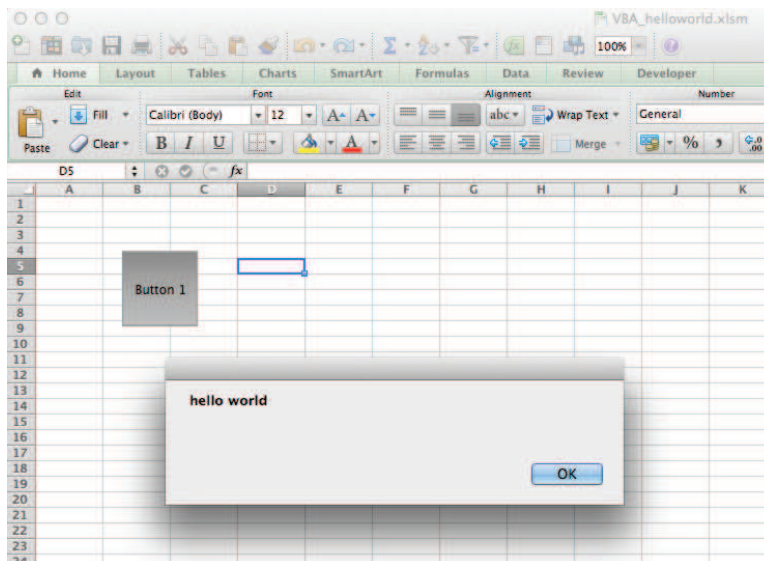


Figure 26.1: Macintosh screen shot of Excel spreadsheet after executing a simple “hello world” VBA program.

26.2 A program using the Euler method

We once again consider our model problem

$$\frac{dy}{dt} = -y, \quad y(0) = 1. \quad (26.1)$$

We use the Euler method with $\Delta t = 0.1$. The VBA source code for the simulation is in the file `VBA_Euler.xlsm`. The initial conditions and system parameters are entered in the Excel spreadsheet, displayed in Fig. 26.2. The initial value of $t = 0$ is in cell F2, and that of $y(0) = 1$ is in cell G2. The step size $\Delta t = 0.1$ is in cell H2, and the final value of $t = 1$ is in cell I2.

The macro code, embedded within the spreadsheet is given by

```
Sub Button1_Click()
Dim i As Integer 'iteration counter
Dim tstart As Double 'initial t value
Dim ystart As Double 'initial y value
Dim dt As Double 'incremental t value
Dim n As Double 'final iteration value
```

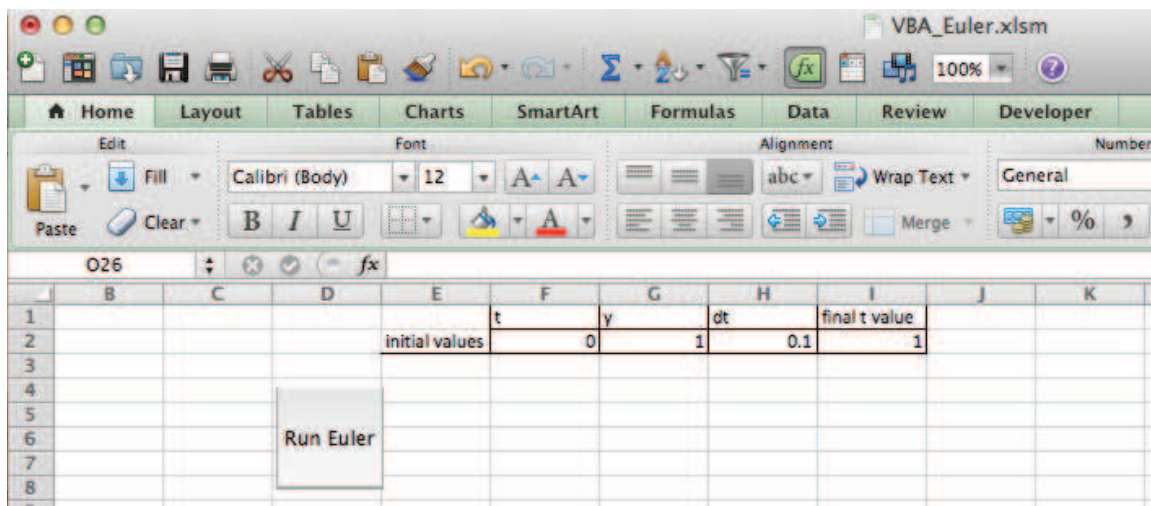


Figure 26.2: Microsoft Excel spreadsheet showing parameters to solve $dy/dt = -y$, $y(0) = 1$ with $\Delta t = 0.1$.

```

Dim tend As Double 'final t value--end of domain
Dim t As Double
Dim y As Double

'Assigning User-Input Parameters
tstart = Cells(2, 6)
ystart = Cells(2, 7)
dt = Cells(2, 8)
tend = Cells(2, 9)

'Initializing
t = tstart
y = ystart
n = Round(((tend - tstart) / dt), 0) 'rounded to 0 decimal places

'Running Euler
For i = 1 To n
    t = t + dt
    y = y + (-1 * y * dt)
    'Printing Values to Cells
    Cells((i + 2), 6) = t
    Cells((i + 2), 7) = y
Next i

```

End Sub

By inspection, one can infer that the program

- Declares variables of the appropriate type,
- Identifies input variables from the Excel spreadsheet and reads them,
- Executes a loop in which is embodied the Euler method,
- Has comments following the symbol '.

Following execution via clicking the “Run Euler” button, the spreadsheet has the display shown in Fig. 26.3. The results are identical to those shown in Fig. 25.3.

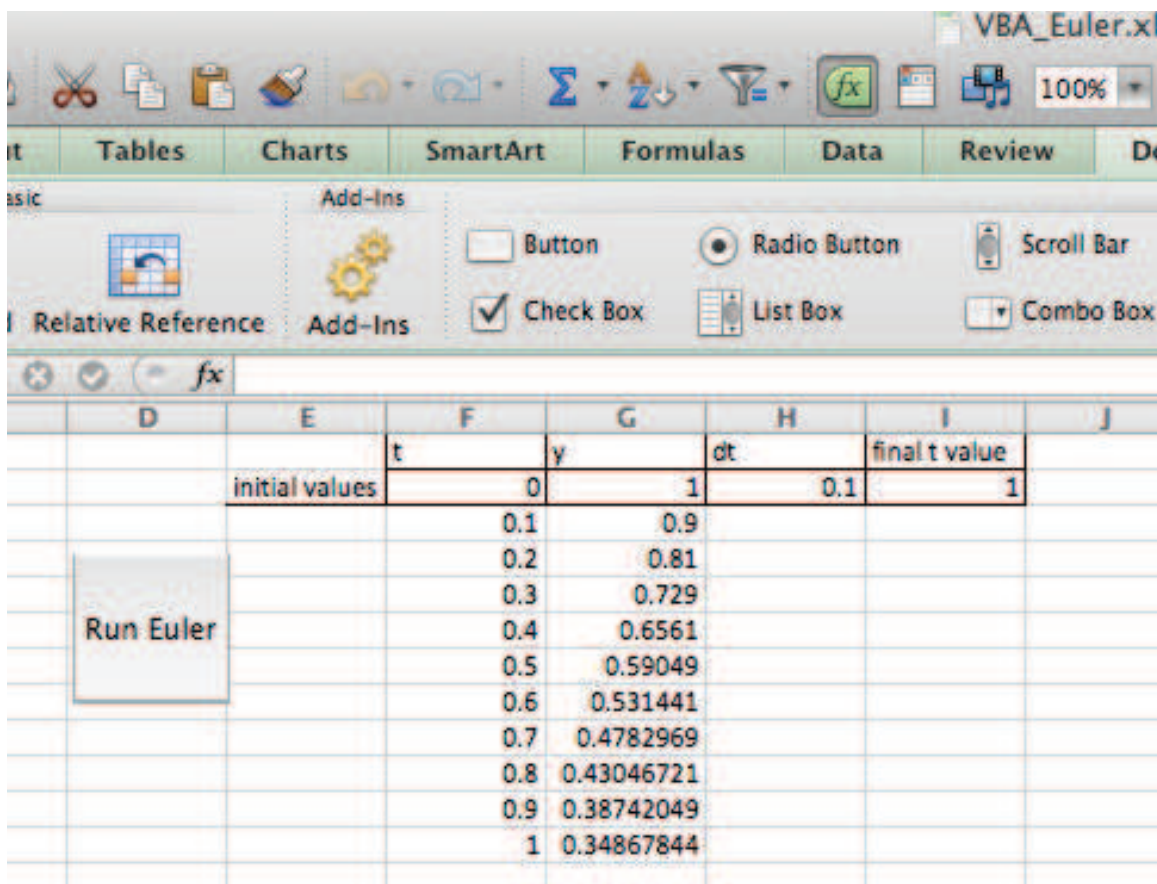


Figure 26.3: Microsoft Excel spreadsheet showing approximate solution via the Euler method of $dy/dt = -y$, $y(0) = 1$ with $\Delta t = 0.1$.

26.3 Another VBA Example

Programming with VBA in Microsoft Excel provides the user the opportunity to use the worksheet layout to organize input and output. This section will demonstrate how to tackle the second order differential equation,

$$m \frac{d^2 y}{dt^2} + b \frac{dy}{dt} + ky = F_o \sin(\nu t), \quad (26.2)$$

with the Euler method encoded in VBA. The user may establish parameter values within the code in the VBA workspace, or the program can call upon cells to initialize these values. For this example, the program will draw upon a row of populated cells to obtain parameters for the problem. This allows the user the ability to change parameters, like a damping coefficient or forcing frequency, on the worksheet, and rerun the macro without opening the code.

26.3.1 Creating a calculation macro

One may begin by creating a Form Control Button as shown in Fig. 26.4. The user can

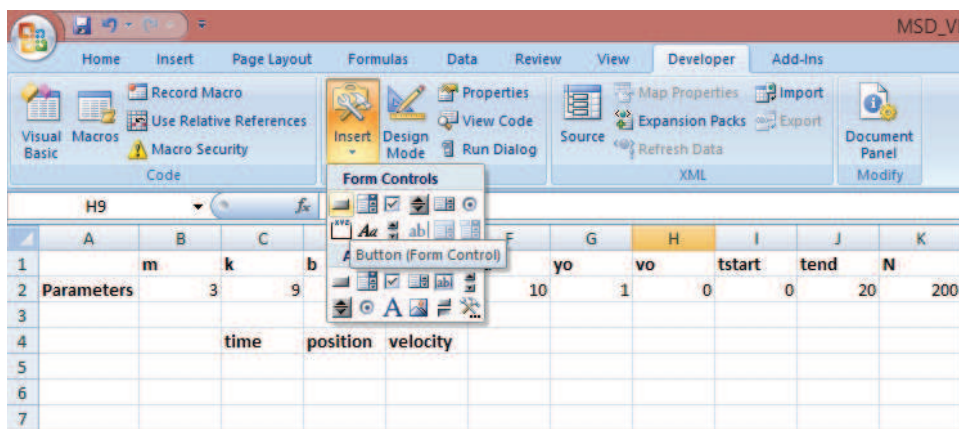


Figure 26.4: Creating a Form Control Button.

draw the button on the worksheet somewhere that will not interfere with the desired location of input or output. After naming the macro (in this case `MSD_solve`) as shown in Fig. 26.5, clicking **New** will bring the user into the Visual Basic workspace where he/she will code the program to be activated by the button.

Closing the workspace at any time will automatically save the script and return the user to the spreadsheet. To access the VBA workspace again, the user may navigate into the **Developer** tab and select **Visual Basic** as shown on the left of the screen in Fig. 26.6. The code for this macro can be seen below.

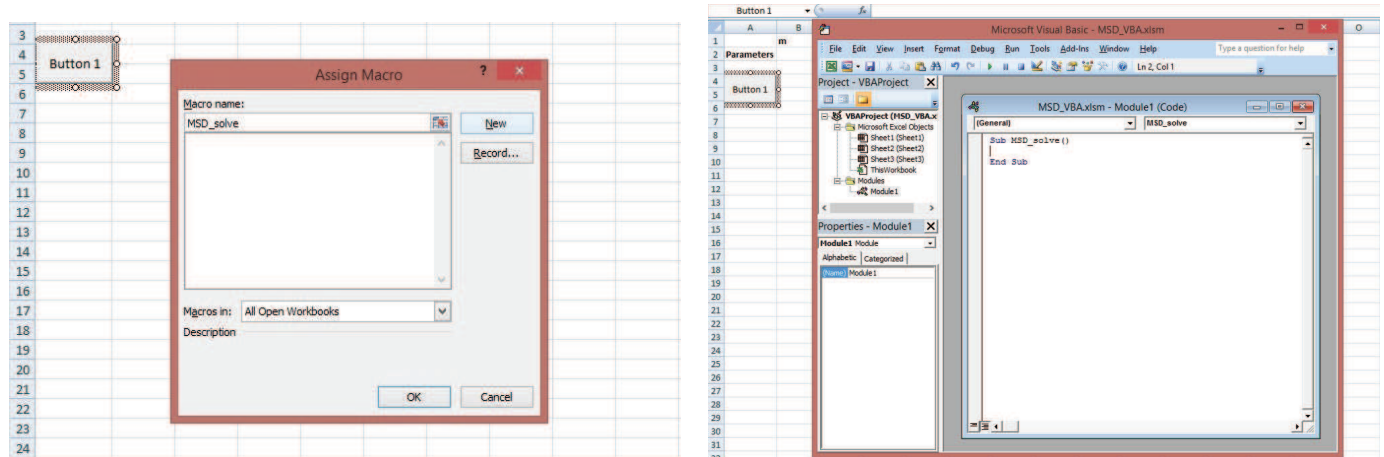


Figure 26.5: Creating macro and VBA workspace: a) Drawing button, naming macro, and beginning program, b) Visual Basic editor.

```
Sub MSD_solve()
Dim i, N As Integer
Dim m, k, b, Fo, nu, yo, vo, tstart, tend As Double 'Parameters
Dim y, v, t, ynew, vnew, tnew As Double 'Values to be calculated

'Initialize Variables
m = Cells(2, 2)
k = Cells(2, 3)
b = Cells(2, 4)
Fo = Cells(2, 5)
nu = Cells(2, 6)
yo = Cells(2, 7)
vo = Cells(2, 8)
tstart = Cells(2, 9)
tend = Cells(2, 10)
N = Cells(2, 11)

dt = (tend - tstart) / N

'At t=0
t = tstart
y = yo
v = vo
Cells(5, 3) = t 'print initial t
Cells(5, 4) = y 'print initial y
```

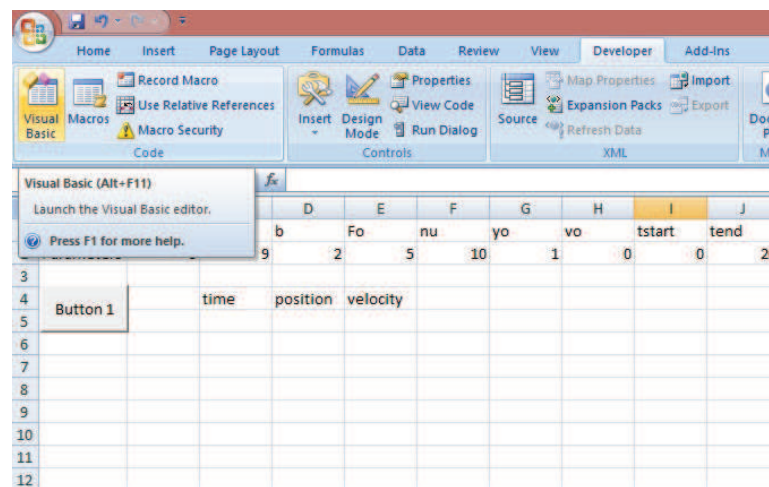


Figure 26.6: Navigating back into VBA workspace.

```
Cells(5, 5) = v 'print initial v

'Calculating solution
For i = 1 To N
    tnew = t + dt
    ynew = y + dt * v
    vnew = v + dt * (-(k / m) * y - (b / m) * v + (Fo / m) * Sin(nu * t))
    t = tnew
    y = ynew
    v = vnew
    'print updated t, y, v
    Cells(i + 5, 3) = t
    Cells(i + 5, 4) = y
    Cells(i + 5, 5) = v
Next i

End Sub
```

Note the following within the program:

- It dimensions every parameter and variable appropriately as a `Double` or an `Integer`.
- It establishes the value of these variables by calling on the contents of `Cells(i, j)`.
- It iterates through a do loop using the syntax `For i` and `Next i`.
- It prints the values from successive iterations to the cells in a column `j` using the format

`Cells(i + 5, j)`, and it offsets the row index by 5 in order to begin output beneath the title cells.

After exiting the editor, one may activate the program by clicking on the button. Results for $t \in (0, 20 \text{ s})$, $N = 20$, and the parameters listed in Table 26.1, are shown in Fig. 26.7.

Table 26.1: Mass-Spring-Damper System Parameters

m (kg)	k (N/m)	b (N s/m)	F_o (N)	ν (Hz)	y_o (m)	v_o (m/s)
3	9	2	10	2	1	0

	A	B	C	D	E	F	G	H	I	J	K
1		m	k	b	Fo	nu	yo	vo	tstart	tend	N
2	Parameters	3	9	2	10	2	1	0	0	20	20
3											
4	Button 1		time	position	velocity						
5			0	1	0						
6			1	1	-3						
7			2	-2	-0.96901						
8			3	-2.96901	3.154322						
9			4	0.185314	9.027081						
10			5	9.212395	5.750947						
11			6	14.96334	-27.5336						
12			7	-12.5703	-55.8565						
13			8	-68.4267	22.39399						
14			9	-46.0327	211.7852						
15			10	165.7525	206.19						
16			11	371.9425	-425.484						
17			12	-53.5417	-1257.68						
18			13	-1311.23	-261.622						
19			14	-1572.85	3849.015						
20			15	2276.166	6002.453						
21			16	8278.619	-4830.97						
22			17	3447.644	-26444.3						
23			18	-22996.7	-19156						
24			19	-42152.7	62601.48						
25			20	20448.83	147326.1						
26											

Figure 26.7: Output generated by macro.

26.3.2 Clearing data

26.3.2.1 Manual clearing

While the macro will overwrite $N + 1$ cells in columns D and E every time the button is pushed, it will not clear the cell contents in these columns before re-populating them. Thus, one can imagine why having a **Clear** macro would be needed to erase leftover data in cases when t_{end} or N is decreased between runs. One can clear a block of cells by selecting the top cell (in this case C5), pressing the the key combination **Ctrl + Shift + ↓** and then right clicking the selected cell block to choose **Clear Contents**, as shown in Fig. 26.8. To clear

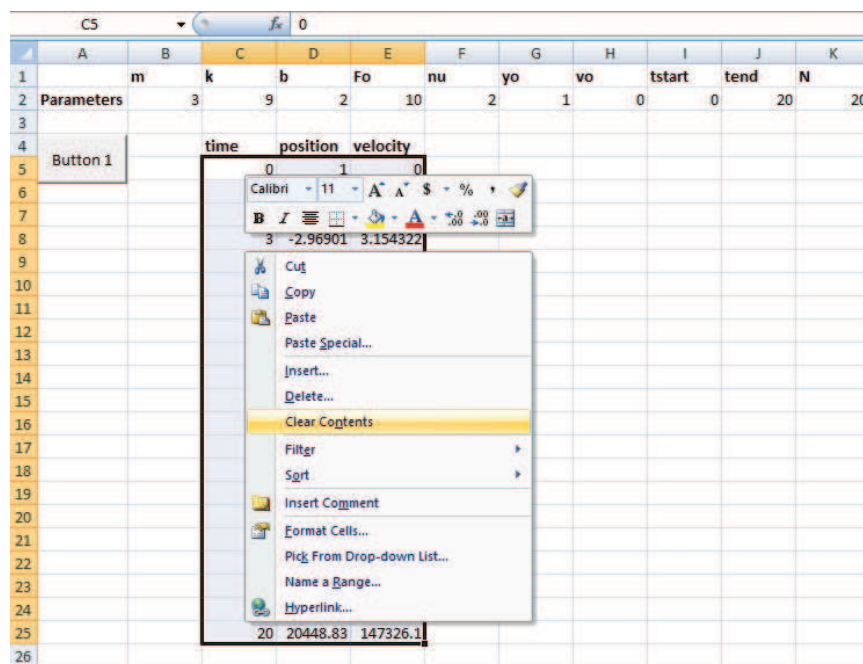


Figure 26.8: Manually clearing cell block contents.

a cell block spanning more than one column, one can use **Ctrl + Shift + ↓ + →** after selecting the top left cell.

26.3.2.2 Automated clearing

For this activity, the user will automate the clearing process by recording a macro which will clear the desired block of data upon every click of the button. After running the calculation macro, one can press **Record Macro** in the **Developer** tab. After naming the macro **Clear** and pressing **OK** as shown in Fig. 26.9, the macro will record all the user's following actions until he/she presses **Stop Recording**.

For this function, the user should again select the top left cell in the output block, hold **Ctrl + Shift + ↓ + →**, and then clear these cells. After pressing **Stop Recording** and then navigating to **Macros** within the **Developer** tab, one can view his/her encoded actions by selecting the new **Clear** macro and choosing **Step Into**. Upon viewing the script for the new **Clear** subroutine in the VBA workspace, one should see a code of the form below.

```
Sub Clear()
,
' Clear Macro
,
,
```

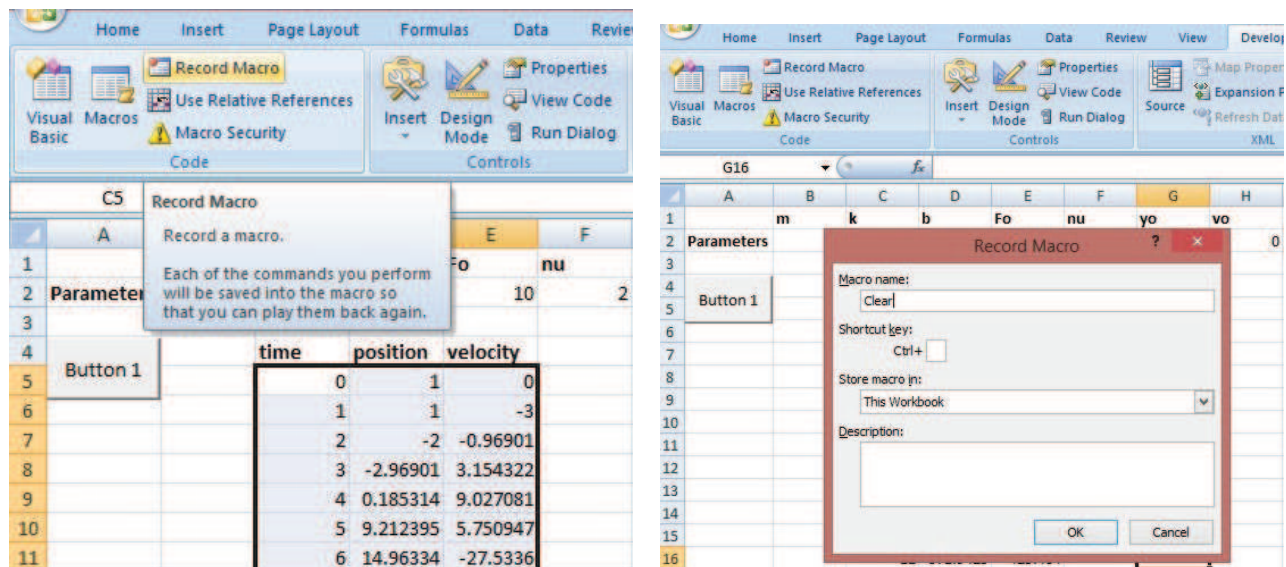


Figure 26.9: Recording a Clear Data Macro, a) Navigating to Record Macro, b) Naming macro.

```

Range("C5").Select
Range(Selection, Selection.End(xlDown)).Select
Range(Selection, Selection.End(xlToRight)).Select
Selection.ClearContents
End Sub

```

Brief inspection of this code will demonstrate the following:

- It selects the top left cell C5 pressed by the user.
- It extends the selection to the last cell of the populated cell block within the first column, denoted `Selection.End(xlDown)`.
- It extends the selection to the rightmost populated column with the populated cell block with `Selection.End(xlToRight)`.
- Once the entire cell block is included in the `Selection`, it depopulates it with `ClearContents`.

To embody this macro in a button, one can again choose a **Form Control** button in the **Developer** tab. After placing the button on the worksheet, the user will assign the **Clear** macro to the button as shown in Fig. 26.10. The two buttons now allow the user to perform as many runs as desired for the Euler approximation of the mass-spring-damper system with any specified time range and $N+1$ number of points.

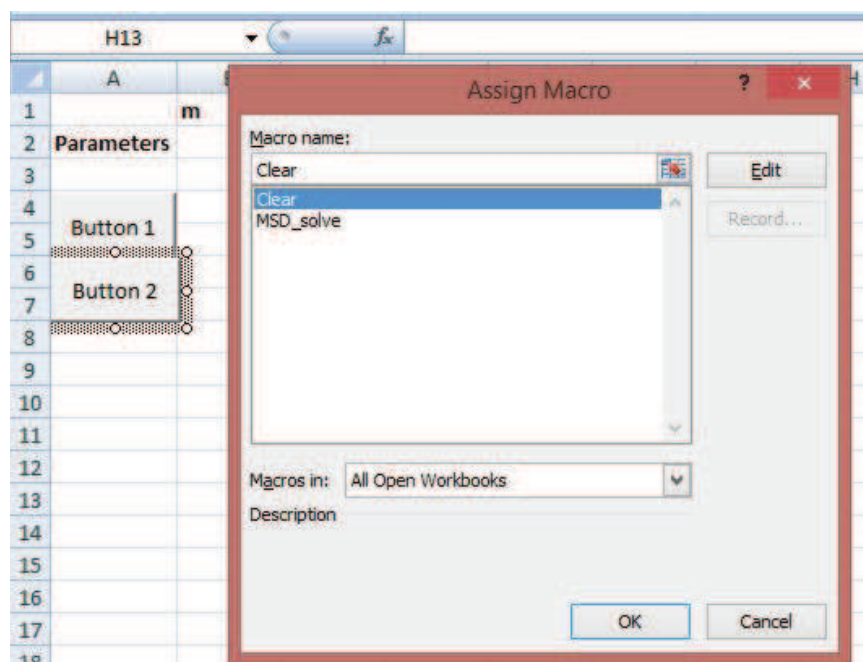


Figure 26.10: Assigning Clear macro to button.

26.3.3 Automated plotting

One may automate a visual representation of output by creating a plot linked to the output cell block. This example will demonstrate how the macro will automate plots of the position and velocity of the system.

The user can create the plot by selecting the title cell reading **time**, using **Ctrl + Shift + ↓** to select all time values in the column, holding **Ctrl** and selecting the **position** cell, and then using **Ctrl + Shift + ↓** again to select the entirety of the position data. Once this data is selected, choosing **Scatter** from the **Insert** tab will display a scatter plot of the position as a function of time.

Performing the same sequence and choosing the **velocity** title cell instead of the **position** one will generate an analogous plot of velocity as a function of time. Figure 26.11 demonstrates the results of this process.

These plots, synchronized with the cell blocks containing the time, position, and velocity data, will update each time a macro is activated. Because of the **Ctrl + Shift + ↓** command initially utilized to select the plot data, the scatter plots will update to include as many entries as are generated with each run. For example, if the user increases or decreases t_{end} or N and then reruns the calculation macro, the number of points plotted in the scatter plots will correspond to the new number of populated cells in the **time**, **position**, and **velocity** blocks.

One may embellish the generated plots in the **Chart Tools** ribbon.

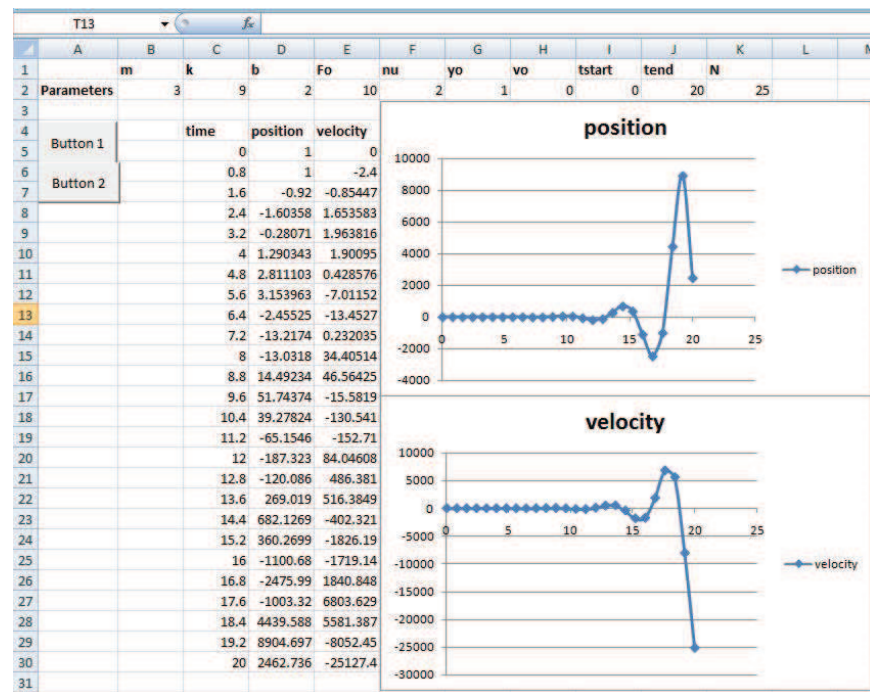


Figure 26.11: Automated scatter plots.

26.3.4 Miscellaneous

Other important items when programming in VBA:

- One can customize the button by right clicking on it, choosing **Edit Text** and/or **Format Control**.
- One should save the Excel file as a **.xlsm** (macro-enabled) instead of **.xlsx**.

Chapter 27

Introduction to Mathematica

Visit Wolfram's Mathematica reference guide.

The **Mathematica** software can be a useful tool for engineering computing. Among its advantages are

- It can do exact symbolic mathematics in terms of variables.
- It can do numerical calculations with arbitrary precision.
- It can render high quality graphics.
- It is available on campus cluster machines, and can be downloaded from the OIT website for use on personal computers, <http://oit.nd.edu>.
- It has a very particular syntax, forcing the user to employ precision in writing code.

Among its disadvantages are

- It has a very particular syntax, forcing the user to employ precision in writing code.

27.1 A hello world program

A particularly simple **Mathematica** version of a `hello world` code is given in `helloworld.nb`: It is trivially constructed by entering “hello world” and selecting the drop-down menu option `Convert to Text Cell`. A screen capture is given in Fig. 27.1.

27.2 A simple code

A particularly simple **Mathematica** code is given in `ch26a.nb`:

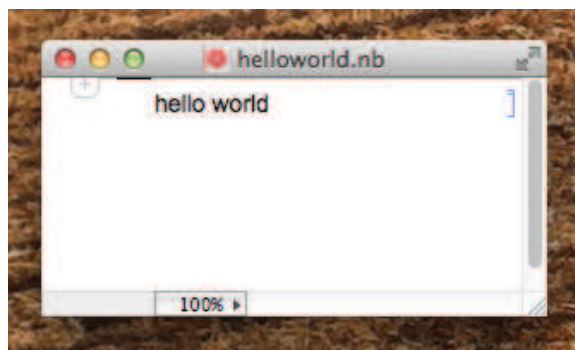


Figure 27.1: hello world in a Mathematica window.

```
In[1] := 1 + 1
```

```
Out[1] = 2
```

Note:

- The input is $1 + 1$.
- One employs “shift-return” from the keyboard and gets the output.
- Both input and output are integers.

If downloading this file with a web browser, you will likely find a long text file. You should be able to save this file as something with a `.nb` extension. Most web browsers will try to force you to do something else, but you should be able to override their default.

27.3 Precision

Mathematica has many ways to handle precision. Some of them are shown in `ch26b.nb`:

```
In[3] := 1/3
```

```
Out[3] = 1/3
```

```
In[4] := 1./3.
```

```
Out[4] = 0.333333
```

```
In[5] := 1/3 // N
```

```
Out[5] = 0.333333
```

[illegible]

- When rational numbers, i.e. ratios of integers, are entered, rational numbers are output.
- Finite precision numbers result when the decimal point is employed.
- The default number of digits output to the screen for a finite precision number is six.
- The command `// N` asks for a finite precision version of a number.
- **Mathematica**-defined functions are commenced with capital letters and encased in square brackets; the **Mathematica** function `N` asks for the finite precision version of a number.
- We can request as much precision as we desire. Twenty digits are requested for x by `N[x, 20]`; fifty digits are requested by `N[x, 50]`.

Mathematica has straightforward ways to deal with lists of numbers. One-dimensional lists can be thought of as vectors. Two-dimensional lists can be thought of as matrices. It is possible to have higher dimensional lists. The code `ch26c.nb` has several examples of how to deal with lists, including comments:

```
In[1]:= (* This is a comment *)

(* Assign the value 3 to variable x, "=" for assignment; then cube it *)
x = 3
x^3
```

```
Out[1]= 3
```

```
Out[2]= 27
```

```
In[3]:= (* Now assign x to be a vector, i.e. a list of numbers; {...} *)  
x = {2, 4, 5}  
(* Display x in matrix form *)  
x // MatrixForm
```

```
Out[3]= {2, 4, 5}
```

```
Out[4]//MatrixForm=
```

$$\begin{pmatrix} 2 \\ 4 \\ 5 \end{pmatrix}$$

```
In[5]:= (* Now assign A to be a matrix, i.e. a list of lists *)  
A = {{1, 3, 2}, {4, 5, 2}, {0, 9, 1}}  
(* Display A in matrix form *)  
A // MatrixForm
```

```
Out[5]= {{1, 3, 2}, {4, 5, 2}, {0, 9, 1}}
```

```
Out[6]//MatrixForm=
```

$$\begin{pmatrix} 1 & 3 & 2 \\ 4 & 5 & 2 \\ 0 & 9 & 1 \end{pmatrix}$$

```
In[7]:= (* Take the matrix-vector product b=A.x *)  
b = A.x
```

```
Out[7]= {24, 38, 41}
```

```
In[8]:= (* Solve for x, knowing A and b via x = Inverse[A].b *)  
x = Inverse[A].b
```

```
Out[8]= {2, 4, 5}
```

```
In[9]:= (* Solve for x if b = {1,2,3} *)  
b = {1, 2, 3}  
x = Inverse[A].b
```

```
Out[9]= {1, 2, 3}
```

```
Out[10]= {5/47, 16/47, -(3/47)}
```

```
In[11]:= (* Select isolated components of x *)
x[[1]]
x[[2]]
x[[3]]
```

```
Out[11]= 5/47
```

```
Out[12]= 16/47
```

```
Out[13]= -(3/47)
```

```
In[14]:= (* Select an isolated component of A *)
A[[1, 1]]
```

```
Out[14]= 1
```

```
In[15]:= (* Create some structured lists *)
x = Table[0, {i, 1, 5}]
x = Table[i, {i, 1, 5}]
x = Table[i^2, {i, 1, 5}]
A = Table[Table[i*j, {j, 1, 3}], {i, 1, 3}]
A // MatrixForm
```

```
Out[15]= {0, 0, 0, 0, 0}
```

```
Out[16]= {1, 2, 3, 4, 5}
```

```
Out[17]= {1, 4, 9, 16, 25}
```

```
Out[18]= {{1, 2, 3}, {2, 4, 6}, {3, 6, 9}}
```

```
Out[19]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

```
In[20]:= (* Give the LaTeX form of A *)
A // TeXForm
```

```

Out[20]//TeXForm=
\left(
\begin{array}{ccc}
1 & 2 & 3 \\
2 & 4 & 6 \\
3 & 6 & 9
\end{array}
\right)

```

27.5 Algebra

The code `ch26d.nb` gives some algebra operations:

```

(* This code will do some problems in algebra. *)
(* Expand a polynomial *)
Expand[(x + 1)^3]

```

```

Out[1]= 1 + 3 x + 3 x^2 + x^3

```

```

(* Factor a polynomial *)
Factor[x^2 + 4 x + 3]

```

```

Out[2]= (1 + x) (3 + x)

```

```

In[3]:= (* Solve an equation; "==" for equals *)
Solve[ x^2 + 5 x + 6 == 0, x]

```

```

Out[3]= {{x -> -3}, {x -> -2}}

```

```

(* Define an equation, then solve it; "=" for assignment *)
eq = x^2 + 5 x + 6 == 0
Solve[eq, x]
(* Result is a list of two rules; rules have -> *)

```

```

Out[4]= 6 + 5 x + x^2 == 0

```

```

Out[5]= {{x -> -3}, {x -> -2}}

```

```

In[6]:= (* Isolate the two solutions *)
(* /. denotes, "evaluate with" *)

```

```
{s1, s2} = Solve[eq, x]
```

```
x1 = x /. s1
```

```
x2 = x /. s2
```

```
Out[6]= {{x -> -3}, {x -> -2}}
```

```
Out[7]= -3
```

```
Out[8]= -2
```

```
In[9]:= (* Solve a general quadratic equation and give the result in Fortran form *)
```

```
{s1, s2} = Solve[a*x^2 + b*x + c == 0, x]
```

```
x1 = x /. s1
```

```
x2 = x /. s2
```

```
Out[9]= {{x -> (-b - Sqrt[b^2 - 4 a c])/(2 a)}, {x -> (-b + Sqrt[b^2 - 4 a c])/(2 a)}}
```

```
Out[10]= (-b - Sqrt[b^2 - 4 a c])/(2 a)
```

```
Out[11]= (-b + Sqrt[b^2 - 4 a c])/(2 a)
```

```
In[12]:= x1 // FortranForm
```

```
Out[12]= (-b - Sqrt(b**2 - 4*a*c))/(2.*a)
```

```
In[13]:= x2 // FortranForm
```

```
Out[13]=(-b + Sqrt(b**2 - 4*a*c))/(2.*a)
```

27.6 Some plots

Let us plot

$$y = 2x^2 + 1, \quad x \in [-2, 2].$$

This is achieved with

```
Plot[2 x^2 + 1, {x, -2, 2}, AxesLabel -> {"x", "y"},  
  LabelStyle->{FontFamily->"Times"}]
```

The plot is in Fig. 27.2a. Had we not specified any `FontFamily`, we would have obtained the default font.

We can make the fonts more readable and the axes labels italicized with the

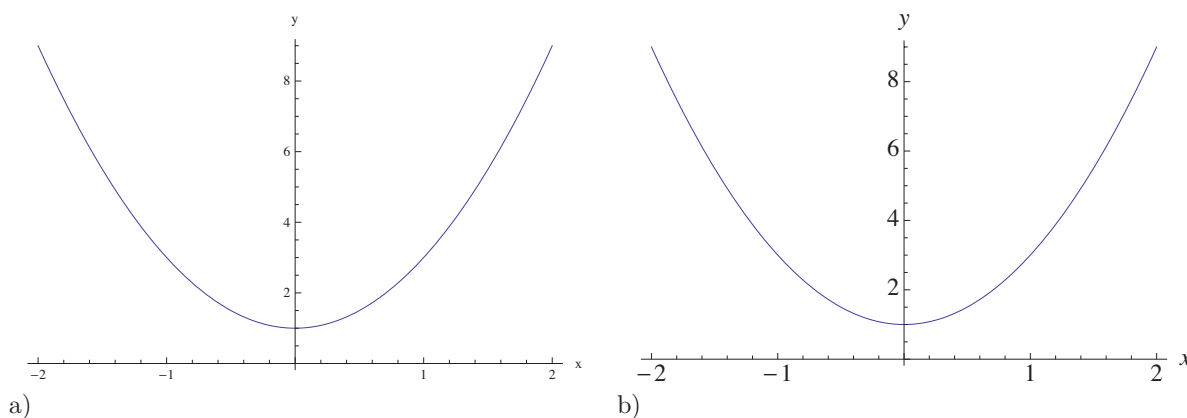


Figure 27.2: Plots of $y = 2x^2 + 1$: a) small fonts, b) large fonts.

```
Plot[2 x^2 + 1, {x, -2, 2}, AxesLabel -> {x, y},
     LabelStyle->{FontFamily->"Times"}, AxesStyle -> Directive[16]]
```

The plot is in Fig. 27.2b.

We can plot two curves by plotting a list containing the two functions to be plotted. Let us plot

$$y = 2x^2 + 1, \quad y = x^3, \quad x \in [-2, 2].$$

This is achieved by the command

```
Plot[{2 x^2 + 1, x^3}, {x, -2, 2}, AxesLabel -> {x, y},
     LabelStyle->{FontFamily->"Times"}, AxesStyle -> Directive[16]]
```

The plot is in Fig. 27.3.

We can render log-log plots. Consider the function

$$y = 2x^2 + 1, \quad x \in [0.001, 10],$$

in a log-log plot. This is achieved by the command

```
LogLogPlot[2 x^2 + 1, {x, .001, 10}, AxesLabel -> {x, y},
            LabelStyle->{FontFamily->"Times"}, AxesStyle -> Directive[16]]
```

The plot is in Fig. 27.4.

We can render plots of data points. Consider the data

$$(x, y) = (0, 0), (1, 1), (2, 2), (10, 10).$$

We can plot the data by the command

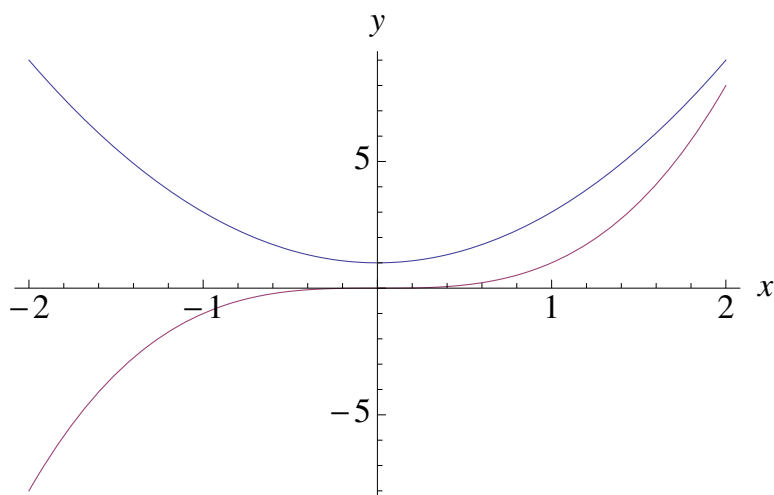


Figure 27.3: Plot of $y = 2x^2 + 1$ and $y = x^3$.

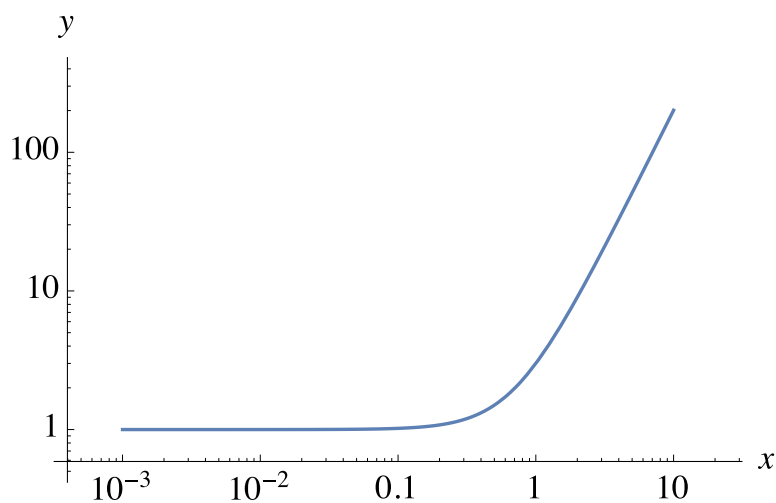


Figure 27.4: Log-log plot of $y = 2x^2 + 1$.

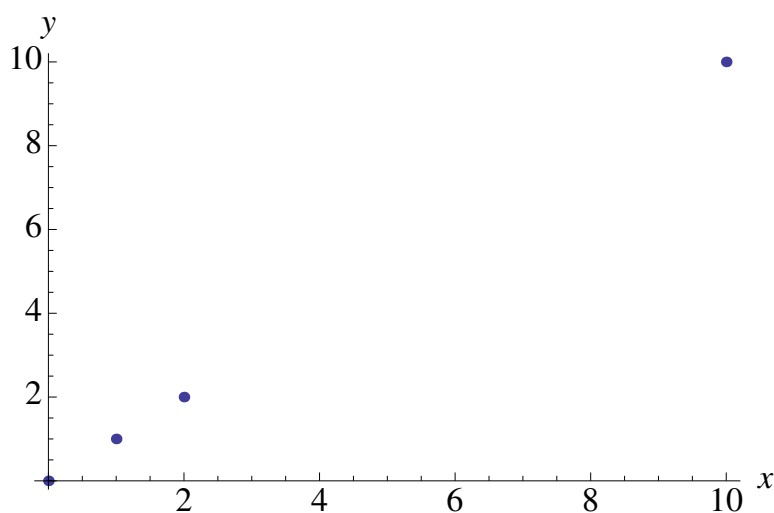


Figure 27.5: List plot of the data $(0,0)$, $(1,1)$, $(2,2)$, $(10,10)$.

```
data = {{0, 0}, {1, 1}, {2, 2}, {10, 10}}
ListPlot[data, AxesLabel -> {x, y}, LabelStyle -> {FontFamily -> "Times"},
  AxesStyle -> Directive[16], PlotStyle -> PointSize[0.015]]
```

The plot is in Fig. 27.5.

We can create two different types of plots and merge them via the following commands:

```
data = {{0, 0}, {1, 1}, {2, 2}, {10, 10}}
p1 = ListPlot[data, AxesLabel -> {x, y}, LabelStyle -> {FontFamily -> "Times"},
  AxesStyle -> Directive[16], PlotStyle -> PointSize[0.015]];
p2 = Plot[x, {x, 0, 10}];
Show[p1, p2]
```

The plot is in Fig. 27.6.

We can make three-dimensional surface plots. Consider the function

$$z = x^2 + y^2, \quad x \in [-4, 4], \quad y \in [-4, 4].$$

```
Plot3D[x^2 + y^2, {x, -4, 4}, {y, -4, 4}, AxesLabel -> {x, y, z},
  LabelStyle -> {FontFamily -> "Times"}, AxesStyle -> Directive[14]]
```

The plot is in Fig. 27.7.

We can save figures in a variety of formats, including `.eps`. This is achieved by using the mouse to select the figure, then going to the menu under **File**, and choose **Save Selection As**. These plots were generated directly from **Mathematica**. We could have also imported them into **Adobe Illustrator** for further processing.

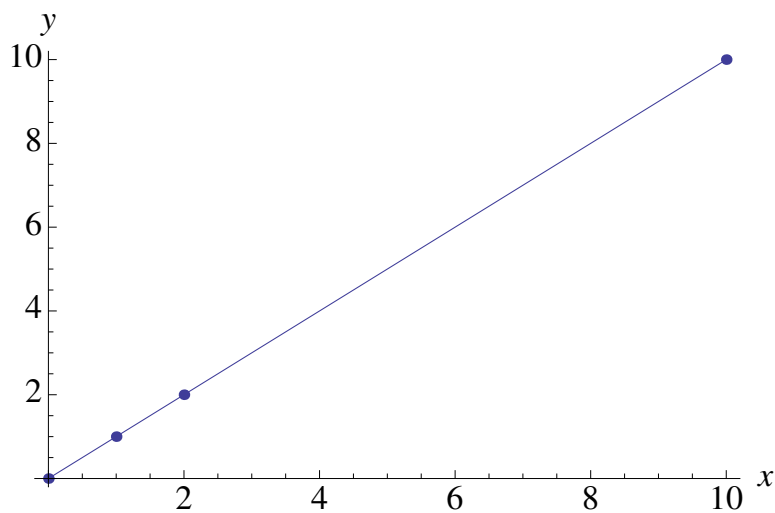


Figure 27.6: Plot of the data $(0, 0)$, $(1, 1)$, $(2, 2)$, $(10, 10)$ combined with a plot of $y = x$.

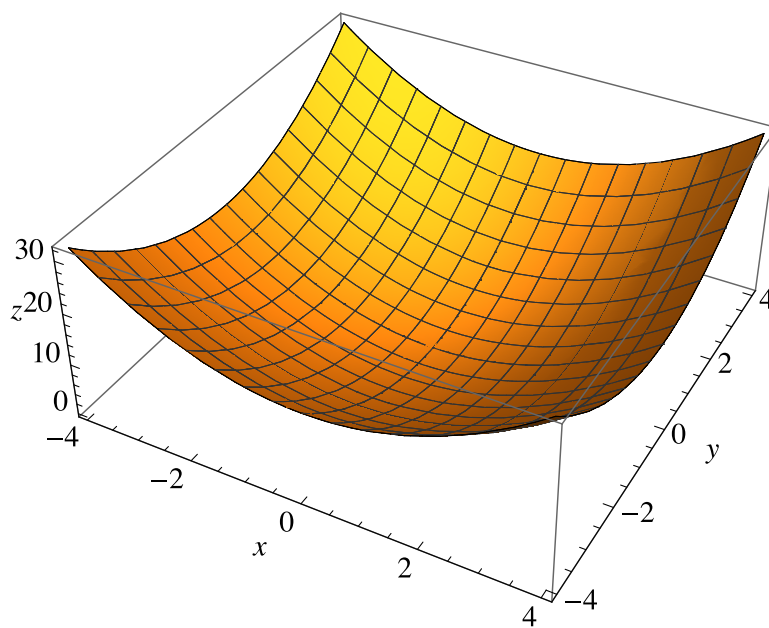


Figure 27.7: Plot of $z = x^2 + y^2$.

27.7 Calculus

A variety of operations from calculus are described in `ch26e.nb`:

```
In[65]:= y = 2 x^2 + 3
(* Indefinite integral *)
Integrate[y, x]
(* Definite integral *)
Integrate[y, {x, 0, 1}]

Out[65]= 3 + 2 x^2

Out[66]= 3 x + (2 x^3)/3

Out[67]= 11/3

In[68]:= (* First derivative *)
D[y, x]

Out[68]= 4 x

In[69]:= (* Second derivative *)
D[y, {x, 2}]

Out[69]= 4

In[121]:= (* Differential equation *)
Clear[y]
DSolve[{y''[x] + y'[x] + 4 y[x] == 0}, y[x], x]
{sol} = DSolve[{y''[x] + y'[x] + 4 y[x] == 0, y[0] == 1, y'[0] == 0},
  y[x], x]
yy = y[x] /. sol
Plot[yy, {x, 0, 10}, PlotRange -> All, AxesLabel -> {x, y},
  LabelStyle -> {FontFamily -> "Times"}, AxesStyle -> Directive[14]]

Out[122]= {{y[x] ->
  E^(-x/2) C[2] Cos[(Sqrt[15] x)/2] +
  E^(-x/2) C[1] Sin[(Sqrt[15] x)/2]}}
```

```
Out[123]= {{y[x] ->
  1/15 E^(-x/
  2) (15 Cos[(Sqrt[15] x)/2] + Sqrt[15] Sin[(Sqrt[15] x)/2])}}
```

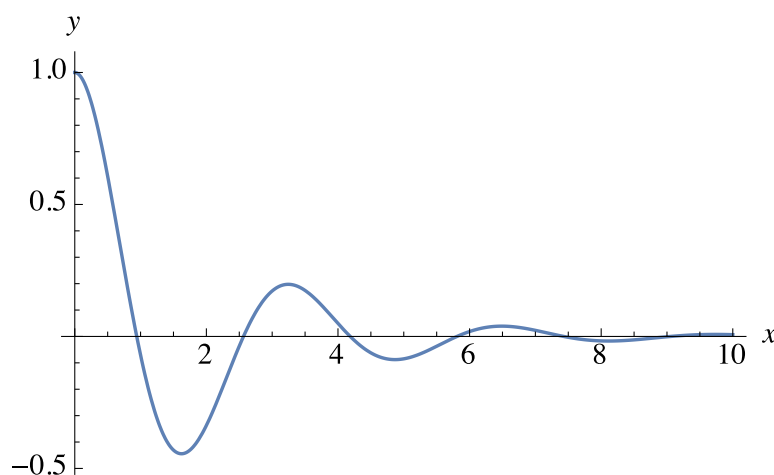


Figure 27.8: Plot of solution to $y'' + y' + 4y = 0$, $y(0) = 1$, $y'(0) = 0$.

```
Out[124]= 1/15 E^(-x/
2) (15 Cos[(Sqrt[15] x)/2] + Sqrt[15] Sin[(Sqrt[15] x)/2])
```

The plot is in Fig. 27.8.

27.8 The Euler method, DSolve, and NDSolve

We can solve differential equations with **Mathematica**. In `ch26f.nb`, we solve

$$\frac{dy}{dt} = \lambda y, \quad y(0) = 1, \quad \lambda = -1, \quad t \in [0, 1], \quad (27.1)$$

via three different methods:

- the exact solver **DSolve**,
- the Euler method, and
- the **Mathematica** numerical solver **NDSolve**.

Here is the code:

```
In[1]:= (* Solve dy/dt = -y, y(0) = 1 exactly *)
lambda = -1;
eqs = {y'[t] == lambda*y[t], y[0] == 1}
{sol} = DSolve[eqs, y[t], t]
yy = y[t] /. sol
p1 = Plot[yy, {t, 0, 1}, AxesLabel -> {t, y},
```

```

LabelStyle -> {FontFamily -> "Times"}, AxesStyle -> Directive[16]];

Out[2]= {Derivative[1][y][t] == -y[t], y[0] == 1}

Out[3]= {{y[t] -> E^-t}}

Out[4]= E^-t

In[6]:= (* Solve dy/dt = -y, y(0) = 1 with the Euler method *)

nt = 11;
p = 20
yyy = Table[0, {i, 1, nt}];
ttt = Table[0, {i, 1, nt}];
tstop = 1;
dt = N[tstop/(nt - 1), p];
yyy[[1]] = 1;
ttt[[1]] = 0;
Do[
{
  yyy[[i + 1]] = yyy[[i]] + dt*lambda*yyy[[i]],
  ttt[[i + 1]] = ttt[[i]] + dt
},
{i, 1, nt - 1}]
yt = Table[{ttt[[i]], yyy[[i]]}, {i, 1, nt}]
p2 = ListPlot[yt];
Show[p1, p2];

Out[7]= 20

Out[15]= {{0, 1}, {0.10000000000000000000,
  0.90000000000000000000}, {0.20000000000000000000,
  0.81000000000000000000}, {0.30000000000000000000,
  0.72900000000000000000}, {0.40000000000000000000,
  0.65610000000000000000}, {0.50000000000000000000,
  0.59049000000000000000}, {0.60000000000000000000,
  0.53144100000000000000}, {0.70000000000000000000,
  0.47829690000000000000}, {0.80000000000000000000,
  0.43046721000000000000}, {0.90000000000000000000,
  0.38742048900000000000}, {1.00000000000000000000,
  0.34867844010000000000}}

```

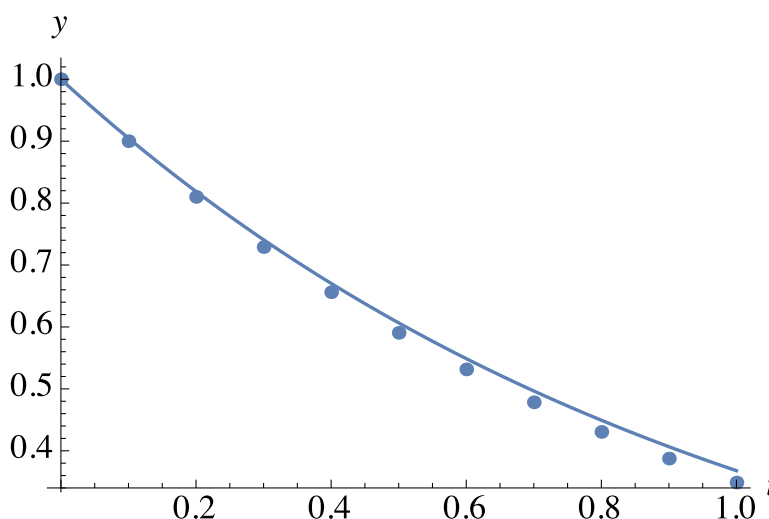


Figure 27.9: Plot of exact and Euler solution to $dy/dt = -y$, $y(0) = 1$.

```
In[18]:= (* Solve dy/dt = -y, y(0) = 1 with Mathematica's numerical solver *)
lambda = -1;
p = 20;
eqs = {y'[t] == lambda*y[t], y[0] == 1}
{sol} = NDSolve[eqs, y[t], {t, 0, 1}, WorkingPrecision -> p]
yy = y[t] /. sol
Plot[yy, {t, 0, 1}, AxesLabel -> {t, y},
  LabelStyle -> {FontFamily -> "Times"}, AxesStyle -> Directive[16]];

Out[20]= {Derivative[1][y][t] == -y[t], y[0] == 1}

Out[21]= {{y[t] -> InterpolatingFunction[{{0,1.000000000000000000}},<>][t]}}

Out[22]= InterpolatingFunction[{{0,1.000000000000000000}},<>][t]
```

The plot of the exact and Euler solution is in Fig. 27.9. Some miscellaneous features include

- Use a semicolon, “;” to suppress output to the screen.
- Greek letters can be formatted by use of the `esc` key. For example for the symbol ϕ , one enters `esc f esc`.
- `C` is reserved for use by Mathematica. So is `N`.
- Online help for commands is available in many ways. For example for more information on `NDSolve`, enter `?NDSolve`.

- Occasionally, you will need to “reboot” **Mathematica**. This can happen when its cache gets cluttered with old variables and you would like to refresh them. One can achieve this by quitting the kernel. To do so, go to the **Evaluation** menu and select **Quit Kernel**.

Bibliography

S. J. Chapman, 2018, *Fortran for Scientists and Engineers*, Fourth Edition, McGraw-Hill, New York.

This useful text is written for first year students in engineering.

I. Chivers and J. Sleightholme, 2015, *Introduction to Programming with Fortran with Coverage of Fortran 90, 95, 2003, 2008 and 77*, Third Edition, Springer, London.

This text gives an excellent introduction to the neophyte as well as useful information to the expert. Practical details as well as underlying programming philosophy are included. Moreover, the authors recognize that scientific computing is often the purpose of programming in **Fortran**, and include useful examples directed at mathematical science problems which extend beyond programming.

N. S. Clerman and W. Spector, 2012, *Modern Fortran: Style and Usage*, Cambridge, New York.

This is a modern text which does a good job explaining the usage of the **Fortran** language.

M. T. Heath, 2002, *Scientific Computing: An Introductory Survey*, Second Edition, McGraw-Hill, New York.

This advanced undergraduate text focuses on algorithms for solving important mathematical problems in science.

H. P. Langtangen, 2016, *Python Scripting for Computational Science*, Fifth Edition, Springer, Berlin.

This text focuses on how to implement the **Python** scripting language for problems in scientific computing. It includes a discussion of how to interface with lower level languages such as **Fortran** or **C**.

A. Markus, 2012, *Modern Fortran in Practice*, Cambridge, New York.

This is an introductory text for learning the **Fortran** language.

W. E. Mayo and M. Cwiakala, 1995, *Programming with Fortran 77*, Schaum's Outlines, McGraw-Hill, New York.

This is a readable guide which is very useful for learning programming in **Fortran 77**. While it does not have all of the modern upgrades to the language, it is nevertheless a useful source, especially with regard to legacy code.

- W. E. Mayo and M. Cwiakala, 1995, *Programming with Fortran 90*, Schaum's Outlines, McGraw-Hill, New York.

This is a readable guide which is very useful for learning programming in **Fortran 90**.

- M. Metcalf, J. Reid, and M. Cohen, 2011, *Modern Fortran Explained*, Oxford, Oxford.

This is a fine textbook which introduces the **Fortran** language.

- W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, 1992, *Numerical Recipes in Fortran 77: The Art of Scientific Computing*, Second Edition, Cambridge, New York.

This book is a *tour de force* of practical programming combined with applied mathematics to build and execute a wide variety of useful algorithms to solve problems in scientific computing. It is highly recommended.

- W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, 2007, *Numerical Recipes: The Art of Scientific Computing*, Third Edition, Cambridge, New York.

This update to the second edition uses the **C++** language.

- G. Strang, 2007, *Computational Science and Engineering*, Wellesley-Cambridge, Wellesley, MA.

This readable text presents material usually considered to be at the graduate level. That said, an interested undergraduate can learn much from it, as the author has a style which lends itself to students who are motivated by challenging material.

Index

- 32-bit architecture, 41, 49
- 64-bit architecture, 21, 41, 49, 50
-
- abs, 108, 130
- acos, 108
- acosh, 108
- Ada, 10
- Adobe Illustrator, 28, 61, 204
- AFS, 15, 32
- aimag, 129
- ALGOL, 10
- algorithm, 12, 35, 66, 83, 146, 150
- Andrew File System, 15
- arithmetic, 45
- ASCII, 22, 27, 41, 52, 96
- asin, 108, 132
- asinh, 108
- assembly language, 22
- atan, 57, 108
- atanh, 108
-
- BASIC, 10, 183
- binary executable, 12, 15, 103, 116, 117
- binary representation of integers, 48
- bit, 21, 41, 49
- byte, 21
-
- C, 10, 13, 15, 175
- C++, 10, 109, 110
- C#, 10
- cat, 31
- cd, 30
- Central Processing Unit (CPU), 22
- character, 39, 42
- cloud, 15
- cmplx, 108
-
- COBOL, 10
- comment statement, 40
- compiler, 14, 15, 18, 22, 40, 41, 82, 97, 117, 129, 132, 165, 166, 174, 175, 178
 - error, 41
 - option, 18
 - warning, 41
- complex, 129
- complex, 41
- complex conjugate, 130
- complex vectors, 75
- conjg, 130
- contains, 111
- cos, 108
- cosh, 108
- cp, 31
- Cygwin, 18, 19
-
- derived types, 137
- do, 39
- do while, 119, 123, 125
- DOS, 11
- dot_product, 73, 74
- Duffing equation, 150, 154
- dummy variable, 112, 142
-
- Emacs, 19, 28
- Euler method, 123, 125, 145, 150, 176, 180, 184, 207
- executable script file, 117
- exp, 108
-
- format, 41
- formatted, 99
- Fortran, 10, 12–15, 18–22, 27, 32, 39–41, 43, 45–47, 51, 56, 57, 63, 65, 67, 69, 73–

75, 78, 80, 81, 83, 84, 86, 87, 99, 107,
 109, 110, 112, 119–121, 124, 127, 129–
 132, 165
Fortran 2003, 11, 14, 18, 165
Fortran 2008, 11, 132, 165
Fortran 77, 165–168
Fortran 90, 11, 14, 18, 165
Fortran 95, 11, 165
 function subroutine, 107, 111–113, 115, 116

gedit, 19, 28, 52
gfortran, 18, 174

hello world, 14, 165, 175, 179, 183, 195
HTML, 28, 30

if, 39
ifort, 18, 22, 41, 49, 51, 97, 132, 166
imag, 129
implicit none, 40, 43
 input output, I/O, 39, 41, 43
int, 108
integer, 39, 41
 integer division, 46
intent, 111, 142
 interpreted language, 21

Java, 10, 109, 110

Kate, 19
kind, 41

L^AT_EX, 27, 200
less, 31
Linux, 11, 13, 15–19, 27, 28
load, 61
 local variable, 142
log, 108
log10, 108
logical, 42
ls, 30

Make, 118
 mass-spring-damper system, 150
Mathematica, 11, 195

MATLAB, 11, 20–22, 60, 61, 73, 80, 173
matmul, 73, 78, 80, 168
 matrix addition, 84
 matrix-matrix multiplication, 80
 matrix-vector multiplication, 78
Microsoft Excel, 11, 22, 179, 183
Microsoft Windows, 11
mkdir, 31
module, 109–111, 113, 150, 163
more, 31
mv, 31

NaN, Not a Number, 108
netfile, 15
 non-linear ordinary differential equations, 145,
 150

 object-oriented programming, 35, 109
open, 96
 operations, 45
 ordinary differential equations, 10, 123, 145
OS-X, 11, 13
 output, 89
 overflow, 49, 93

Pascal, 10
 pointers, 139
 pre-defined functions, 107
print, 39
pwd, 30
Python, 173

 quadratic equation, 119, 143, 201

 Random Access Memory (RAM), 22
read, 39, 99
real, 39, 41, 108, 129
rm, 31
rmdir, 31
 Runge-Kutta method, 146, 150

 secure shell, 15
sin, 108
sinh, 108

- software, 22
- `sqrt`, 108
- `ssh`, 15
- subroutine, 35, 141–143, 150, 161, 172, 173
 - function, 107, 111–113, 115, 116
- `tan`, 108
- `tanh`, 108
- Taylor series, 173
- Terminal, 15
- terminal, 17, 19, 22, 31
- TeXnicCenter, 27, 28
- TeXShop, 27, 28
- text editor, 19, 22
- transpose, 76
- `transpose`, 73
- undefined variables, 51
- underflow, 49
- unformatted, 41, 99
- Linux, 13
- UNIX, 11, 19, 20, 30–32, 117
- user-defined functions, 109
- VBA, 10, 183
- `vi`, 19, 28
- Windows, 13, 17, 19, 27, 28, 31, 183
- `write`, 39, 52, 60
- WYSIWYG, 13, 30, 31
- X Windows, 17
- X11, 17
- xming, 17
- XQuartz, 17