

Chapter 0

Quo Vadis?

How to avoid reading this book

There's a certain degree of excitement when new software's been loaded onto a computer. What can it do? What will we get out of it? Textbooks don't provide such an alternative and many people avoid reading them because of that. That is why there is a software option to accompany this text. It is the second version of Chapter 0 – flick through to it now, and maybe start there.

Some people enjoy the prospect of sitting down with the documentation for their new software and getting down to the nitty-gritty, working through it in detail and getting to know all the features of their new program in a thorough, methodical way. Others, however, want to get right into it, do something with it right away, something useful. They try to do whatever they can immediately and see how far they can get without reading anything. They will learn what's required if and when they have to. Being practical we have to find a balance between these two extremes, between avoiding confusion and avoiding boredom and each person has his own way of finding this balance.

If you happen to be in the unfortunate position of having to pass some sort of exam or tests based on what's discussed in this book, and the prospect of reading it doesn't appeal then it makes sense to start doing the sorts of things that you could be asked to do in such an exam. One way of coming to grips with such tasks is to write computer programs to do them. It's particularly relevant here because that is what this text is all about: computer programs.

Because you cannot write instructions for a computer unless you yourself know precisely what has to be done, writing programs makes sure that you know what the topics are about. Unless you do get it right, the computer will not give you the answers you want. It presents you with the challenge: *get it right*. It tests you on whether you have succeeded or not — yes or no — all or nothing.

There is another little electronic book to guide people who need some help on getting started with writing computer instructions. Of course the way to avoid reading that book is to read this one; but try to avoid reading either of these books as much as you possibly can and doing as much as you can on your own.

It may turn out that reading bits of this book now and again will help. One should appreciate that a textbook cannot be written like a novel, where through some 950 page saga the hero learns through bitter personal experience and tragedy the significance of round-off errors. The textbook is a synopsis of the significant facts and as short as possible. It should be a reference book with explanations for first time readers. It is not intended to be read from cover to cover and though some material may depend on understanding earlier work, one should be able to read any section in isolation.

Talking to different people, students, friends and instructors might also help. Other books might help. Material you find online might help. If you work hard enough then it is highly likely that you can get to know everything that is discussed in this book, whether it be for the sake of an exam or just for your own general knowledge, without having to actually read much of this book at all but by just glancing through headings and checking with others that you haven't missed anything. You have to find your own path in your own way.

Some justification for this book's existence

What does someone interested in building their professional career around involvement in the computer industry need to know about mathematics, as the sub-title of this book might suggest? It seems like a reasonable question to ask, but it is the *wrong* question.

The sub-title does try, however, to put the proper slant on the approach taken. This book may provide a few mathematical facts and some mathematical techniques, but its major aim is to engender certain habits of thought that come from a mathematical perspective.

Only mathematicians study mathematics for the sake of the mathematics. Computing, like any other science, or systematic body of knowledge, has its own organization and structure. That is exactly what *mathematics* is: an outline of the structure and order (in an abstract sense) in particular areas of application. Every area of endeavour has its own mathematics. The correct question to ask then is why would we want to express the basic structural elements of computing in abstract mathematical terms.

The answer is easy: it gives us more flexibility and professional competence. You can check the truth of this assertion yourself by thinking about any activity you might have been involved in, or even in any of the games you might have played. Once you understand the principles underlying the activity, your performance lifts from that of a naive player to one who develops strategies (based on principles). Another way of saying it is that to get better at anything we always go looking for "the idea behind it".

Even someone dealing with nothing more than word processing must understand the organization of the filing system on a computer; that is, they must understand the principles underlying the creation of sets and subsets, whether these are called directories or folders or whatever. Extensive use of a system will make one subconsciously familiar with these principles, but they may remain clouded by the special terminology of a particular system. Seeing the structure for what it is in abstract (or *mathematical*) terms breaks one free of any particular terminology and capable of comprehending the organizational principles. The competent professional has both extensive experience and an understanding of the principles.

This book tries to provide a framework into which you can cast some of your experiences. It tries to speed up your ability to comprehend the principles that permeate the lives of computer specialists. It is not trying to teach you mathematics. It is trying to help you learn computer science.

0.1 The Nature of the Beast

From silicon chips to software packages there are innumerable levels at which people work and play with computers. The less we need to know to get our job done, the happier we are, but the more we know and less we have to think about what needs to be done, the more efficient we are. Occasionally we can't get our job done because we don't know enough. What we need to understand and know depends on where we fit in to the big picture. We don't need to understand about bits and bytes if our focus is producing documents; we can happily enhance our photographs without needing to know anything too much about pixels. So what is the big picture; what is the nature of the machine?

When we switch a computer on (*i.e.* when we *boot the computer*), an operating system is established in the central processing unit. Various hardware is recognized: *input devices* such as keyboard, mouse, scanner and microphone; *output devices* such as screens, printers and speakers ; *storage devices* such as the computer's built in (or hard) drives and devices with removable storage elements such as DVD drives that may be built in and cameras that may or may not be connected to the computer; and *network connection* hardware which whether by cable or wireless connect the computer to other computers. Then, the software is made available for use.

With the input devices we send messages to the computer; it sends messages back to us with the output devices. In the storage devices it stores all the (software) programs and data it can use to interpret our message and generate its response through its central processing unit.

The function of a computer is to communicate with us, whether it does so in a car, giving information about where we are and what shops there are around (that would hope to get some of our money), or on the street, giving us messages sent by our friends' computers or those of people

hoping to get some of our money, or at home or at work (where we're hoping to get some of our money from someone else). Once upon a time, people who got too absorbed with communicating with their computers were sometimes called *nerds*. Today, it's hard to imagine life without computer communication.

Various buttons or icons may appear on the screen once the computer is booted up. These represent different programs that we might like to try. The input required from us is to simply click on the icon. The output from the computer may be to display something on the screen (such as the time and date), or to play some music or makes some sound or to open a window on the screen in which we may enter, by clicking, typing or talking, new instructions or data. Such a window represents an environment, or *workspace*, which is a sub-system of the overall operating system, and in which messages can be sent asking for the execution of tasks more specialized than the operating system can interpret.

For example, in the *Windows* operating system, pressing the *start* button may lead to viewing a list of all programs. If the Microsoft Office package is installed, then we could choose *Word* from among these programs. If we do, then we will be presented with a fresh window for text input. Most importantly the operating system makes a workspace available for running the *Word* program. Within this workspace, a variety of word processing tools can be used. Each tool is just another program for which the *Word* program will make a workspace available within its own workspace. For the *Replace* command, several pieces of input are required: a block of text (either highlighted or otherwise identified), the characters to find (usually a word or phrase) and the characters with which they are to be replaced. The output is a block of text with the replacements made.

If we had chosen *Excel* from the list of programs in the *start* menu, then a blank spreadsheet would be presented and various spreadsheet tools would have been made available in that workspace. Here we would have seen that to use some tools requires us to type instructions onto an *input line* in the *Excel* window. Having icons for everything becomes too messy when there are lots of functions. Ultimately typing instructions for an input line (or *Command line*) is the preferred option.

In mathematical terms we would describe the interaction with a computer as being the execution of some *function*: input is processed to give an output. It is entirely analogous to the simple functions we deal with in

elementary mathematics: *e.g.* 3 (some data) squared (a specified process) returns 9 (a result). With different input data, of course, we expect that there could be different results, but the relationship between input and output is always the same: whatever number is input to the square function the output is always that number multiplied by itself. If a function has no inputs then the output should always be the same.

Notice that we should be careful about what exactly the input data is. Suppose we *Open* a file by (for example) clicking on a button labelled *Open* and then choosing the name *fred.doc* from a menu of file names. The input to the function *Open* is the name *fred.doc*. However, what we see on the screen as a result may well be different from what we saw when we opened the file called *fred.doc* yesterday. The contents of the file may have been changed. We seem to have a different output from the same input. But this is not so, because *fred.doc* is *not* the data but just a name for the data. We can change the name of the file and still get the same output because the input — what is in that file — will be the same. If we give another file that particular name, then the input is different even though the name given to the *Open* function is the same.

The *name* of the file is only a pointer to particular data.

File names are variable and when they are entered as input to a function it is the data they point to that is used by the function.

Similarly if *My_number* is 3, then the square of *My_number* is 9, but tomorrow *My_number* might be 15. In mathematics classes it is common to use names like *x*. Such names are like the pronouns, *he*, *she*, *it*, *they* etc. which may refer to different things at different times or in different contexts. They are referred to as *variable names* or simply *variables* because the data that a particular name represents might vary.

One of the most important functions in any environment is the process of saving files. Invariably this function is called *Save* (in English) and its input is the data in the file and a name for the file. The output is data written to a storage unit and associated with that particular name. The software that establishes and runs the functions available in any particular environment is itself stored in a collection of files.

Effective programming depends crucially on the use of variables. The usefulness of a command like *Open* depends on its being able to open

a file (perhaps restricted to a particular type) no matter what its name might be. This means that the list of instructions written to implement an *Open* command must be able to cope with a variable name for the file which is to be opened. One cannot over-emphasize the importance of variable names that point to the location of specific data. So we see that it is in its very nature of computing to manipulate symbols that represent something else.

A computer is a beast that does things, it carries out certain functions. The instructions for performing these functions are held in files, whose names are *variables*, and the instructions must refer to working with the data found under different names (*variables*). Any data that we input, or obtain as output from some process, are, if saved, stored in files under *variable* names. So the computer is very much a beast wandering in a world of symbols.

0.2 Computer Programs

When we browse through the directory on a computer hard drive or a disk of downloaded software we see files that are identified as plain text files, by the .txt extension, files that are picture files and identified by .jpg or .gif (for example) and many others which hold various types of information and can only be opened from within certain applications. The applications themselves might be run by programs in files with a .exe extension – executable files – programs that will run to set up an environment in which certain types of task can be done, like *Notepad* (for reading and editing text files), *PhotoStyler* (for viewing and editing pictures), *Warcraft* for playing a game, *Excel* for a spreadsheet or *Mathematica* for working with mathematical operators.

These compiled programs are large pieces of work, sometimes the products of years and even decades of work by dozens or hundreds of people. The process of software development is in itself a whole area of study with it's own principles of operation.

Within each environment there is a collection of tools (just another name for smaller programs) on which the success of the software depends. Tools must do the things that people want to do in particular environments, be

it word processing, image processing, engineering design, mathematical analysis or whatever. Not only must the tools be the right ones, they must also be easy to use and versatile, *i.e.* useable in many different situations.

Unlike the large scale drivers of applications which have no input and establish a computing environment, tools will normally have variable input. You can draw a rough analogy with a trained carpenter and his toolbox or a plumber and his toolbox but a computer toolbox is different in that the tool actually 'knows' how to do the job. It 'knows' how to do the job because not only does the tool have a name, but there is also a file in which a sequence of instructions is listed and which will be executed in order by the computer's processing unit on demand. It is more like the tradesman and his robots.

"*Garbage in – garbage out*" might remind us that no amount of analysis can improve the reliability of the original data, but it assumes that in using a computer some information (data) will be provided by a user and that, after some processing, the computer will return certain information. The *input* data may involve a number of variables as may the *output*. The time at which data is given to a computer might also be a part of the input if the processing involved depends on time. For example, retrieving a bank balance may well depend on when the request was made. The computer is expected to be absolutely reliable in returning information in that, if those features of the data that affect the processing are the same in two separate inputs, then the outputs should be identical. *Processing identical input data returns identical output data*. If we can't be sure of that, then the machine or the processor is faulty and unusable.

In mathematical terminology the steps of the process used to produce a unique output from a particular input constitute the *algorithm* for a *function*. An *algorithm* is a description of a particular process. A *function* is the relationship between the input and the output.

There may be different ways (algorithms) of getting to the same result. For example, for an input of some number, represented (or pointed to) by the variable name x , a *doubling* of this number can be achieved by either performing the addition $x + x$ or the multiplication $2 * x$. The relationship between input and output is *doubling* in either case. Some algorithms may prove to be faster than others. In fact, we will see examples later in this book of algorithms that take years to compute something that another

algorithm can accomplish in seconds. It can be shown theoretically that some algorithms would eventually finish a job but that the time taken would be longer than the age of the universe. So, coming up with an algorithm might not necessarily be the solution of a problem.

In mathematical terms we see a computer as something that can store and run algorithms that implement various functional relationships. Functional relationships are given names such as *twice* or *Word* and there will be specific instructions on how these functions can be invoked. Some functions may require no input. It follows that they must have a unique output, such as delivering the data for a complete environment of activities involving many other functions. The process of booting a computer with a particular operating system is in itself a function that produces a particular environment in which various other software packages (collections of functions) may be run.

To sum up then, a computer implements *functions* by means of instructions stored in *algorithms* which use data stored in *variables*.

0.3 Functions and other Relationships

The abstract definition of a *function* is that it is a relationship between two variables called the input and the output for which it is assured that identical inputs return identical outputs.

There are other relationships between pairs of variables besides functional relationships. For example, if the two variables both represent fractions then certain pairs are related by the fact that they represent the same rational number; for example, $21/28$ and $27/36$ are fractions that both could represent the decimal fraction $0.75 = 75/100$. We say that they are *equivalent* fractions. There are many fractions equivalent to 0.75 . In fact, every fraction is equivalent to a host of other fractions all equal to the same rational number. This is an example of an *equivalence* relationship between pairs of variables. Every time we categorize things into distinguishable groups we are expressing an equivalence relationship between the members of each group.

Ordering relationships on a set of things may be defined if there is some sort of measure in terms of a real number for each member of the set.

If, for example, the measure is the floor space of an apartment, then apartments could be ordered in terms of this measure. There are other ordering relationships that do not depend on a measure, but note that a measure should be a function of the things to which it is applied.

With every relationships there is a test that can be applied to see whether there is a given relationship between two objects or not. It may be a comparison of measures or some other test which returns a result of *yes* or *no*. Such a test is necessarily a function (called a *relationship function*). It is an example of a *Boolean* function — *i.e.* a function that has only two possible results. For example, two fractions, a/b and c/d are equivalent if it is true that $a \star d = b \star c$. Other common Boolean functions, besides the test of equality function ($=$), are the arithmetic comparison functions $>$ (greater than), $<$ (less than), \neq (not equal). Such test functions are very important in computing where what is done next depends on whether certain relationships hold.

Complicated relationships will have relationship functions that are difficult to evaluate. Some algorithms may be impossible to describe in simple terms, such as the procedures to be followed to establish a word-processing environment. Some functions may have no known algorithm. For example, if the function is to return the n^{th} prime number. Certainly given any input n there is a unique n^{th} prime number, but our only way of finding it is to look up a table of primes. Thus we know that the first prime number is 2, the tenth prime number is 29 and the five hundredth prime is 3581. When the table runs out, we have to do a lot of work to find function values beyond the range of the table. We know there is a ten thousand billionth prime but we have no way of finding out what it is, because, in this case, as far as I know, the only way to do the job is impracticable and cannot be completed in our lifetime.

In the following pages we will restrict ourselves to simple, mostly arithmetic, functions for which we can write easily developed algorithms but one should not be fooled into thinking that all function algorithms are therefore simple, maybe even just one line formulae. If it were so, there'd be little need for computers. Computers are so magnificently powerful since they can compute intricately complicated functions and use the results in other functions to produce all our software. The algorithms for these functions of course all involve *variables* that represent all sorts of data. Our aim is to gain some insight into this process.

Chapter 0

The J way

How to avoid reading this book

This is a companion workbook for the text *Discrete*. It's not meant to be something that's just read but a guide to do things on a computer that illustrate and support the discussion in the text. Apart from the exercises that are designed to illustrate particular points, this book gives some hints on things you might do. You can do a lot more and a lot differently. Discovering this for yourself and playing around with the different things that you can do is the best way to gain understanding. Don't sit down to read this book. Sit down to work on your computer.

- **Download J**

First of all download **J** to your computer from

<http://www.jsoftware.com/>

and start it up.

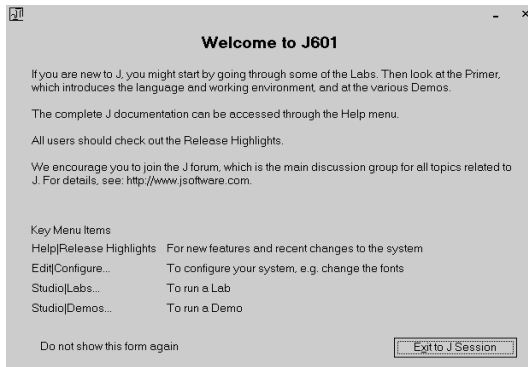
Some justification for this book's existence

J is a powerful symbolic array processing computer language invented by Kenneth E. Iverson and developed by him, Roger Hui and their associates. It is a sound logical structure which makes it particularly suitable for illustrating basic ideas in computing and mathematics. Among the most important of the features of **J** is the ability to combine and modify functions in ways similar to the way we linguistically modify action verbs.

This workbook might seem at first sight to be an introduction to **J** but there is much material available with the **J** package for that purpose. The goal of this book is to use **J** to provide executable and concrete illustrations of those basic principles and ways of thinking that are discussed in another little book by the author called *Discrete*. The chapters and sections of this workbook parallel those in the *Discrete* book. Only those features of **J** needed to achieve this aim that will be introduced here.

0.1 The nature of the beast

The **J** welcome screen invites you to start out with some of the pre-prepared *Labs*. It also lists a few Key Menu Items which you can access from the **J** window menu bar after you exit the welcome screen.



When you choose to **Exit to J Session** the cursor will appear at the top of a blank screen with a menu bar at the top. This is the **J** workspace. The cursor is on what is called the *Command Line*. Instructions are typed in on this line for the interpreter to execute immediately. **J** is not

a language that has to be *compiled*. It is an *interpreter*. The results of instructions such as 3+4 will be displayed on the screen when the *Enter* key is pressed anywhere on the command line provided, of course, that what has been typed on the command line can be interpreted. We say that the instruction has then been *executed*. If it cannot be interpreted, as in say 3+, then an error message will appear.

The arithmetic operations of addition, subtraction, multiplication and division are indicated by +, -, * and %. These symbols are the function names for the operations. Inputs are written on the left and the right of the function names.

It might come as a pleasant surprise to see that if we have a long list of numbers that we would like to multiply by say 0.15, then we can simply write

$$0.15 * 2.7 3.1 4.2 5.8 6.4$$

and **J** will interpret this as a command to multiply every number in the list by 0.15.

J accepts whole lists of numbers as inputs. So to add corresponding elements of two lists together, one writes for example,

$$1 2 3 4 5 + 2.7 3.1 4.2 5.8 6.4$$

The lists must be of the same length otherwise the instruction will not be interpretable, except for the case in which the same number is to be added to, subtracted from, multiplied or divided by every element of the list.

A consequence of this is that negative numbers cannot be indicated by the common minus sign, because the instruction 6 -5 4 3 will result in 1 2 3. If we wanted the list in which the second element was negative 5, then we have to represent it differently. In **J** the negative is indicated by the underscore _ and the required list would be 6 _5 4 3. The symbol - is a function name and it must be respected as such and not used to indicate the sign of a number. ¹

The capacity to store large amounts of data is one of the most significant features of modern computers, so it is important that we know how to save data.

¹Some elementary texts use a high minus to indicate negatives, as in ⁻5, but such a symbol is not available on standard keyboards.

Within a workspace, data can be stored under variable names by using the assignment symbol

`==:`

The variable name must be on the left of this symbol and the data it refers to on the right. For example, we can store the numbers 6 5 4 3 under the variable name `frodo`, by executing the instruction

```
frodo ==: 6 5 4 3
```

To save the data to a file that we can access later, the assignment statement should be written into a text file. Text files can be executed in **J** using the menu bar in the execution window when we want to recover our variable. Workspaces cannot be saved as such but the variables used in them can be saved into text files.

To do this go to **File** on the menu bar and choose **New ijs**. This will open a text file the same as **Notepad** or any other text editor would open. The data you want to keep is written in to this file stored under variable names. Many variables may be defined in the one file and these will all be defined in the workspace when this `.ijs` file is **Run** in **J**.

The reason why symbol `%` is used for division in **J** is that the more common symbol `/` was used as in **J**'s predecessor APL for the modifier *over*. For example, `+ / 2 3 4` reads "sum over 2 3 4" and returns 9. Similarly, `* / 4 2 9` reads "product over 4 2 9" and returns 72. The modifier can be applied to any function and is so convenient that it is too hard to discard. In APL the symbol `÷` was available for division, but **J** has constrained itself to the characters found on the standard keyboard, so another symbol had to be found for division. The classical European usage of `:` for division was also not convenient. And so it turned out that `%` is used for division in **J**.

J allows the user to assign names to functions (operations) as well as names to variables. So for example one might execute such statement as

```
sum ==: +/  
product ==: */
```

if one would like these functions to have English language names, but the user is free to assign whatever names he wishes, using the available character set. Such assignments can also be saved in script (`.ijs`) files and recovered by running the appropriate file from the execution window

menu bar or from the command line with the `load` command.

Given the assigned names for function `sum` and variable `frodo`, the sum of the numbers in the list could be obtained from

```
sum frodo
```

- **Configure J** if you like: you can customize your window using different colour schemes under `Color`. Notice that you can attach your own choice of colour to various objects. For example, you might choose the `BlueJay` scheme but change the text colour to gold. There are other options to choose from as well in configuring your interactive window.

0.1.1 Exercises on the nature of the beast

- **Counting and Negative Numbers**

The `J` operator `i.` produces the counting numbers. Execute, for example,

```
i.21
```

to see what it produces.

For counting numbers starting at 1 (the *natural numbers*), execute:

```
1 + i.21
```

Shifting the numbers around will produce different ranges of integers:

```
(-10) + i.21
```

Notice how the negative numbers are represented. Check out the results from:

```
3 -4 5 6 9 7
```

```
3 4 5 -6 9 7
```

```
3 4 5 6 -9 7
```

```
i. _21
```

Show that the interval between 3.1 and 7.5 is be divided into 8 equal parts by the points

```
3.1 + (i.9) * (7.5-3.1)%8
```

Compute the points that divide the interval from `_4` to `6` into 16 equal sub-divisions.

- **Arithmetic and Precedence**

Examine the results from the following instructions

12 - 4 10 8 - 2 4 5	(12 - 4 10 8) - 2 4 5	12 - (4 10 8 - 2 4 5)
12 % 4 10 8 % 2 4 5	(12 % 4 10 8) % 2 4 5	12 % (4 10 8 % 2 4 5)
12 * 4 10 8 + 2 4 5	(12 * 4 10 8) + 2 4 5	12 * (4 10 8 + 2 4 5)
3 * 2 + i. 4	3 * i.2 + 4	i. 3 * 2 + 4

Notice that in the absence of parentheses, **J** works from right to left in evaluating expressions. The precedence laws of elementary arithmetic are not observed because then precedence rules have to be given for all other operators as well, for example, does `i.` come before or after `sum`.

The right to left evaluation sequence is the classical mathematical order. For example, the expression $\cos \log \tan x$ is evaluated by first giving x a value, then calculating $\tan x$, then the logarithm of this, $\log(\tan x)$ and finally the cosine of the value obtained from that, $\cos(\log(\tan x))$. In **J**, this order applies to all operators so that $5 * 3 + 1$ is not 16 but 20. To have the multiplication done first write either $1 + 5 * 3$ or $(5 * 3) + 1$. Expressions inside parentheses are always evaluated first, starting from the outer most set. For example $3*((2*5)-7)+4$ is the same as $4+3*_7+2*5$

- **Assigned Variables and concatenation.**

Forty years of rainfall data (in millimetres) can be stored in a variable called `Rain` by executing the statements:

```
Rain =: 438 360 381 594 635 289 481 283 568 381 383 431
Rain =: Rain, 465 335 400 156 430 443 406 535 268 812
Rain =: Rain, 598 441 269 231 471 369 328 400 123 527
Rain =: Rain, 336 395 379 366 445 488 278 321
```

In **J** all 40 entries could probably be typed in on the one line, but when a list is too long for that, subsequent lines of numbers can be concatenated² to the list as shown above. The `,` is a symbol for a function which just like `+` or `-` has input data to the left and right of the symbol.

While some lists are easy to generate; the years for which the rainfall data is recorded for example might be generated from

```
years =: 1950 + i. 40
```

we would not want to have to type in long lists of numbers or text whenever we wanted to work with it. In **J** the above lines could be typed into a script file which is saved to disc and run within **J** whenever the data is needed.

²To *concatenate* two lists is to combine the two in sequence to form a single list

- **Storing Data: script files and comments.**

In the Command Window select **File/New ijs** from the menu bar and type in the Rain data as above. Also add the line defining the variable **years** and the function **sum**.

Now select **File/Save As...** and save the file under a name of your own choice. Remember the exact directory path to this file so that you can find it again.

Select **Run/File** from the Command Window menu bar and choose the file just saved. Check that variable **Rain** now exists in your workspace by simply executing **Rain**. Check that **years** is also there. Execute **sum** — it should show the definition of the function.

Execute **sum Rain** to find the total rainfall over the period.

You may wish to add explanatory comments to the file so that anyone opening the file will know what it contains. Comment lines begin with **NB**. Select **File/Open** from the menu bar and select the file. Add some comments such as for example

NB. Stoney Creek Rainfall, 1951--1990 in millimetres.

Select **File/Save** to over-write the original file. To see the file contents on screen as they execute, select **Run/File Display**.

Notice that you can also **Run** lines or selections of lines of a script and that you can copy and paste data into a script file through the Clipboard. You should check however that each line is a **J** statement that can be executed.

- **Lists and Tables**

The number of elements in a list is returned by **#** in **J**. Thus **# Rain** should return 40; so should **# years**.

There are several ways of making a table out of these two lists. A two column table is produced by

```
years ,. Rain
```

and a two-rowed table by

```
years ,: Rain
```

The **J** function **\$** constructs a table by filling in the rows with entries from a given list. Thus with

```
L =: # Rain
```

the instruction

```
(2,L) $ years, Rain
```

will produce the same two-rowed table as above.

Function `$` can be usefully employed to construct tables with various patterns because it re-uses elements from the list given when the list is exhausted. For example, try

```
7 7 $ 4 1 0 0 0 0 1.
```

Construct a table (sometimes called a *matrix*) in which every element is 1.

Counting through a table is done by the `i.` function. Try

```
i. 4 5.
```

If we think of a table as a *page* of data, then several pages will make a data *book*. We can count through a book of 6 pages with

```
i. 6 4 5
```

Several books make up a *shelf*. Count through a shelf of 6 books each of 4 pages with 5 rows and 8 columns of data on each page. How many data items in the shelf?

The model of a libraries to describe collections of data is a limited vocabulary, so instead of persisting with it, such collections of data are referred to as (multi-dimensional) arrays. The data objects of **J** are arrays and **J** is an array processing language.

- **Storing Instructions: named functions**

Calculate the average rainfall at Stoney Creek over the forty years from 1951 to 1990.

To do this one needs to compute the sum of all the entries in the list, `Rain`, of recorded rainfalls and to divide it by the number of entries in that list. Define the function, `mean`, by executing the statement³

```
mean =: sum % #
```

Two useful functions for lists can be named by executing the assignments

```
behead =: }. and curtail =: }:
```

Describe what each function does to a list. Also execute

```
diff =: curtail - behead
```

and describe what it does to a list.

³Mathematics books define the quotient of two functions f and g as the function $h = f \div g$ which takes the values $h(x) = f(x) \div g(x)$. The product, sum and difference of two functions is defined by similar definitions. In **J** these definitions can be executed as shown, so that `mean(x)` returns `(sum x) % (# x)` (where x is any list).

To divide the interval between two given points a and b into n equal parts, the size of the steps from one sub-division point to the next is computed from

$$s =: (\text{diff } a,b) \% n$$

and then adding n of these steps to a

$$p =: a + s * i.n+1$$

Assigning particular values to a, b and n and executing the above two lines will produce the required sub-division points and store them in the variable p .

At right are shown a script file that is stored with incomplete lines for the input variables a, b, n ; the same script file with the input variables assigned values and the Command Window after that script is run in display mode.

```

J:\subdiva.js [1] C:\discrete_book\book\subdiva.js
a=:
b=:
n=:
s=: (b-a) % n
p=: a + s * i. n+1

J:\subdiva.js [2] C:\discrete_book\book\subdiva.js
a=: 3.1
b=: 7.5
n=: 8
s=: (b-a) % n
p=: a + s * i. n+1

J:\sub* [15] C:\MATLAB\help\help
load 'C:\discrete_book\book\subdiva.js'
a=: 3.1
b=: 7.5
n=: 8
s=: (b-a) % n
p=: a + s * i. n+1
p
3.1 3.65 4.2 4.75 5.3 5.85 6.4 6.95 7.5

```

Variable p has been created and is displayed by executing its name. How to turn these instructions into a defined function will be discussed in the next session.

0.2 Computer Programs

The algorithms for primitive functions in a computing environment, that is, those functions that are pre-defined in the environment, such as $+$ $-$ $*$ $\%$, are not usually available to users. For example, the J operator $+$: will double the entries in any list to which it is applied but we do not know whether this is done by adding the numbers to themselves or multiplying each number by 2 or just how. The algorithms for some more complicated operators may even be patented or hidden away as proprietary information of the vendor.

User defined functions are those whose algorithms are composed by someone using the primitive functions.

Many computer languages employ English words to indicate various operations, but any symbols would do. While it might be more meaningful

for English speakers to use English words such as **begin** and **end**, for non-English speakers `<` and `>` might be easier to understand and remember. Computer users from other language backgrounds often have to learn a collection of English words that various computer languages use for common operations. No doubt they are grateful for the fact that there are universal mathematical symbols such as `+` and `-` so that they don't also have to memorize words like *plus* and *minus* in order to do arithmetic.

J uses only the symbols on the standard keyboard *i.e.* the basic ASCII⁴ character set to define its primitive functions. **J** code can therefore be pasted into email correspondence and understood without having to express that understanding in English.

All the special symbols on an ASCII keyboard, namely

```
= < > _ + * - % ^ $ ~ | . : ,
; # ! / \ [ ] { } " ' @ \ & ?
```

apart from `_`, represent operations or operation modifiers in **J**.

The number of pre-defined (or primitive) operations is extended by adding either a dot or a colon to each of these symbols. Thus `+.` and `+:` are also symbols for primitive operations.

Furthermore most of the lower case alphabetic letters and some uppercase ones, followed by a dot have special meanings as well. `a.` for example will return the list of available symbols. See the **J** *Help* menu under *Vocabulary* for more information. Note that the *Vocabulary* page is a webpage and that nearly every word is a link to more extensive coverage.

In **J** simply giving names to primitive functions is the simplest way of producing user defined functions, but combinations of functions are required for any useful work. For example, we defined the arithmetic combination of the functions `+/` (sum) and `#` (length) in `+/ % #` to give an algorithm for computing the mean of a list of numbers.

On occasions it may prove useful and efficient to carry intermediate variables through an algorithm, as in the example of sub-dividing an interval. The algorithm for this function consists of two steps:

⁴American Standard Code for Information Interchange

```

s =: (b - a) % n    NB. computing the step size
p =: a + s * i.n+1  NB. adding n steps to a

```

Before we can implement these steps, however, we need to identify the three input numbers. The three numbers can be given as a list but we must be clear about the order in which they are given. Will it be `a,b,n` or `n,a,b` or some other variation?

It will be useful to define the functions

```

first =: {.
last  =: {:

```

so that if we take the order `a,b,n` then we can begin the algorithm by specifying that, if `y` is the input list, then

```

a =: first y
b =: first behead y
n =: last y

```

To begin the explicit definition of a function, called say `subdivs` in `J` we execute the line

```

subdivs =: 3 : 0

```

The 3 indicates that `subdivs` will be a function (there could also be adverbs and conjunctions) and the 0 indicates that the lines of the algorithm will be typed in immediately from the keyboard (they might also be read in from a file). The cursor will then return to the beginning of the next line waiting for the lines of the algorithm, which we then type in. The function definition is concluded by typing a right parenthesis alone on a line. The final definition is given below. Note that the output is not assigned to a variable. The output is the last thing computed. To assign the output to a variable execute, for example,

```

points =: subdivs 3.1 7.5 8

```

Note too that the assignment of values to intermediate variables used in the algorithm is with the operator `=`. rather than `=:`: This is a local assignment within the workspace of the function rather than a global assignment in the Command Workspace.

```

subdivs =: 3 : 0
a=. first y NB. first input the starting point
b=. first behead y NB. second input the end point
n=. last y NB. third input the number of subdivisions
s=. (b - a) % n
a + s * i.n+1
)

```

NB. On the opening line there must be a space on either side of the colon.

The fact that **J** is a symbolic language can make it difficult to read because reading symbols demands an intimacy with the meanings of those symbols. For that reason some people find it hard to understand written mathematics just as others find it hard to read music. However, every written language is symbolic. It is just that we have become so familiar with the alphabet that we use from early childhood that we forget about the symbolism and concentrate only the meaning. Faced with Cyrillic, Greek, Arabic or Urdu or some script that we are not familiar with, we are totally lost. Words such as **first**, **last**, **behead**, **curtail** etc. may well make it easier to read **J** code.

0.3 Functions and other Relationships

In booting the computer, in downloading software, in starting up **J** we have executed certain rather complicated functions. Sometimes these complicated functions return either a 1 or a 0 indicating whether or not the goal was successfully achieved or not. For example if a file was saved as required a function might return 1, if not, then it would return 0. Functions that return either 1 or 0 are called *Boolean functions*

The simplest relationships between numerical quantities are whether they are equal or not, or whether one is larger or smaller than another.

0.3.1 Exercises with programs/functions

- **The comparisons** = > < >: :<

Using the function `subdivs` defined in the previous section divide interval 0, 4 into 10 equal sub-divisions. Store the subdivision points in variable `x`.

Now compute the corresponding values of $(x^2) + (.3*x) + 2$ and store these values in the variable `y`.

Test whether any of these is 0 by executing `y = 0`

The result of this test is a list of 1s and 0s indicating whether the corresponding `y` value is, or is not, zero. A list with only 1s and 0s in it is called a Boolean list.

If we assign the Boolean list $y = 0$ to a variable, called b , with

```
b =: y = 0
```

then the corresponding x value can be obtained by using the operator $\#$ with a left and right input as in

```
b # x
```

The locations of the y values that are negative are found from

```
neg =: y < 0
```

and the non-negative values ($y \geq 0$) from pos =: y
>: 0

We can compare these two Boolean lists by putting them in the rows of a matrix with

```
pos ,: neg
```

This highlights the fact that there are two sign changes. We can highlight the values by executing

```
(pos*y) ,: neg*y
```

and the corresponding x with

```
(pos*x) ,: neg*x
```

The y value corresponding to an x value of 2 is 0. In mathematical jargon, one says that $x = 2$ is a *zero* of the quadratic $x^2 - 3x + 2$. There is another zero of this quadratic somewhere between 0.8 and 1.2 and we could narrow down a search for it by executing

```
x =: subdivs 0.8 1.2 10
```

```
y =: (x^2) + (.3*x) + 2
```

It turns out we actually hit on a zero again at $(y=0)\#x$.

Using this technique, or otherwise: A rectangular block 20 metres wide is subdivided off a square allotment leaving a block of 861 square metres. What are the dimensions of the block that remains?

- **Max and min** $> .$ $. <$

While $5 <: 2.24 \wedge 2$ tests whether the square of 2.24 is greater than or equal to 5, the operation

```
5 <. 2.24 ^ 2
```

returns the smaller of these two numbers. The operation

```
5 >. 2.24 ^ 2
```

returns the larger of the two numbers.

Show that

```
5 >. (2.24 ^ 2) >. 5.1
```

returns the largest of the three numbers, 5, 2.24^2 and 5.1.

The largest element in a list of numbers can therefore be found by applying the function

```
max =: >. /
```

Define a function, `min`, to find the smallest element in a list.

Find the maximum and minimum yearly rainfall totals for Stoney Creek over the period 1951-1990 with

```
(min , max) Rain
```

- **Function Tables**

When there are left and a right list inputs to an operator like `*/` the operator (`*`) is applied between every element on the left and every element on the right to form a tabular output which is a function (multiplication) table for the operation. With

```
L =: i.11
```

try

<code>L +/ L</code>	<code>L -/ L</code>	
<code>L */ L</code>	<code>L =/ L</code>	
<code>L ^ L</code>	<code>x: L ^ L</code>	
<code>L % L</code>	<code>x: L %/ L</code>	Note effect of <code>x:</code>
<code>L >/ L</code>	<code>L >:/ L</code>	
<code>L >./ L</code>	<code>L <./ L</code>	
<code>-.L </ L</code>	<code>-.L =/ L</code>	Note effect of <code>-.</code>
<code>-.L +/ L</code>	<code>(L+5) -.L</code>	

- **The n^{th} prime number**

`J` provides the primitive `p:` to return the y^{th} prime number. For example, the 3rd prime is returned by

```
p: 3
```

and the millionth prime is given by

```
p: <:1e6.
```

However, while this function will return the 105097564th prime it will not return the 105097565th prime.

Show that `p: i.20` gives a list of the first 20 prime numbers. Create a 25 by 15 table of the first 375 primes.

How many sets of twin primes (primes whose difference is 2) are there in this set? Construct a two column table of these twin primes.