

1. Introduction. Getting Started



1.1 Introduction to the Course

- Course aims
- Assessment
- How to learn to program and do well on the tests
- Class schedule

1.2 Introduction to Programming Languages

- Objectives
- What is programming?
- What is a programming language?
- The level of a programming language
- Types of programming languages
- Other important languages

1.3 Introduction to Mathematica

- What is Mathematica?
- Rationale for adopting Mathematica's language

1.4 References and Resources

- Recommended text
- Bibliography
- Wolfram Research

1.5 Getting Started

- The Front End and the Kernel
- Notebooks
- Input
- When you do not get what you expect
- Basic Syntax
- Cells
- Referring to previous output
- Getting help

1.6 Exercise: A Tour of Mathematica

- Touring Mathematica in the Help Browser
- Introductory topics
- Important but not introductory topics

1.1 Introduction to the Course

Course aims

The aims of the Programming for Engineering course are:

- To understand different high-level programming paradigms useful for modeling engineering systems.
- To be able to create and document computational models for the exploration, design and optimization of engineering systems.

The course will use the programming language built into the Mathematica software.

Assessment

■ Tests [70 marks]

You will be assessed at the Laboratory Classes (weeks 4 to 11 inclusive) on an understanding of *all* the previous work, but with emphasis on the previous week's work. The assessment will take the form of an individual *open book* test during the scheduled laboratory class, using the software in the laboratory. Class participation will be included in this mark.

Each test should be answered in a fresh Mathematica notebook, and by starting a fresh session of Mathematica. The test number, your name, and student number should be included at the *top* of the notebook. The notebook will be assessed by your instructor during the laboratory session, after the test is completed.

There will be 8 tests of which those 7 with the highest marks will be used. Each test will be worth 10 marks, making a maximum of 70 marks.

Students who fail a test, or who cannot sit a test for a good reason, will be given the opportunity to sit again for a test on the whole semester's work at the end of the semester. However, the maximum marks obtainable in these circumstances will be 5 marks for each test failed or missed.

■ Project [30 marks]

You will also undertake an individual Programming Project towards the end of the semester worth 30 marks. This project will require you to program the solution to a technological problem, document and package your program, and create a User Guide suitable for use by other technical professionals.

The laboratory classes for weeks 11 and 12 of semester will take the form of Workshops in which you can work on your project and get assistance as required.

It is expected that *all* students will attend these workshops, as there may be new material introduced or solutions to common problems discussed.

How to learn to program and do well on the tests

- Do not underestimate the amount of practice you need to become comfortable programming.
- The time spent in the lab classes is *not enough*.
- Doing just the exercises and problems in the handout notes is *not enough*.
- You need to practise at least another 4 hours per week.
- The notes and lab work are only an outline to the topics.
- You need to take charge of your own learning. Read the relevant sections in one of the texts and explore them, testing out your own understanding until you are comfortable.
- You need to develop *familiarity* with the functions available to you.
- I recommend you get *The Mathematica Book* and use it.

A notebook of simple review and *example* past test questions and solutions is included in these Course Notes. The best way to do well on the tests is to ensure you understand how to do these past test questions.

Class schedule

The main activities where you will *learn* programming are in your weekly scheduled computer laboratory class *and* in the work you do in the student laboratories (or at home if you have a student version of Mathematica on your own computer) expected of you in your own time.

Your objective in the laboratory classes is to ensure that you have fulfilled the learning objectives for each topic as stated in the course notes. The course notes have been designed to enable you to read them and attempt the exercises and problems in your weekly self study time *before* the relevant laboratory class. You are encouraged to make frequent reference to the prescribed text (*The Mathematica Book*) for clarification and further exercises on the concepts introduced. It is intended that the scheduled laboratory class be reserved for addressing any problems you have encountered.

WEEK	LABORATORY TOPICS	#	TESTS
1	Introduction. Getting started		
2	Expressions. Evaluation		
3	Rules. Patterns		
4	Writing functions	1	Test on weeks 2 & 3
5	Using modules	2	Test on week 4
6	Functional programming	3	Test on week 5
7	Rule - based programming	4	Test on week 6
8	Graphics programming	5	Test on week 7
9	Advanced techniques	6	Test on week 8
10	Packaging your program	7	Test on week 9
11	Project workshop	8	Test on week 10
12	Project workshop		
13			Project due

1.2 Introduction to Programming Languages

Objectives

To understand the concepts of *programming language* and *syntax*.

What is programming?

Roughly speaking, programming may be viewed as telling a computer (or some software residing on it) how you want it to deal with inputs of a given type that you may give it later. For example

$$\square^2 + \square^3$$

may be viewed as a program that is telling some computer to add the square and cube of *any* input (represented by the blank square \square).

When we feed an input (say, 4) to the program ($\square^2 + \square^3$), the computer replaces the blanks with our input, and gets

$$4^2 + 4^3$$

It then looks at its *inbuilt rules* to see if it has any which can apply to this. If it does, it applies them. The process of applying the rules is called *evaluation*.

$$4 \rightarrow \square^2 + \square^3 \rightarrow 4^2 + 4^3 \rightarrow 80$$

Input \rightarrow **Program** \rightarrow **Evaluation** \rightarrow **Output**

What is a programming language?

A *programming language* is the language which the computer (or its software) has been set up to understand. If you feed it a string of symbols which it recognizes, you have used its programming language. For example

Sin [π]

is something that Mathematica understands because it has *inbuilt rules* that it can apply to it (for example, it knows how to simplify this to **0**).

However, if you try to say

sin (π)

to Mathematica, it will not understand, because it has no inbuilt rules for dealing with such a string of symbols.

■ Syntax

The *syntax* of a language is the set of rules which tell us which are the valid strings of symbols in the language. That is, it tells us how to 'spell' and organize our commands correctly so that the software understands what we are saying. Most computer languages are not smart enough to allow for even a small spelling mistake. Mathematica goes some way towards this by letting you know if it detects a symbol name that differs by one letter from one it understands.

Getting the syntax right does not guarantee of course, that your program will produce meaningful results. It is easy to write grammatically correct nonsense.

A programming language which has only a small number of rules by which you can tell if the syntax of a string is correct (that is, understandable by the computer software) is clearly a simpler language to learn. We will see that this fortunately is the case with Mathematica.

The level of a programming language

Programming languages may be classified according to their level. Roughly speaking:

Low-level programming languages speak directly to the computer (which is not very intelligent without some additional software) and need to use binary machine code (strings of 1's and 0's). The language is extremely simple, but because of this, programs to do even the simplest things are likely to require a lot of commands. Moreover, they are quite inscrutable to us humans. You may use a low-level language when you program a micro-processor.

Higher-level programming languages (like Fortran, Pascal, Basic and C) are more comprehensible to us humans, and require less commands for accomplishing the same task than a low level language may require. The programming software does the translation of your higher level program into the low-level binary machine code. Although you can create a program of any sophistication with any of these languages, the usual practice, if you want to do anything complex is to rely on a preprogrammed library of functions which you attach to your program.

High-level programming languages (like those behind AutoCad, Excel and *Mathematica*) go one more step higher. They are generally written with a particular type of task in mind, but for completing that task are much easier to use than a language like Fortran, Pascal, Basic or C. Again, the high level languages must eventually be translated into something intelligible to the computer (that is, machine code) and are often written therefore in some variant of C.

For example, *Mathematica* is written in a variant of C which has been extended to deal with symbolics.

Types of programming languages

The two principal types of programming languages are the *compiled* languages, and the *interpreted* languages.

Roughly speaking, a compiled language requires you to write a complete piece of a program which the computer processes (compiles) before it is available for use. Compiling allows the computer to optimize the program you give it. Hence compiled programs are often best for

speed of execution. Fortran, Basic and C are typical compiled languages. They can handle numeric expressions but not symbolic ones.

An interpreted language, on the other hand, is more like a *conversation*. You write something and then enter it in to the computer. The computer will then give some sort of response immediately. Because of this immediate response, interpreted languages are ideal for exploring, prototyping and programming, but are generally slower for speed intensive calculations than compiled languages.

Mathematica is an interpreted language, but has an inbuilt compiler for speed intensive calculations. It can also handle symbolic expressions.

Other important languages

There have been many other computer languages created over the last several decades. Each of these had different characteristics, and were often designed with specialized objectives in mind.

Three of particular interest to engineers have been Prolog, Lisp and APL.

Prolog was designed to make it easy to program logical arguments. It was adopted in Japan as the language in which to write artificial intelligence software.

Lisp stands for *List* processing software, and was designed with extremely simple syntax (everything was a list). This was the language chosen by the western world for writing AI software.

APL stands for A Programming Language. It was designed to make computations with arrays (for example, lists and matrices) of numbers very easy to program (this was tedious in Fortran for example). It had a lot of advantages for engineers because we are often dealing with data sets, vectors and matrices.

There have also been languages designed for more mathematical computations based on the notion that all operations can be viewed as functions. Functional programming then consists simply of applying functions.

Although each of these languages has been designed to facilitate a certain type of programming and computation, each of them was a "fully fledged" computer language. That is, *any* computation could be done in *any* of the languages.

Stephen Wolfram, who designed Mathematica, was familiar with all these languages. His design has enabled him to incorporate all the powerful features from each of them into the one language, from programming with logical expressions and functions, to programming with lists and arrays.

1.3 Introduction to Mathematica

What is Mathematica?

- Mathematica is a 'system for doing mathematics'.
- It is a powerful high level computer language suited for engineering computations.
- The language is an interpreted language. It converses with you.
- It knows how to integrate, differentiate, plot graphs, invert matrices, find eigenvalues, solve equations, find Laplace transforms, compute Fourier series, ...
- It computes both numerically and symbolically.
- It allows visualization with graphics, animation and sound.
- It is a programming language with multiple paradigms: procedural, functional, rule-based, object-oriented.
- It is a calculation engine that can be connected to other software: Excel, Word, Spyglass, MATLAB, LabView, Visual Basic.
- It is a technical report documentation system with sophisticated mathematical typesetting.
- It is a software package prototyper, allowing interaction with the user via buttons and palettes.
- It has inbuilt converters for converting to or from other languages and file formats: C, Fortran, Tex, Postscript, HTML, AutoCad.
- It can generate MathML code (the mathematical equivalent to HTML) for inclusion of mathematical symbols and computation into web pages.

Rationale for adopting Mathematica's language

Standard software packages do the things they were designed to do, often very well. As an engineer, if you find one that does what you want, it is generally economically sensible to use it.

However, we hope that at least some of your career will be involved in *innovation*. And because innovation, by its nature, is new, you may not find existing software that does what you want. Hence you need to be able to write programs yourself.

You also need a system in which you can document your programs and calculations into the one 'live' report.

But most importantly, modern competitive engineering is about being able to get *reliable* products out onto the market place before your competitors. This means that you must *design* the products carefully. Good design means being able to *predict* performance before letting the product loose on the customers. That means we have to mathematically *model* the product's performance and failure modes. A programming language that has all the mathematics we are likely to need already built in will give us the opportunity for a competitive edge. We can use more sophisticated mathematical models and be more assured of getting them right.

We have chosen Mathematica's programming language, because we feel it will give you a competitive edge in your professional careers.

1.4 References and Resources

Recommended text

Wolfram, Stephen
The Mathematica Book
Fourth Edition, Wolfram Media/Cambridge University Press, 1999

Bibliography

A good medium level book on *Mathematica* programming is

Gray, John W.
Mastering Mathematica: Programming Methods and Applications
Second Edition , AP Professional, 1998

Other books on programming are

Wagner, David B.
Power Programming with Mathematica McGraw-Hill, 1996

Gaylord, R J, Kamin, S N, and Wellin, P R.
Introduction to Programming with Mathematica Telos, 1993

Comprehensive texts for graphics programming are

Wickham-Jones, Tom
Mathematica Graphics Telos, 1994

Ruskeepaa, Heikki
Mathematica Navigator Academic Press, 1999

A good introductory text to the *Mathematica* interface (not programming) is

Glynn, Jerry, and Gray, Theodore
The Beginner's Guide to Mathematica Version 4 Cambridge University Press, 1999

Books on *Mathematica* for engineers and scientists are

Bahder, Thomas B.
Mathematica for Scientists and Engineers Addison Wesley, 1995

Robertson, John S.
Engineering Mathematics with Mathematica McGraw Hill, 1995



There are now hundreds of books applying *Mathematica* to various disciplines.


Wolfram Research

The Wolfram Research site has a comprehensive collection of books and other resources to help you learn and use Mathematica. Explore it at <http://www.wolfram.com>.

1.5 Getting Started

The Front End and the Kernel

- Mathematica is divided into Front End and Kernel.
- The Front End is the Notebook interface (what you see on the screen).
- The Kernel is the calculation engine (normally not seen).
- They talk via MathLink which can also connect other software.
- You can edit a notebook while the Kernel is doing a calculation.
- The Kernel can be run by itself with a command-line interface.
- The Kernel icon is . The Mathematica (Front End) icon is .

To fire up Mathematica, double-click on the Mathematica icon .

Notebooks

After you have fired up Mathematica you should see a blank untitled window. This is a new *notebook*.

Click in the notebook and type

2 + 2

Then press Shift-Enter or simply Enter on the numeric keypad.

This first entry starts up the Kernel and sends the expression to it, the Kernel then evaluates it and returns

4

Note that input is in bold type, but the output is in ordinary type.

Click in the notebook under the last output and a horizontal line will appear. This is where your next typed input will appear. (The horizontal line appears by default after an output, so you do not have to click if you see it there already).

Input

Type in each of the expressions below *exactly* as they are shown, evaluating each of them *as you go* by pressing Shift-Return after each complete entry. See below if you get into trouble.

```

a + a + b + b + b
N[Pi, 1000]
Expand[(x + a) ^ 8]
Solve[x ^ 3 + x + 1 == 0, x]
D[x ^ n, x]
Integrate[x ^ n, n]
Reverse[{a, b, c, d}]
Plot[Sin[x], {x, 0, 8 Pi}]
Simplify[(Sin[x]) ^ 2 + (Cos[x]) ^ 2]

```

You can press Shift-Return with the cursor anywhere in the cell.

When you do not get what you expect

If you enter something that Mathematica cannot understand, it will complain, and try its best to set you right. For example, try entering

```

N[Pi, 1000
Syntax::bktmcp : Expression "N[Pi, 1000" has no closing "]"
N[Pi, 1000

```

Here, there was a bracket matching problem, and it has told you so. In fact any purple brackets showing up in an expression as you type it means that it does not have a matching bracket. In the message, the offending expression is underlined in red.

Recovery is simple: just go back to the offending input line, edit it and enter it again. If you have done it correctly you should receive no more complaints.

Another common error is to spell commands incorrectly. Try entering

```

expand[(x + a) ^ 8]
General::spell1 : Possible spelling error: new symbol
name "expand" is similar to existing symbol "Expand"
expand[(a + x) ^ 8]

```

Notice however, that in this case, the output you get is just the same as your input. (Although the ordering of symbols in sums may be changed, and the notation might be more mathematical, these differences are not important).

Again, the solution is to replace the **expand** by **Expand** in the original expression and press Shift Return again.

When you get back exactly what you entered, you have probably entered a syntactically correct expression, but one for which there are no inbuilt rules for doing anything with it.

If an expression is syntactically correct, but not 'close' to one for which there are inbuilt rules (like **Expand**), there will be a simple return of your original input, with no message. For example

```
f[x, y]
```

```
f [x, y]
```

Basic Syntax

■ Naming conventions

Mathematica is *case-sensitive*. That means, for example, that **A** is a different symbol to **a**.

All built-in names and symbols start with a capital letter: **Integrate**, **Plot**, **D**, **E**, **Pi**, ...

If a name consists of two or more words, the first letter of each is capitalized, and there are no spaces: **NIntegrate**, **ListPlot**, **NDSolve**, ...

Most names are full. Abbreviations are used only when they are more commonly used than the full name, like **Abs**, **Cos**, **Det**, **D**, ...

■ Parentheses, brackets and braces

Parentheses () are used just as in algebra - to group terms to force the correct order of evaluation. For example **(a-(a-b))**.

Brackets [] are used (unlike in mathematics) for all functions. Using brackets instead of parentheses for functions has been necessary to ensure that the parser can interpret input unambiguously. For example, the difference between the *function* **sin(x)** and the *product* of the symbol **sin** with the symbol **x** would not be clear to a parser. On the other hand the difference between the function **Sin[x]** and the product **sin(x)** is clear.

Braces { } are used for lists. For example **{a,b,c}**.

■ Commas

Commas (,) are used to separate the arguments of a function, or the elements of a list. For example **{f[x,y,z],x,y,z}**.

■ Spaces and multiplication

Multiplication can use either an asterisk (*) or a multiplication sign (×). But the simplest method is simply to use a space (or any number of spaces). For example, each member of the following list is interpreted as **2 times 3**.

```
{2 * 3, 2 × 3, 2 3, 2   3}
```

```
{6, 6, 6, 6}
```

Spaces are generally ignored *except* where they could be interpreted as multiplication.

Spaces are not needed when multiplication is clearly intended, for example, when a number precedes a symbol, or when a number or symbol is outside parentheses.

```
{ 2 a, 2 ( 3 + 4 ), a ( 3 + 4 ), a ( 2 b ) }
```

```
{ 2 a, 14, 7 a, 2 a b }
```

Cells

As you are doing the calculations above you will notice large blue brackets down the right hand side of your notebook. These indicate *cells*.

Cells are a way of organizing the various types of material in your notebook into a hierarchy. Input and output cells are grouped together. Heading cells group what is under them. (You can choose a heading style from the Format:Style menu).

Double-clicking a cell bracket which spans other cell brackets will hide what is beneath the top cell. Double-clicking again makes it reappear.

Referring to previous output

Most often, you will want to use the outputs you get as inputs in further computations. You can refer to the immediately last output with the % sign. For example, if you expand $(x+a)^5$, you can write **Factor[%]** to check the result.

```
Expand [ ( x + a ) ^ 5 ]
```

$$a^5 + 5 a^4 x + 10 a^3 x^2 + 10 a^2 x^3 + 5 a x^4 + x^5$$

```
Factor [%]
```

$$(a + x)^5$$

However, this connection is very dependent on the *order* in which you perform a series of calculations, and this order may be different if you re-evaluate your notebook at a later date. The % should only be used as a temporary measure.

The best way to refer to previous output is to give it a *name*. Invent a name, preferably one starting with a *lowercase* letter so that you do not try to use a name that is already reserved for internal use (like **N, D, Sin, Log, ...**). (There are over a thousand functions built in to Mathematica). From then on, refer to your result by its own name. For the example above we might enter:

```
poly = Expand [ ( x + a ) ^ 5 ]
```

$$a^5 + 5 a^4 x + 10 a^3 x^2 + 10 a^2 x^3 + 5 a x^4 + x^5$$

Entering **poly** then gives us the previous output:

poly

$$a^5 + 5 a^4 x + 10 a^3 x^2 + 10 a^2 x^3 + 5 a x^4 + x^5$$

Thus, from now on we can use the symbol **poly** anywhere in future computations instead of the longer expression

$$a^5 + 5 a^4 x + 10 a^3 x^2 + 10 a^2 x^3 + 5 a x^4 + x^5.$$

Factor[poly]

$$(a + x)^5$$

Do not be frightened of using long names if they are more descriptive. It takes little time to type a few more symbols. But we can waste a lot of time trying to remember confusing names that are too abbreviated.

But when you no longer need a name, especially a temporary short name like **x**, you should remove it using **x = .** to avoid confusion when you include **x** in a calculation later (expecting it to be just simply the symbol **x** without a meaning).

△ Keep your desk clean with □ = .

For example if we no longer need the name **poly**, we can enter

poly = .

Now **poly** is no longer assigned as a name for the polynomial above. If we enter it now we simply get it returned.

poly

poly

Getting help

■ The ?

For a quick, brief definition of any of the inbuilt functions, you can use **?** before the name of the function. For example, entering **?D** gives

?D

D[f, x] gives the partial derivative of f with respect to x. D[f, {x, n}] gives the nth partial derivative of f with respect to x. D[f, x1, x2, ...] gives a mixed derivative. More...

You can also use an asterisk as a "wildcard". For example **?Z*** gives all the functions starting with **Z**.

? Z*

System`

[ZeroTest](#) [ZeroWidthTimes](#) [Zeta](#) [ZTransform](#)

The links in blue enable you to get more information.

■ The Mathematica Book

The Mathematica Book by Stephen Wolfram, is the recommended text for this part of the subject. The *Help Browser* (see below) contains the full text online. However for a real understanding of the immense capabilities of Mathematica to assist you in your assignments and projects throughout your course, and later in your professional career, it is recommended that you purchase the text.

■ The Help Browser

The Help Browser is under the Help menu. This is an extremely comprehensive resource, and you should become familiar with what it contains.

1.6 Exercise: A Tour of Mathematica

Touring Mathematica in the Help Browser

Work through the notebooks below by making a (sensible!) change to some item in the input to each of the examples that makes sense to you, and evaluating the notebook to see the new result. Sometimes you will not get the result expected, since only those examples which use built-in functions are likely to work interactively.

Do not worry about changing the state of the Help Browser, it will return to its original condition when you close its window.

From the *Help* menu, select *Getting Started/Demos ...*, then *Tour of Mathematica*.

Introductory topics

Mathematica as a Calculator.

Power Computing with Mathematica.

Accessing Algorithms in Mathematica.

Mathematical Knowledge.

Building up Computations.

Handling Data. (Do not try to change anything here).

Visualization with Mathematica. (You may need to evaluate the function to get the sound to play).

Mathematica Notebooks. (Just read this).

Palettes and Buttons.

Mathematical Notation.

Mathematica and Your Computing Environment. (Just read this).

Important but not introductory topics

The following topics are important to your understanding of the capabilities of Mathematica as a programming language, but at this stage you are not expected to understand the detail. As you progress through the subject, come back to these to see if they make more sense.

The Unifying Idea of Mathematica.

Mathematica as a Programming Language.

Writing Programs in Mathematica.

Building Systems with Mathematica.

Mathematica as a Software Component.