An Introduction to Octave for High School and University Students

Second Edition

Roger Herz-Fischler

Mzinhigan Publishing

ᒪᓯᓇᐃᑲᐣ

An Introduction to Octave for High School and University Students

Second Edition

Copyright © Roger Herz-Fischler 2016

Published 2016 by
Mzinhigan Publishing
340 Second Avenue, Ottawa, Ontario, Canada, K1S 2J2
e-mail: `mzinhigan@herz-fischler.ca`
web site: `http://web.ncf.ca/en493`

"Mzinhigan" is the word for "book" in the Odawa (Ottawa) dialect of Ojibwa. The logo is "Mzinhigan" written in the related Algonquin language of Cree—as spoken on the west coast of James Bay—using Cree syllabics.

Library and Archives Canada Cataloguing in Publication

ISBN 978-0-9693002-9-8

Due to drastic cutbacks at Library and Archives Canada, the classification information was not available at the time of printing.

The following appears at the startup of Octave:

Additional information about Octave is available at `http://www.octave.org`

Preface

While teaching engineering, computer science and mathematics courses at the university level, I wrote *A Guide to Matlab* for the use of my students. In the period since my retirement all the serious Linux distributions[1] have included Octave, which is open-source and therefore free. These Linux distributions, which are also free, are now all "live" (i.e. they can be run directly from a DVD), available on the web and easy to install, even allowing for a dual boot on a Windows machine.

Octave and Matlab are what I would call "application languages" in the sense that they are both oriented towards applications involving mathematics. They are ideally suited for working with arrays of numbers and because of this many mathematical and logical situations can be programmed in a very concise manner. What other languages can only do via "loops"—a feature which is of course also incorporated in Octave and Matlab—can often be evaluated by a short one-line statement which corresponds to what one would write mathematically. So instead of just writing programs, one also *thinks mathematically*. This feature alone makes them relatively easy to learn and ideal for high-school and university students.

It was with all this in mind that I decided to write an introduction to Octave that would be more suitable for students in the upper grades of high shool and for those beginning university. I have limited the topics to those that I think will prove most useful and have limited the number of new topics in each section. Each section—except for the introductory section 01 and the intrinsically long section 07 on graphing—is only two pages long with each section dealing with just one topic.

Both Octave and Matlab are widely used in engineering and science in universities as well as in professional settings. They have many built-in, easy to use functions, which can be applied to a variety of situations, some of which will be illustrated in this introduction. Octave commands have been made compatible with those of Matlab so that students who will be using Matlab later in their studies can make the transition without any additional steps in the learning process. At a very advanced level, there are packages in one which do not appear in the other, but this is not of concern to us at the introductory level.

More advanced notions are described in my *A Guide to Octave and Matlab*. The latter, as well as revised versions of this *Introduction to Octave* are available—and may be freely distributed—on my web site:

`http://web.ncf.ca/en493`

The web site also has the iso file for **Student Linux** which is pre-configured and includes Octave and other software,.

I may be contacted at:

`student_linux@herz-fischler.ca`

1. Octave is also available for Windows, Apple etc, See:
   `http://www.gnu.org/software/octave/download.htm`

Table of Contents

A ★, here and in the text, indicates a special topic. As indicated by the double star, section 18 involves a more complicated situation.

Section 01: A sample Octave session

Conventions

Octave commands and functions (including user-defined functions) are given in boldface.
File names are given in italics.
`Variables and verbatim texts are given in teletype.`

———————-

A summary of the most useful commands

1. To start Octave:
   Type **octave** at the command line [lower case “o”]
2. To quit Octave:
   Type **quit**
3. To save your work in a diary file *project_01.dia*:
   Open a "diary" using the **diary command**: **diary** *project_01.dia*. This is saved in the directory from which you launched Octave.
   [the extension *.dia* helps you identify your diary files, start your numbering with *01*, not *1* so that your file *project_10.dia* does not appear before file *project_2.dia* in the computer “dictionary” ordering of your files.]
   To toggle the diary: **diary off**, **diary on**.
   To end the diary session: **diary off**.

———————-

A very strong suggestion

In principle, you can do everything without a diary. In practice however, especially when you are learning or when you run a program more than once, or if you run several programs, you are going to spend a lot of frustrating time figuring out what went wrong, what you want to change etc. Diaries permit you to direct the output of different trials or programs to different files. Those files that you want to retain can be easily edited. Do not worry about having a pretty output. When everything is working, use a text editor or word processor to put things in the form that you wish.

———————-

4. To list the quantities and variables that you have created:
   Type **whos** [note the “s”]
5. To clear the work space:
   Type **clear**
   To clear just some variables: type **clear** `a, A`
6. To clear the screen [not the work space!]:
   Type **clc** [short for “clear command”]

7. To list the files in the current directory:
   ls [lists all files]
   ls *.dia [lists all dia (diary) files]
   ls *.m [lists all m (function, program and data) files]

Notes and warnings

a. Octave is case sensitive; thus variables `A` and `a` are not the same.

b. To help you distinguish vectors and scalars (section 04) from matrices (section 08), it is suggested that you use lower case (`a2`, `vect_1`) for scalars and vectors and uppercase (`A3`, `STORE_4`) for matrices.

c. Variable names can be any length and may contain an underbar e.g. a_1. Do not use dashes, e.g. `a-1`, because Octave will think that you are subtracting. Do not write `1_a` (i.e. don't start the name with a number) as Octave thinks that you are dealing with a number.

d. Do not use the name of an Octave function for your variables because Octave will think that you are trying to evaluate the function and forgot the ( ). Thus since **sum ( )** is a built in Octave function, you can not use it as a variable name. A very common error is to write something similar to:

```
> sum = 2 + 5
```

If you next use the Octave function **sum( )** you will find that it does not work!

```
> b = 2 + 5
> sum(b)
```

Octave responds with an error message: error: A(I): index out of bounds; value 7 out of bound 1

```
????  error:  A(I): index out of bounds; value 7 out of bound 1
```

Instead of calling your variable "`sum`", call it "`sum1`" or "`sum_1`" or "`total`" etc.

Similarly do not use `prod` for product, or mean for the mean (average).

e. Octave functions are all lowercase and use parentheses. The name usually gives a good idea of what the function does.

f. Octave is often able to interpret statements that are incorrect in the sense that they do not stand for what you wanted them to stand for. In particular you can often get away without using parentheses, but not only may the answer be different from what you had intended (see II.1), but also later you won't be able to understand what you have written. So *USE PARENTHESES*. Always test programs and defined functions on examples for which you can do a pencil and paper or calculator verification.

g. There are many “shortcuts” in Octave in the sense that you can combine statements or leave out certain things. Other calculations can be done with special, more involved, commands. None of these are mentioned in this introduction. *Do not try to be a hero*; split up you statements into several commands that you know will work and which you can verify later on if needed. The time that you save in not typing a few extra symbols, you will more than make up at the debugging stage.

h. If you quit Octave and then return to Octave and use a diary name that you have previously used, Octave adds the new data to the previous diary file.

A sample session

[N.B. For ease of reading the octave prompt before comments have been erased (the number of the next prompt remains unchanged). Similarly the spacing has been increased between certain sub-portions of the output.]

```
%    first open a diary
octave:1> diary section_01.dia
%    this diary appears in the directory from which
%    Octave was launched.  When the diary is closed
%    you can edit it with a text editor.
octave:2> a = 2
      a = 2
%    put a semicolon ; so that what you typed is not repeated
octave:3> a = 2;
octave:4> b = 3;
octave:5> c = a + b;
%    type ‘‘whos’’ to list the variables that you have created
%    size 1x1 indicates that it is a number
%    vectors will be e.g. 1x5 or 6x1 and matrices e.g. 4x5
octave:6> whos
     Name   Size
     a   1x1
     b   1x1
     c   1x1
%    to find the value of a variable, just type it
octave:7> a
      a = 2
%    multiplication is indicated (for numbers) by *
octave:8> b*c
      ans = 15
%    division is indicated (for numbers) by /
octave:9> b/c
      ans = 0.60000
%
octave:10> d1 = 2+3/5;
octave:11> d1
```

```
      d1 = 2.6000
%    is that the answer that you expected OR
%    did you want to add 2+3 first?
%    Use () to make sure that you obtain the answer that you wanted!
octave:12> d2 = (2+3)/c
      d2 = 1 %    you obtain a different answer with ()
%    powers are indicated (for numbers) by ^
octave:15> 2@^5
      ans = 32
%    make sure you close the diary to save it
octave:16> diary off
```

Try these

i. Open a diary [*power_01.dia*]. [Do not forget to close the diary at the end.]

ii. Compute 2^3^2. Compare with 2^(3^2) and (2^3)^2.

iii. Use a text editor to view and edit the diary.

Section 02: Common functions

Trigonometric functions

sin(a**)**, **cos(**a**)**, **tan(**a**)**. [a is measured in radians, multiply by $\frac{180}{\pi}$ for degrees]

Logarithmic functions

The "natural logarithm", usually written ln(a): **log(**a**)**
The logarithm to the base 10: **log10(**a**)**

pi, e, complex numbers

pi
e: the base of the "natural logarithm"
i: $\sqrt{-1}$

```
octave:1> pi
      ans = 3.1416
%
%    e, the base of the ‘‘natural logarithm’’
octave:1>e
      ans = 2.7183
%
%    always use ( ), even when not strictly necessary
octave:2> (e)^(2)
      ans = 7.3891
%
%    square root of a negative number
octave:3> sqrt(-5)
      ans = 0.00000 + 2.23607i
%
%    cube of a complex number
octave:5> (2 -5i)^(3)
      ans = -142 + 65i
```

Displaying the answer

Octave stores values to a very high degree of accuracy. If you want to see the answer to 15 places, use **format long**.

```
octave:3> format long
octave:4> pi
      ans = 3.14159265358979
```

To show your answer in "scientific" ("floating point") notation use:
format short e
or
format long e

To show the answer to 2 decimal places, use:
format bank [money in dollars and cents is shown to 2 decimal places]

```
octave:6> format short e
octave:7> 1/7
      ans = 1.4286e-01
%
octave:8> format long e
octave:9> 1/7
      ans = 1.42857142857143e-01
%
octave:10> format bank
octave:11> 1/7
      ans = 0.14
```

Absolute value, rounding the answer etc.

absolute value: **abs**(a)
round to the nearest integer: **round**(a)
round downwards: **floor**(a)   [= “greatest integer function”]
round upwards: **ceil**(a)
round towards 0 : **fix**(a)

The following function is not built-in; we will create it in the next section.
decimal part: **decimal**( )

Octave has many more built-in functions, and we will see some of these when we talk about vectors and matrices.

Try these

1. Find $\log_{10}\left(10^{271.6}\right)$.
2. Compute $10^{0.5}$. Compare the answer with the Octave square root command: **sqrt**(10).
3. Evaluate the numbers 2.71 and -2.71, first using the definitions of **abs**, **round**, **floor**, **ceil**, **fix** and **sign** and then by using Octave.

Section 03: Defining your own functions

To create a function you create an “m-file” whose name—without the extension—is that of the function. Thus to create **my_function_01 (x)** we create a file *my_function_01.m*.

Note the following steps:

1. start with the word **function**
2. assign a symbol for the dependent variable; y is a good choice
3. the name of the m-file must be the same as the name of the function
4. assign a symbol for the independent variable; x is a good choice
5. a % indicates a comment and is ignored by Octave; use comments so that you can more easily retrace what you have done
6. end with the *single* word **endfunction**

```
% [my_function_01.m]
%    use underline (_), not a dash (-), or Octave will think that
%    you are subtracting!
%
%    this function first adds 5 to the given value, then raises
%    to the 4th power, and finally divides by 17.
%
function y = my_function_01(x)
%    now state how to evaluate y from x
%    do it in steps to make checking easier
%    you can use any variable symbol---except the ones in
%        the function line---for the intermediate steps
%    indent the intermediate statements for ease of reading
%    use () to avoid errors
%    use ; to suppress printing of the intermediate steps
%
  z1 = x+5;
  z2 = (z1)^4;
  y = (z2)/17;
endfunction     % one word, not end function
```

Now we want to evaluate the function:

```
%    to list all the ‘‘m-files’’ in the current directory
octave:1> ls *.m
      my_function_01.m
%
%    use any symbol for the value of the function
octave:2> w1 = my_function_01(0)
      w1 = 36.765
%    check by hand
octave:3> (0+5)^(4/17)
      ans = 36.765
```

```
%    for just one value you do not need to use a variable name
octave:4> my_function_01(sqrt(2))
      ans = 99.569
```

We now create the function **decimal** that was listed in Section 2. This is an example of building up a function from those which are built-in or previously created.

```
% [decimal.m]
%
%    we want to find the decimal part of number as
%    a positive number
%
%    first use the absolute value function abs( )
%    the function floor( ) gives the integer just below the number
%
function y = decimal(x)
  y1 = abs(x);
  y = (x1 - floor(x1));
endfunction    % one word
```

Try these

1. Create a function **my_function_02** which will evaluate $[2 - \sin(x + \pi)]^3$. [Do not forget the semi-colon, to suppress printing.]
2. Check the function **decimal** with the values π and $-\pi$.
3. Create a function **my_function_03** which will first evaluate $[2 - \sin(x + \pi)]^3$ and then find the decimal part of the answer. Do this by calling up **my_function_02** in the m-file for **my_function_03**. [note: you have to do this in two steps; first call **my_function_02**, then use the function **decimal**.]

Section 04: Vectors – basic concepts

1. `a` = [x1 x2 x3 . . .] is called a *vector*. `x1 x2 x3` ... are called the *elements* of the vector. In Octave a vector is simply an array of numbers enclosed in square brackets,

```
octave:1> a = [-3 7 0]; % suppress printing with ;
```

2. To display a vector, either at the Octave prompt or inside an “m-file”, use the Octave function **disp**(`a`) . At the Octave prompt you can also simply type `a` .

```
octave:2> disp(a)
      -3 7 0
octave:3> a
      a = -3 7 0
```

3. To add, multiply, or divide `a` by 5, just write `a`+5, 5*****`a`, `a`/5 .

```
octave:3> a1 = a + 5;
octave:4> disp(a1)
      2 12 5
octave:5> a2 = 5*a;
octave:6> disp(a2)
      -14 35 0
octave:7> a3 = a/5;
octave:8> disp(a3)
      -0.60000 1.40000 0.00000
```

4. Suppose that `b` = [y1 y2 y3 . . .] is another vector of the *same length* as `a`. We can do element by element (`x1+y1, x2+y2`...) *addition* of `b` to `a` by writing `a+b`.

```
octave:9> disp(a)
      -3 7 0
octave:10> b = [2 -1 5];
octave:11> disp(b)of the vector.
      2 -1 5
octave:12 c1 = a+b;
octave:13 disp (c1)
      -1 6 5
```

5. We can *multiply* each element of `a` by the corresponding element of `b` (`x1*y1, x2*y2`...) by using the *dot* notation: (`a`)**.***(`b`). Use parentheses even in simple examples.

```
octave:14> c2 = a*b;
error:  operator *:  nonconformant arguments (op1 is 1x3, op2 is 1x3)
% eh?
% you forgot the dot (welcome to the club!)
% Octave gives you another chance with the same prompt number
octave:14> c2 = (a).*(b);
octave:15> disp(c2)
      -6 -7 0
```

6. We can *divide* each element of `a` by the corresponding element of `b` (`x1/y1, x2/y2`...) by using *dot* notation: (a)./ (b). Use parentheses even in simple examples.

```
octave:16> c3 = (a)./(b);
octave:17> disp(c3)
      -1.50000 -7.00000 0.00000
```

7. To raise each element of `a` to the 3rd power use the *dot* notation: (a).^ . Use parentheses even in simple examples.

```
octave:18> c4 = (a).^(3)
octave:19> disp (a6)
      -27 343 0
```

Try these

Open a diary for the following operations. At the end, edit out any mistakes etc..

i. Add `vector_2` above to `vector_1`.

ii. Subtract `vector_2` from `vector_1`.

iii. Use Octave to multiply 1 by 6, 2 by 7 ... 5 by 10.

iv. Use Octave to divide 1 by 6, 2 by 7 ... 5 by 10.

v. Use Octave to raise each of the integers 1, 2 ... 5 to the 9^{th} power.

The dot notation also allows us to raise each element of one vector by the corresponding element of another vector:

vi. Use Octave to raise the integers 1, 2 ... 5 to the powers 2, 0, 1, -4 and $\frac{2}{3}$ respectively.

vii. Pretend that, as above, you multiply `vector_1` by `vector_2`, but that you *forget* the dot. What happens? [This type of multiplication is reserved for *matrices* of "matching" dimensions.]

Section 05: Functions of vectors; data analysis; statistics, part 1

Applying functions to vectors

All the functions of Section 2 can be applied directly to a vector a; e.g. the “assignment” b = **tan**(a) will produce a new vector b whose elements are the tangents of the elements of a. This will be particularly important when we graph functions.

```
octave:1> a = [pi/4 pi/2 pi];
octave:2> b = tan(a);
octave:3> disp(b)
      1.0000e+00 1.6331e+16 -1.2246e-16
```

The second value is “infinity”, whereas the third value is “zero”.

Data analysis

The following functions of vectors are very useful when examining large sets of data:

i. The length of a vector a: **length**(a)
ii. The sum of the elements of a: **sum**(a)
iii. The largest element of a: **max**(a))
iv. The smallest element of a: **min**(a)
v. To sort the elements of a vector a from smallest to largest: **sort**(a)
vi. To sort the elements of a vector a from largest to smallest: first use **sort** and then use **fliplr** [lr = left to right].

[N.B. Note how in the following Octave session **disp** is applied directly]

```
octave:1> a = [2 -4 0 8 3];
octave:2> disp(length(a))
      5
octave:3> disp(max(a))
      8
octave:4> disp(min(a))
      -4
%    we want to use the sorted vector later, so we ‘‘assign’’ it
%    to the vector b
octave:5> b = sort(a);
octave:6> disp(b)
      -4 0 2 3 8
octave:7> c = fliplr(b);
octave:8> disp(c)
      8 3 2 0 -4
% we could also combine the two functions in one step
octave:9> fliplr(sort(a))    % note the two sets of ( )
      ans = 8 3 2 0 -4
```

Statistics

The following Octave functions are used in basic statistics. Octave also has functions that are used in advanced statistics.

i. The mean (average) of a data set a: **mean**(a)
ii. The median (middle value) of a data set a: **median**(a)
iii. The standard deviation (a measure of data dispersion around the mean) of a data set a: **std**(a)
iv. Octave also has many other statistical functions, including:

mode(a) (most frequently occuring value)

var(a) (variance)

range(a) (the difference between the largest and smallest values)

quantile(a); this gives the 25%, 50%,75% quantile values

quantile(a, [0 : .1 : 1]) gives the 10% quantile values

```
%    ‘‘mean’’ is a ‘‘reserved name’’,
%    so DO NOT USE ‘‘mean’’ as a variable name!
%    Do NOT WRITE ‘‘mean-a’’ (Octave interprets as subtraction)
octave:10> mean_a = mean(a);    % use underscore
octave:11> disp(mean_a)
      1.8000
%    the mean is just the sum divided by the number of elements
octave:12> sum_a = sum(a);
octave:13> length_a = length(a);
octave:14> average_a = (sum_a)/(length_a);
octave:15> disp(average_a)
      1.8000    % the same answer
%    the median is the middle value
octave:16> median_a = median(a);
octave:17> disp(median_a);
      2
octave:18> standard_deviation_a = std(a);
octave:19> disp(standard_deviation_a)
      4.3818
```

Try these

1. Consider the following set of values: $\{15.21, -0.384, -83.1, 6.04\}$

   First apply the function **decimal** of Section 03, then multipy the results by 10, then **round** these decimals parts (Section 02) and finally arrange the resulting *integers* in decreasing order.
2. Obtain the heights of several of your friends to the nearest cm. Compute the Pick a number – part 1 mean, median and standard deviation of these heights.

Section 06: Vectors of indices; products of elements

Vectors of indices

When writing programs, for plotting and for other purposes, we want to have a concise way of writing sets of equally spaced indices or values:

1. To obtain the sequence {-1, 0, 1, 2, 3}, use the colon notation; `k1 = [-1:4]` (a jump of +1 is implicit if you do not indicate otherwise).
2. To obtain the sequence {-1, 1, 3, 5, 7} (increase by 2 each time), use the double colon notation; `k2` = [-1 **:** 2 **:** 7].
3. To obtain the *decreasing* sequence {-1, -3, -5, -7. -9} (decrease by 2 each time), use the double colon notation; `k3` = [-1 **:** -2 **:** -9].
4. To increment values by the value .1 and obtain the sequence {0, .1, .2 ... 1}; write `x` = [0 **:** .1 **:** 1].

[N.B. For clarity, a space has been left before and after the colon, but there is no need to do this when using Octave. The symbols `n, n1, k` ... are good ones to use for sets of indices.]

```
octave:1> k1 = [-1 : 4];
octave:2> disp(k1)
      -1  0  1  2  3  4
octave:3> k2 = [-1 : 2 : 7];
octave:4> disp(k2)
      -1  1  3  5  7
octave:5> k3 = [-1 : -2 : -9];
octave:6> disp(k3)
      -1  -3  -5  -7  -9
octave:7> x = [0 : .1 : 1];
%    normally you would not want to print out the huge set of val-
ues
octave:2> disp(x)
    Columns 1 through 11:
    0.00000 0.10000 0.20000 0.30000 0.40000 0.50000
    0.60000 0.70000 0.80000 0.90000 1.00000
```

Products of elements, factorials

The Octave function **prod( )** will multiply the elements of a together. The following Octave session illustrates its use.

We start with the index set: [5 : -1 : 1] = {5 4 3 2 1}. Next we take **prod( )**, which is just 5 x 4 x 3 x 2 x 1. We recognize this product as 5 factorial (5**!**), which is the number of ways of placing 5 *distinct* items in 5 slots. Octave has a function **factorial( )** which will also evaluate this for us.

```
octave:1> n1= [5 : -1 : 1];
octave:2> disp(n)
      5  4  3  2  1
```

```
octave:3> product1 = prod(n);
octave:4> disp(product1)
      120
%    check via the Octave function factorial
octave:5> factorial(5)
      ans = 120
```

Vectors with all of the elements the same

Sometimes it is useful to generate a vector or matrix, all of whose elements are the same. Octave has a function **ones(1 , n)** which will produce a vector all of whose elements are equal to 1. Then all we have to do is mulitply by the constant that we want. Suppose that we want to generate {5 5 5 5 5}:

```
octave:6> a = ones(1,5);
octave:7> disp(a)
      1 1 1 1 1
octave:8> b = 5*a;
octave:9> disp(b)
      5 5 5 5 5
```

Octave also has a function **zeros(1 , n)** which will produce a vector of 0s. In section 12 we will see how to use **zeros( )** to store values that we have generated.

Try these

1. Generate the index vector corresponding to {10, 15, ... 95}. Calculate the product of the elements.
2. Produce a vector of length 7 each of whose elements is equal to $\frac{1}{4}$. Multiply the elements together. Compute the answer in a different way.
3. The Octave function **ones( )** has two variables because it also works with *matrices* (section 20). If we used another first number instead of 1, e.g. **ones(3 , 4)** we would obtain a 3 by 4 *matrix* of 1s. Create a function **ones_vector(n)** of one variable so that one does not have to type in the 1.
4. Each way of ordering objects is called a *permutation*. Octave has a function **perms**(a) which will list all permutations of the elements of a. [If two elements are the same, then each possibility will be repeated twice.] List all permutations of the set {-4 9 0 7}and count visually how many there are. Check using **factorial**.

   ★ With 4 elements we could count by hand, but suppose that we had 23 distinct elements! In this case we would let A = **perms**(a)**;** be the *matrix* of permutations. Then we would use the function **size**(A) which will give the number of rows and columns of A. Do the calculations for a = [1 : 23]. [Make sure that you put a semicolon so that A does not print out!] Figure out how many columns A will have *before* you do the calculation.

Section 06A*: Pick a number, part 1

Suppose that 15 people are asked to pick a number from {1, 2, ... 100}. We ask how many ways can *different* numbers be picked by the 15 people. The first person has 100 choices, the second now has only 99 choices if the number that the second person picks is to be different, the third 98 choices etc. The 15th person will have $100 - 15 + 1 = 86$ choices (to see that you have to add 1, check with 2—the second person—instead of 15). So the number of ways is:

$$100 \cdot 99 \cdot 98 \ldots \cdot 86 = \textbf{prod}\,([100:\text{-}1:86])$$

```
octave:1> n2 = [100 : -1 : 86];
octave:2> prod2 =prod(n2);
octave:3> disp(prod2)
      3.3128e+29
%    a large number of choices indeed!
```

Note that if instead of decreasing the numbers by 1 each time we used 100 each time we would have 100^15 = 10^30.

Now if, instead of asking *how many ways*, we ask for the *probability* that the 15 people all pick a *different number*.

The first person—being the first!—has 100 choices out of 100 of picking a different number from all the the preceeding persons and so their probability is $\frac{100}{100}$. The second person has 99 choices out of 100, so the probability is $\frac{99}{100}$ etc.. So we have:

$$\text{probability the 15 people all pick a \textit{different number}} = \frac{100}{100} \cdot \frac{99}{100} \cdot \frac{98}{100} \cdots \frac{86}{100}$$

To evaluate this we first form the vector `[100 : -1 : 86]`, then we multiply the vector by $\frac{1}{100}$ and finally we multiply the elements[1]:

```
octave:4> n2 = [100 : -1  : 86];
octave:5> n3 = (1/100)*n2
octave:6> prob_different = prod(n3)
      prob_different = 0.33128
```

The next step is to find the probability that *at least* two people pick the same number. To do this *directly* is very complicated because maybe three people picked the same number or perhaps 5 picked one number, 2 others picked another number, 3 others picked another etc.. There are just too many possibilities to count them. Fortunately there is an *indirect* way of calculating the probability. All we have to do is "think heads":

tails is the opposite of heads

so:

probability of tails = 1 – probability of heads

In our case we think of heads as being, "all 15 pick a *different* number" and the opposite is tails, namely that "*at least* two people pick the same number". Thus:

probability that *at least* two people pick the same number = 1 – 0.33128 = 0.66872

The birthday problem

If we assume that a given person has 1 out of 365 chances of their birth falling on a *given* date[2] then finding the probability that *at least* two people in a group of k have the same birthday is *exactly* the same problem as above, with 365 replacing 100 and k replacing 15. Here is a four line Octave function **birthday( )** which will enable us to do the calculations (functions were discussed in section 03):

```
% [birthday.m]
%
function p=birthday(k);
  a = [365 : -1 : (365-k +1)];
  b = (1/365)*a;
  c = prod(b);
  p = 1-c;
endfunction
```

We evaluate the function for groups of 10, 20, 30, 40, 50, 60 people:

```
octave:7> p10 = birthday(10);
      0.117
```

The other probabilities are:

p20 = 0.411; p30 = 0.706; p40 = 0.891; p50 = 0.970; p60 = 0.994.

We see that the probabilities climb very quickly and by the time that there are 60 people in a group, it is over 99 percent certain that *at least* two people in the group will have the same birthday; a surprising result indeed!

These probabilities will be graphed in section 07.

Notes

1. This gives a greater accuracy than if we took the huge number 3.3128e+29 and divided it by 100^{15}.
2. We neglect the possibility of February 29 in a leap year. Statistically, births are not distributed equally throughout the year.

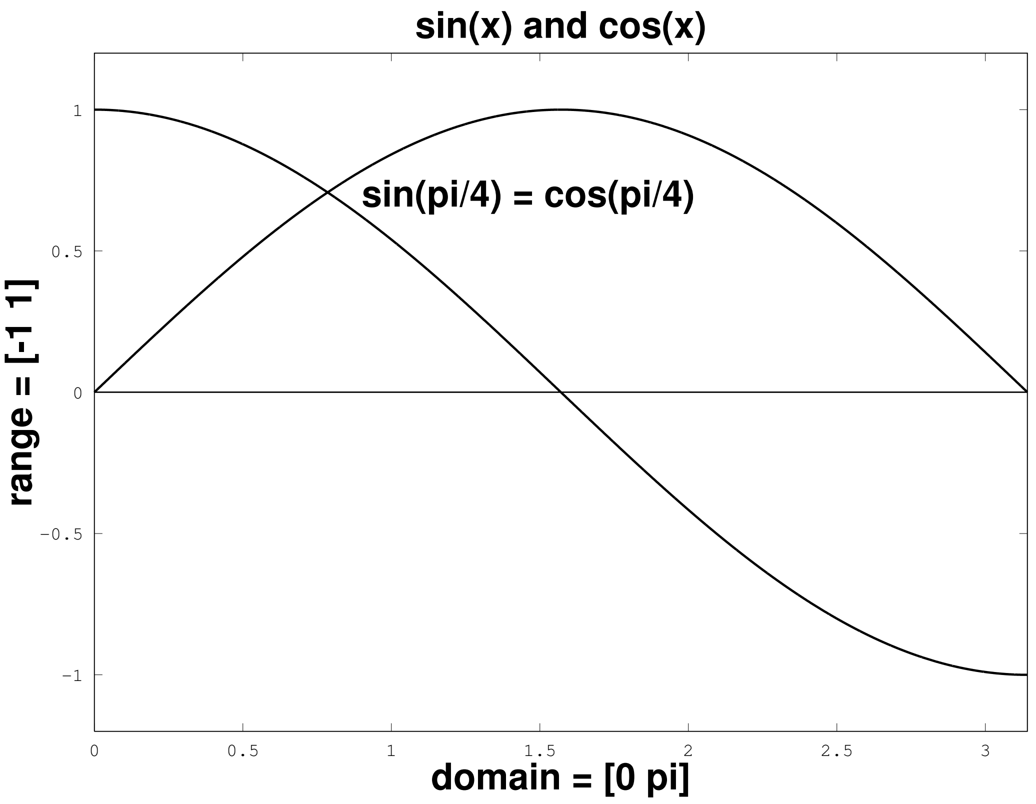
Try these

1. Modify **birthday( )** and create a function **pick_a_number(k,n)** of two variables which calculates the probability that if k people pick a number from {1,th 2 ... n} then at *least* two people pick the same number. Check that you obtain the same answer as above for k = 15 and n = 100. Evaluate for k = 10, 20 ... 99.
2. What can you say about the answer to (1) in case k = 100 and n = 100? Do the evaluation using **pick_a_number(100, 100)**.
3. What will the probability be if k = 101 and n = 100? [The answer is known as the "pigeon hole principle".]

Section 07: Graphing

Octave uses Gnuplot to do the graphing. One could learn how to use Gnuplot directly, but it is better to learn Octave and let it do the work.

The following example, in which we plot y = sin(x) and y = cos(x) on the interval $[0, \pi]$, shows all the basic steps involved in plotting, labelling, and printing. There are many options, e.g. using dashed lines, label fonts and sizes, but one should first learn the basics. You should start with one function, then add a second function, then labels etc. You can play with line widths etc. to suit your taste.



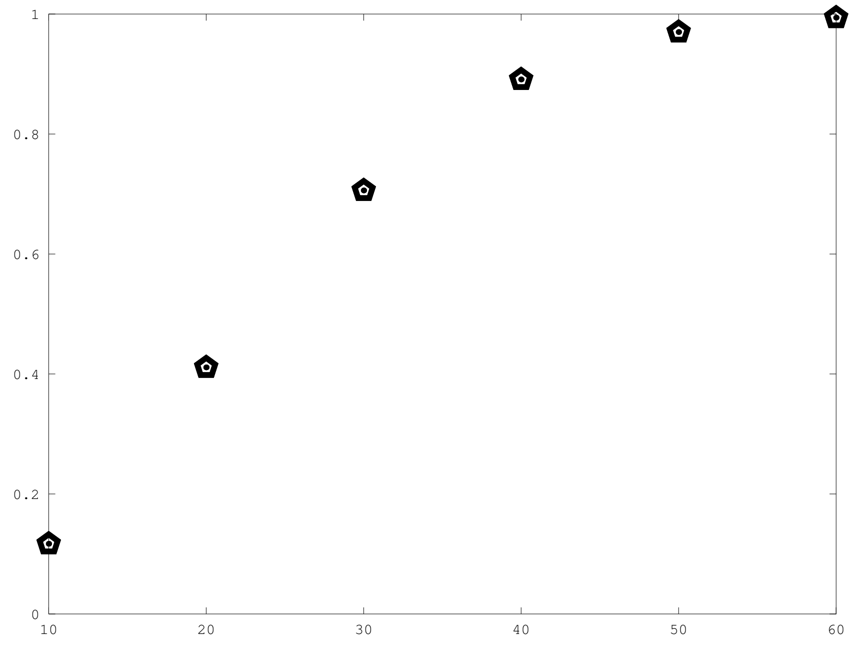
```
%    first clear all preceeding graphs
octave:1> clf % = ‘‘clear function’’
%
%    tell Octave the range of values of interest
%    the sine and cosine go from -1 to 1, but the graph
%    will be nicer if we go slightly below and above
%
%    draw the Octave axes, which show the values
octave:2> axis([0 pi -1.2 1.2])
%
%    keep the Octave axes and all succeeding plots
octave:3> hold
%
```

```
%    Draw the real x-axis from (0,0) to (0,pi)
%    a = [0 pi] gives the x coordinates of the x-axis
%    b = [0 0] gives the y coordinates of the x-axis
%
octave:4> a = [0 pi]; % NOT [0:.001:pi]
octave:5> b = [0 0];
%                     FIRST, SECOND
%    ALWAYS: plot( x-values, y-values)
%    increase the line thickness, by writing 'linewidth', 3
octave:6> plot(a, b , 'linewidth', 3)
%
%    now we are ready to plot the two graphs
%    give the x-values for the plot; use increments of .005
octave:6> x = [0 : .005 : pi];    % could use .001
%
%    first graph y = sin(x); use y1 for the name of the vector
octave:7> y1 = sin(x);
%
%    we plot the y-values (y1) against the x-values (x)
%    from now on graphs are thicker using 'linewidth', 5
octave:8> plot(x, y1, 'linewidth', 5)
%
% the same font descriptions were used)
%    second graph y = cos(x); use y2 for the name of the vector
octave:9> y2 = cos(x);
octave:10> plot(x, y2, 'linewidth', 5)
%
%    now add a title
%    the text is enclosed in apostrophes, not quotation marks
%    from now on we use:    'fontweight', "bold", 'fontsize', 20
octave:11> title('sin(x) and cos(x)', 'fontweight', "bold",
       'fontsize', 20)    % if the line is too long, push return
%
%    labels for the x and y axes, note ``xlabel'', not ``x-label''
octave:12> xlabel('domain = [0 , pi]')
octave:13> ylabel('range = [-1 , 1]')
%
%    include a text to indicate the point of intersection
%    the text is placed at (.9 , .7) [obtained by trial and error]
octave:14> text(.9 , .7, 'sin(pi/4) = cos(pi/4)')
%
%    save the graph in three formats:
%    1. png: image, better than jpg
%    2. pdf: for immediate printing
%    3. eps: encapsulated postscript; for importing into a Latex file
%    specify the full name of the output file
octave:15> print -dpng graph1.png
octave:16> print -dpdf graph1.pdf
octave:17> print -deps graph1.eps
%    check that the graphs are there using ls = ‘‘list’’
octave:18> ls *.png
      graph1.png    % yes, its there!
```

Above we plotted a continuous function, but the same holds for discrete data. The one difference is that to indicate that we just want to plot the points using a pentagon ("p") of large size 12 we add **"p", "markersize",12** to the plot command. [Other options are "o" , "+" , "x" , "*", "h" (hexagon), "s" (square),"^" (triangle).]

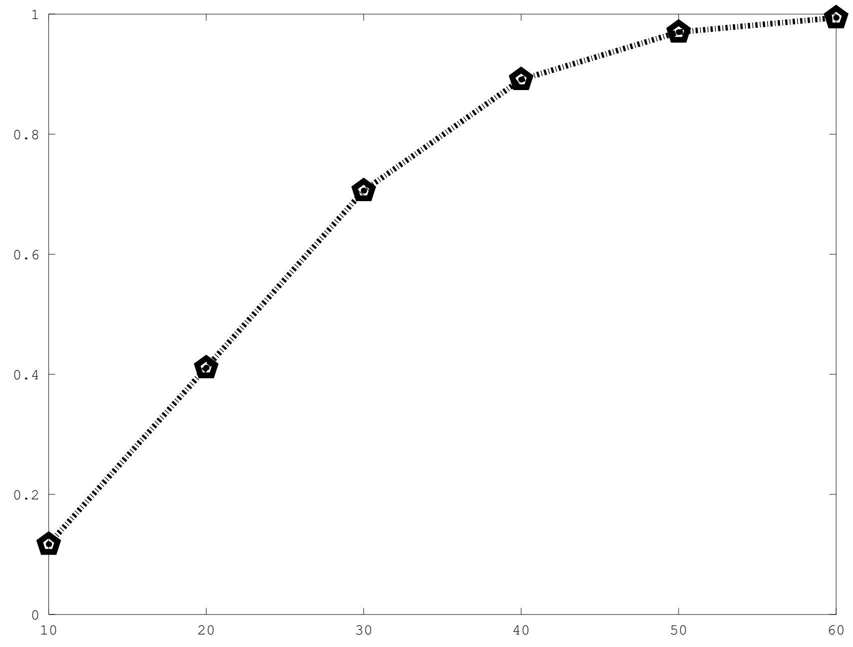
To illustrate this we use the birthday probabilities from section 06A:

```
octave:19> indices = [10 : 10 : 60];
octave:20> birthday_prob = [0.117 0.411 0.706 0.891 0.970 0.994];
octave:21> plot(indices, birthday_prob,"p","markersize",12)
```



If in addition to placing the pentagons at the data points we wanted to connect them by a thin dashed line, all we would have to do is **hold** the graph and then do a new plot which does not have the **"p"** and **,"markersize",12** commands. We can make the line alternate between a dash and a dot by using the command **’linestyle, ’-.’**. [Other options are ’--’ , ’:’ , ’-’ (solid).] Lines and curves can also be coloured.

```
octave:22> hold
octave:23> plot(indices , birthday_prob , ”linestyle" , ’-.’)
```



Try these

1. Plot the graphs of $y = x^2$, $y = \sqrt{x}$, $y = x$ on the same axes. Let x vary between -0.5 and 2.5.

   Print the line $y = x$ with dash marks using the command:
   Plot(x,y, **'linestyle', '--'**) [note the apostrophes and the comma between the last two commands.]

   Print the other two curves with a thicker line using the command:
   plot(x, y, **'linewidth', 8**) [note the apostrophes and the comma between the last two commands.]

   Put the labels "x-axis", "y-axis". The two functions are inverse functions of one another and are symmetric about the line $y = x$. Give the title "Inverse functions" to the graph. Label the points of intersection of the two graphs.

2. Consider an isosceles triangle with angles A, B, A (in degrees) and sides a, b, a.

i. Create a function that determines the ratio $\frac{b}{a}$ as a function of B. [suggestion: use the law of sines; you have to convert from degrees to radians.]

ii. Plot the ratio $\frac{b}{a}$ as a function of B, as B varies from 1^o to 179^o. Check the answers at 1^o and 179^o by making a sketch. Why were 0^o and 180^o not included?

iii. Above we used regular pentagons to mark the points. Use (i) to find the ratio of the diagonal of a regular pentagon to its side. [answer: see `G` in section 08.]

Section 08: Roots of a polynomial and zeros of a function

Polynomials

Suppose that we want to find a positive number such that the square is one more than the number. This leads to the polynomial equation:

$$x^2 = 1 + x$$

and we want to find the roots. Since this equation is quadratic, we can use the quadratic formula, but what if we have a cubic or higher order polynomial? Octave has a routine for finding the roots (or an appoximation) of any order polynomial.

The first step is to put all the terms of the polynomial on the left in *decreasing* order of the exponents:

$$x^2 - x - 1 = 0$$

Now we create a vector whose entries are the coefficients of the polynomial:
Then we use the Octave function **roots ( )**:

roots_1 = **roots**(poly_1)

```
octave:1> poly_1 = [1 -1 -1];
octave:2> disp(poly_1)
      1 -1 -1
octave:3> roots_1 = roots(poly_1);
octave:4> disp(roots_1)
      -0.61803
      1.61803
octave:5> G = (1+sqrt(5))/2;
octave:6> disp(G)
      1.6180
```

The calculation of step 5 shows that the desired solution of the quadratic equation is $\frac{1+\sqrt{5}}{2}$. This is the famous (and infamous) "golden number".[1] Other, surprising, methods of finding G will appear in later sections.

Zeros of a function★

A quick sketch shows that for $y \geq 0$ the line: $y = x$ crosses the cosine curve at $x = 0$ and just one other point. To find this point we first define the function my_function4(x) = $x - \cos(x)$ and what we want to do is find the zeros of $f4(x)$. Octave has a function **fzero( )** which will find this root. So the next step is to write an "m-file" (section 03) which describes the function.

```
function y = my_function_04(x)
  y = x - cos(x);
endfunction
```

To check for errors we evaluate the function at a few points:

```
octave:1> my_function_04(0)
      ans = -1    % correct since cos(0) = -1
```

```
octave:2> my_function_04(pi/2)
      ans = 1.5708 % this is π/2 as it should be
octave:3> pi/2
      ans = 1.5708
```

[As in section 5, we can apply this function to a *vector* of values. This will be done in the next section.]

Because my_function_04(x) is negative at $x = 0$ and positive at $\frac{\pi}{2}$, the zero is someplace between the two.

In general there may be many roots of a function, therefore we have to help Octave by giving a starting point a. So the precise form of the command for **fzero( )** is:

root1 = **fzero(**`'my_function_04'` , a**)**

Note the apostrophes on both sides of the name of the function, which is given without the extension .m.

Since we know that the root lies between 0 and $\frac{\pi}{2}$, we can start with $a = 1$ as a guess:

```
octave:4> root1 = fzero('my_function_04', 1)
      root1 = 0.73909
% check the answer
octave:5> my_function_04(root1)
      ans = 0
```

Try these

1. Find the roots of $x^2 = x + 12$ by first using the quadratic formula and then using **poly**.
3. Find all the roots of $x^3 - 3x - 9x + 5$.
3. Find the square roots of 17 by setting up an equation and then using **poly** .
4. Find all the cube roots of 17. [the answer involves complex numbers as indicated by the symbol i in the solution.]
5. What is the difference between $2 - 2x$ and 10^x when $x = 0$; when $x = 1$? When are they equal?

Cultural note

1. For a mathematical history see my book, *A Mathematical History of Division in Extreme and Mean Ratio*, republished by Dover as *A Mathematical History of the Golden Number*. For the infamous aspect see my book, *The Shape of the Great Pyramid* and the articles on my web page: `http://herz-fischler.ca`.

Section 09: Saving and Loading; Vector logic

Saving the answers

In the last section we defined the function my_function_04(x) and evaluated it at the single point $\frac{\pi}{2}$. As in section 05—where we worked with built-in functions—we can also evaluate this function on a *vector*. Suppose we are interested in the values taken on by this function on the interval $[\frac{\pi}{2}, \pi]$ and wish to find the approximate values of the maximum, minimum and average. To do this we use sub-intervals of length .01 (we could use a much finer grid). We do this just as in section 06; note the semi-colon to avoid printing out all the values:

```
octave:1> x_values = [pi:.01:2*pi];
```

Now we preceed in the same way as sections 05 and 08

```
octave:2> y_values = my_function_04(x_values);
octave:3> max_y_value = max(y_values)
max_y_value = 5.2816
octave:4> min_y_value = min(y_values)
min_y_value = 4.1416
```

Next we use the **save** command to save all the variables to a filed called `example_4.dat`. We then **clear** the work space and use the **load** command to re-obtain the variables. The command **whos** allows us to check (notice how the dimensions are given):

```
octave:5> save example_4.dat
octave:6> clear
octave:7> whos
octave:8> load example_4.dat
octave:9> whos
    max_y_value  1x1
    min_y_value  1x1
    x  1x32
    x_values  1x315
    y_values  1x315
```

Now that we have the data again we can find the mean value:

```
octave:11> mean_y_value = mean(y_values)
mean_y_value = 4.7121
```

Counting the number of elments satisfying a certain condition

Consider the vector `k3` = [-1 -3 -5 -7 -9] of section 06. We want to know how many of the elements are greater than -4. In this case we can see right away that the answer is 2, but imagine that we had measured the heights of 987 people and that we wanted to know how many and what fraction of these 987 people were taller than 1.9 m. Octave logic allows us to do the counting in two concise statements:

1. We make the "assignment" `n3` = [k3 > - 4]. This is just shorthand for the action: "look at each of the elements of `k3` and place a 1 in `n3` if the corresponding element of `k3` is bigger than -4; otherwise put a 0".

```
octave:1> k3 = [ -1 -3 -5 -7 -9];
octave:2> n3 = [k3 > -4];
```

```
octave:3> disp(n3)
      1 1 0 0 0    only -1 and -3 are greater than -4
%    the first two elements satisfy the condition
%
octave:4> l3= length(n3)
octave:5> disp(l3)
      5    % of course, since n3 has the same length as k3
```

2. In order to know how many of the elements of `k3` are greater than -4 all we have to do is count the number of 1s in `n3` and to do this we can just take the sum of the elements of `n3`.

```
octave:6> s3 = sum(n3);
octave:7> disp(s3)
      2    % so 2 elements of k3 are greater than -4
```

3. Finally to find the fraction of elements of `n3` which are greater than -4, we simply divide `s3` by the length of `k3`:

```
octave:8> fract3 = s3/l3;
octave:9> disp(fract3)
      0.40000    % 2 of 5 elements are are greater than -4
```

More logic with vectors ★

4. We can also impose multiple conditions using:
   & = "and"
   | = "or"
   Each condition is written separately inside ( ).

```
octave:11> k3 = [ -1 -3 -5 -7 -9];
%    how many elements of k3 are > -4 and also < -2
octave:12> m3 =[(k3 > -4) & (k3 < -2)];
octave:13> disp(m3)
      0 1 0 0 0 %    only element 2 satisfies both conditions
octave:14> disp(sum(m3)) %    note the double ( )
      1
```

Try these

1. How many elements of the sequence {-5, 1 , -1 , 4, 0} are either less than -2 or strictly positive?
2. "less than or equal" is written **<=**. Repeat (1) with "strictly positive" replaced by "non-negative".
3. ★ Redo steps 11 and 12 if *and* is replaced by *or*.
4. ★ To test for equality Octave uses **==** (two equal signs as distinquished from "assignment" statements such as `n3 = [k3 > -4]`. How many elements of the sequence {4, 3, -2, 4, 1} are exactly equal to 4?
5. ★ "not equal" is written **~=**. How many elements of `k3` are not equal to -7?

Section 10: Matrices; statistics, part 2

In section 04 we discussed the concept of a vector which is simply a row[1] (or 1-dimensional set) of data. A *matrix* extends this idea to rectangular (or 2-dimensional) sets of data. For example:

$$\mathtt{M} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

is called a 2 by 3 matrix, with the first number always referring to the number of rows. It is good practice to designate vectors and variables by lower case letters (e.g. `vector_`, `x`) and matrices by upper case variables (e.g. `M`).

Entering the data for matrices

To enter the data we simply start typing and then, at the point which marks the end of the data for the first line, we put a semi-colon. Note that this is a different use of semi-colons from their use in suppressing printing.

```
%    all the data can go on one line.
%    indicate the separation between the rows by a semi-colon
octave:1> M = [1 2 3; 4 5 6]
    M =
    1 2 3
    4 5 6
```

We could also place the data on two lines. After the first line we need to put three dots to indicate that we are continuing. The same thing would be done if the rows of the matrix were too long to fit on one line.

```
octave:2> M = [1 2 3; ...  % ⇐= three dots
> 4 5 6]    % we continue the input on the next line
    M =
    1 2 3
    4 5 6
```

Building matrices from vectors

Suppose that we have four candidates for a position and we give each of them a battery of three tests, with test 1 being worth a maximum of 30 points, test 2 being worth a maximum of 50 points, and test 3 being worth a maximum of 20 points. We place the scores of each candidate in a vector:

```
octave:1> candidate_1 = [21 34 15];
octave:2> candidate_2 = [29 14 9];
octave:3> candidate_3 = [16 41 17];
octave:4> candidate_4 = [21 32 18];
```

To form a matrix `CANDIDATES` we place the four vectors inside **[ ]** and separate them by semi-colons. Because the names are long we put ... after the third vector (to indicate that we are going to continue) and then go on to the next line:

```
octave:5> CANDIDATES = [candidate_1;
candidate_22; candidate_3; ...
> candidate_4]
% as a check we let Octave print, so there is no ; after the []
%
     CANDIDATES =
     21 34 15
     29 14 9
     16 41 17
     21 32 18
```

Statistics, part 2

Now we want to find the mean and standard deviation on the tests. Since the *rows* of CANDIDATES represent distinct people, whereas the *columns* correspond to distinct tests, we want to find the means and standard deviations of the *columns*. Because of this kind of situation, Octave functions such as **sum**, **max**, **mean** etc., operate on the *columns* of a matrix and not on the rows.

```
octave:6> test_averages = mean(CANDIDATES);
octave:7> disp(test_averages)
       21.750 30.250 14.750
%
octave:8> test_std = std(CANDIDATES);
octave:9> disp(test_std)
       5.3774 11.5000 4.0311
%
octave:10> max_on_each_test = max(CANDIDATES);
octave:11> disp(max_on_each_test)
       29 41 18
```

Note

1. There are also *column* vectors and then one uses the terminology *row* vector instead of just vector.

Try these

1. Create the following vectors using the index method of section 06:
   v_1 = {2, 4 ... 16} ; v_2 = {-3, -6 ... -24}; v_3 = {1, 3 ... 7} .
   Check, using **length( )**, that the lengths of all three vectors are the same.
2. Form a matrix M2 from v_1, v_2 and v_3.
   Find the dimensions of M2 using the Octave command **size( )**.
   Find the minimum value in each column of M2.

Section 11: Simultaneous equations

Suppose that we have the simultaneous equations:

$$\begin{aligned} 2x - 3y &= 1 \\ x + 2y &= 4 \end{aligned}$$

[Check out the following steps on a piece of paper.] To solve these we would first note the $1x$ in the second equation and so we would switch the two equations. Then we would subtract 2 times the new first equation from the new second equation. Next we would divide the second equation by -7. Finally we would subtract 2 times the second equation from the first equation. These steps would result in the following *reduced* (or *equivalent*) equations:

$$\begin{aligned} 1x + 0y &= 2 \\ 0x + 1y &= 1 \end{aligned}$$

From these *reduced* equations we can simply read off the solutions, $x = 2$, $y = 1$ and then—of course—we would check the solutions in the *original* equations.

In these manipulations we only used x, y to make sure that we were working with the right numbers (to see this, just suppose that the second equation had been written $2y = 4 - 2x$).

So we might just as well write the equations in matrix form, with the first column representing the coefficients of x, the second column representing the coefficients of y and the third column representing the constants. [For emphasis the constants are separated from the variables by a vertical bar.]

$$\begin{bmatrix} 2 & -3 & | & 1 \\ 1 & 2 & | & 4 \end{bmatrix}$$

Now repeat *exactly* the same sequence of manipulations on the rows of the matrix that we performed on the equations :

$$\begin{bmatrix} 1 & 2 & | & 4 \\ 2 & -3 & | & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & | & 4 \\ 0 & -7 & | & -7 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & | & 4 \\ 0 & 1 & | & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & | & 2 \\ 0 & 1 & | & 1 \end{bmatrix}$$

From this final matrix we again see that $x = 2$, $y = 1$.

The process that we went through to obtain the *reduced* equations, is variously called "row reduction", "reduction to row echelon form", "Gauss-Jordan reduction", the "pivot method" etc, .

Doing this is tedious, to say the least, especially if there are three or more equations and non-integer coefficients. Fortunately this is the type of calculation that computers can do quickly, efficiently, and—most importantly—correctly. Octave has the built in command **rref( )** which does the work for us.

```
octave:1> M1 = [2 -3 1; 1 2 4]
         M1 =
         2 -3 1
         1 2 4
%
octave:2> solution_M1 = rref(M1)
         solution_M1 =
         1 0 2
         0 1 1
```

The same procedure would be followed in case there were more variables than equations (usually implying an infinite number of solutions where at least one of the variables is treated as a "constant"), or in case there are more equations than variables (usually implying that there are no solutions). The interpretation of the answers in these two cases becomes more involved.

Try this

1. Solve, first by hand and then by using using Octave, the equations:

$$\begin{aligned} 2x - y &= 2 \\ x + y &= 7 \end{aligned}$$

2. Solve, first by hand and then by using using Octave, the equations:

$$\begin{aligned} 2x - y + z &= 2 \\ x + y - z &= 7 \\ x + y + 2z &= 4 \end{aligned}$$

   This problem is done by another method in section 20.

Section 12: Programs; for-loops

A *program* is simply a set of Octave instructions. Often we want to repeat an operation many times and instead of constantly recomputing individual quantities we create a program which contains a "for-loop". This consists of a statement of the form:

for k = [set of indices]

    instructions to do something

endfor

[Note that **endfor** is written as *one* word, and not two words, “end for”. **for** and **endfor** are in lower case letters. We saw the same thing in section 02, where we wrote **endfunction**]

Do not use `i` or `j` as the symbol for the indice as these are reserved for $\sqrt{-1}$.

To create a program we place the instructions, including the for-loop, inside an “m-file”. For illustrative purposes, suppose we want the squares of the first five integers. Here is a program to do this; note the use of a semi-colon at line 3 and **disp( )** to display the answer:

```
% [for_loop_01.m]
for k = [1:5]
square = (k).^2;
disp(square)
endfor
```

To execute this program we go to the Octave prompt and type “for_loop_01”. *without the extension .m.*

```
octave:1> for_loop_01    % name of the program
     1
     4
     9
     16
     25
```

Two things should be noted:

i. For this simple example we could have obtained the answer using element by element powers of a vector:

```
octave:2> k = [1 : 5];
octave:3> powers = (k).^(2);
octave:4> disp(powers)
      1 4 9 16 25
```

   In more complicated situations (section 13A) both vectors and loops will be used.

ii. Octave prints out the answers one at a time. This can be very inconvenient if we are displaying many answers. To get around this we first store all the answers in a vector `store` and only display `store` at the end.

Storing values

Since vectors are just collections of numbers, the individual elements can be displayed by means of the index. Thus if `a` is a vector then the third element is `a(3)`:

```
octave:5> a = [7 -2 5 4];
octave:6> a(3)
      ans = 5
```

Now we want to create a vector of zeros that has the same length as `a`. In the above example we could count, but suppose we had a large vector of data. We use the function **length(a)**, but there is no need to display the value. We then create a vector `store` whose length is the same as that of `a`.

```
octave:7> l = length(a);  % no need to display the value
octave:8> store = zeros(1 , l)
      store = 0 0 0 0
```

Next we change the value of the third element of store from 0 to 8, by setting `store(3)` = `8`.

```
octave:7> store(3) = 8;    an ‘‘assignment’’
octave:8> disp(store)
      0 0 8 0
```

If we use this technique in a for-loop, then all we have to do at the end is write **disp(**`store`**)**. We could also write **mean(**`store`**)** if we wanted the mean etc.

Other sets of indices

In the for-loop above our index set was `1 : 5`, Octave will accept *any* set of indices. Try out the following sets:

```
% [for_loop_02.m]
for k = [10 : -2 : 5]     % or [3 5 7 11 13]
    square = (k).^2;
endfor
```

Try these

1. Run [for_loop_02.m]. What are the values for which the square is evaluated?
2. Run [for_loop_03.m]. What are the values for which the square is evaluated?
3. Write a program that has the statements `total` = 0 and a = [7 -2 5 4] *before* the loop begins. Now loop 4 times. At step k evaluate `a(k)` and add `a(k)` to `total`, by means of the *assignment*:
   `total` = `total` + `a(k)`
   [In an *assignment* the internal counter for `total` replaces the value on the right, by the new value on the left.]
   At the end of the 4 loops, display the value of `total` and compare it to **sum(**`a`**).**

Section 13: More on for-loops

The new elements in the following program are:

i. *before* we start the loop we assign *inital values* to two quantities, a and b.

ii. each time we loop we change the values of a and b.

```
% [fibonnaci.m]
%
% initial values
  a = 1;
  b = 1;
%
%    loop 10 times, each time:
%    1. find the ratio b/a
%    2. add a+b = c
%    3. b becomes the new a
%    4. c becomes the new b
%
%    we want to store the values (section 12)
  store_ratio = zeros(1,10);
  store_sum = zeros(1,10);
for k = [1 : 10]
  ratio = b/a;
  store_ratio(k) = ratio;
%
  c = a+b;
  store_sum(k) = c;
%
%    now assign new values to a and b
  a = b; % do this first, before b changes!
  b = c;
endfor
  disp(store_sum)
  disp(store_ratio)
```

We now run the program:

```
octave:5> fbonacci    % the name without the extension .m
%    the output is:
    1  1  2  3  5  8  13  21  34  55  89  144
    1.0000 2.0000 1.5000 1.6667 1.6000 1.6250
    1.6154 1.6190 1.6176 1.6182
%
%    switch to the high precision display
octave:6> format long
```

```
%    display the last ratio
octave:7> store_ratio(10)    the 10th element of store_ratio
      1.61818181818182    % the repeating infinite series = 1.6182
% we can check, the last ratio should be 89/55
octave:12> 89/55
      1.61818181818182
%
%    recall the ‘‘golden number’’ of section 08.
octave:8> G = (1+sqrt(5))/2;
octave:9> disp(G)
      1.61803398874989
```

The sequence {1 1 2 3 5 8 13 21 34 55 89 144 ... } is called the Fibonacci[1] sequence. The ratios become closer and closer to `G`, alternating between less than and greater than `G`.

Note

1. Fibonacci was active ca. 1200. He presents the sequence which bears his name in connection with the “rabbit problem”. Despite the fact that he was in command of the *geometrical* properties of the golden number, there is no indication that he was aware of the connection. For a discussion, see my book listed in the note to section 08. It is possible that Fibonacci learned about the “rabbit problem” from the Arab world.

Try this

1. Start with 1. Next compute:

$$1+\frac{1}{1}$$

then:

$$1+\frac{1}{1+\frac{1}{1}}$$

then:

$$1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}} \quad \ldots$$

   i. Take a guess as to what will happen when we continue the process. [hint: there is no need to give you a hint!]

   ii. Write a program to compute the first ten values and then compare the result of your computation with your guess.

note: This example belongs to the beautiful theory of *continued fractions*. If we reverse the process and find the continued fraction of an irrational number, e.g. π, then we can find the successive “best” *rational approximations* to the irrational number. For π the successive “best” approximations are $\frac{3}{1}$, $\frac{22}{7}$, $\frac{333}{106}$, $\frac{355}{113}$, $\frac{103993}{33102}$. This last rational is equal to 3.14159265358979, with an error estimate of $\frac{1}{1099482930}$.

Section 14: Conditional statements; input from the keyboard

Sometimes we want to test a series of numbers, or a vector, or a matrix to see if certain conditions are satisfied. In Octave the testing is done via *conditional statements*. In its simplest form a conditional statement looks like:

if *state the condition*
state what to do if the condition is satisfied
endif

[note that the closing is *one* word: **endif**, not “end if”]

Conditional statements can appear in functions as well as in programs. Here is a simple example; we want to write a function which will test if an integer is divisible by 27. Here we use the **floor** function [greatest integer function] of section 02. If a number `n` is divisible by 27 then there is no remainder and so **floor(**`n/27`**)** = `n/27`. If this condition is satisfied then the function prints out `n` and `n/27`. [The statement “`j` =” is not really needed here; it is put in order to follow the function format of section 03.]

```
% [if_simple.m]
%
function j = if_simple(n)
%    state the condition
if floor(n/27) == n/27  % double == for equality
%    state what to do if the condition is satisfied
  disp(n), disp(n/27)
%
endif    % completes the conditional statement
endfunction
```

We run the program, first with 54 and then with 271. For 271 the condition associated with **if** is not satsified so nothing happens.

```
octave:10> if_simple(54)
      54
      2
octave:11> if_simple(271)
octave:12>    % nothing has happened, so the next prompt appears
```

If we wanted something to happen when `n` = 271 we could add an **else** statement for the cases when the condition of the **if** statement is *not* satisfied:

```
% [else_simple.m]
function j = else_simple(n)    %
if floor(n/27) == n/27  % double == for equality
  disp(n), disp(n/27)
```

```
%    say what to do in the alternate (else) case:
else disp(n), disp(ceil(n/27))
endif    % just one end statement for both if and else
endfunction
```

If instead of just an alternative to the **if** condition, we had a second condition, we would replace **else** by **elseif**, e.g.:

```
elseif floor(n/18) == n/18
  disp(n), disp(ceil(n/18))
%   we might perhaps want another elseif statement
elseif fix(n/96) == 0
  disp(n+24)
```

We can have as many **elseif** statements as we need, but at most one **else** statement.

Input from the keyboard

When writing a program that will be used for many different input values, it is useful to be able to supply the data externally. This might happen if we generated the elements of a vector or matrix via one program and then wanted to perform calculations on the vector or matrix. A typical input statement would look like this [note the semi-colon]:

keyboard_vector = **input('** What is the vector, eh? **');**

As an example we tranform the above *function* if_simple( ) into a *program* with input. The statements **function** and **endfunction** no longer appear. [Do not use “input” as part of the name as it is a reserved function.]

```
% [simple_keyboard.m]
%
n = input(' What is the number, eh?  ');
%    [N.B. the Canadian shibboleth, ‘‘eh’’ is optional]
%    the rest of the program is identical to [if_keyboard.m]
if floor(n/27) == n/27  % double == for equality
  disp(n), disp(n/27)
endif
```

Try these

1. Check the first 99 integers to see if they are squares. If the answer is yes. print out the integer and its square root. Repeat using the **input** of a number `n`.
2. Consider all the multiples of $\frac{\pi}{4}$ on the interval [0 ,1]. Check which of the following *mutually exclusive* conditions are satisfied:
   i. the cosine is an integer not equal to 0.
   ii. the cosine is equal to 0.
   iii. the cosine is a negative number, but not an integer.

Section 15: While-loops; alphanumeric texts

In some situations we can not determine in advance how many times we want to loop because we want to keep going until a certain condition is met. In such cases a *while loop* is very useful. In the first example we start with `k` = 5 and we want to find the last value of `k` such that the square is less than or equal to 98.

The hitch is that we have to go above 98 to know when to stop. This is the time to *think mathematically*; if a certain value of `k` puts us above 98, then the value that we want is `k-1`. That is why, *after* we exit the while-loop we set the variable last_k= k-1.

A technical feature that we now add is the display of *alphanumeric text*. The text that is to be displayed is enclosed between two apostrophes: **disp('**the output =**')**.

```
% [while_square.m]
%    while-loops do not have a set of indices as in for-loops
%    rather we provide an initial value, in this case 5
      k = 5;
%
%    we now write while followed by the condition
while (k)^2 <= 98
  k = k+1;    ``increment'' the index value
endwhile
%    at this point we have broken out of the while-loop
%    the last value of k put us over 98 so we go back to k-1
  last_k = k-1;
%    now we give the commant to display the answers
disp('last k for which square <= 98'), disp(last_k)
```

We run the program:

```
octave:1> while_square
      last k for which square <= 98    % alphanumeric output
      9     % numerical answer
```

More interesting examples involve continuing until we are within a certain tolerance. In section 13 we generated ten Fibonacci numbers and their ratios. We saw that the ratios came closer and closer to the golden number. Now we are going to keep going until we are within .00001 of the golden number. To do this we modify [*fibonacci.m*].

```
% [fibonacci_while.m]
%
%    initial values
  a = 1;
  b = 1;
  ratio = 1;    % we now also need a seed value for ratio
%
  G = (1 + sqrt(5))/2;    the ``golden number'' (section 08)
```

```
%    each time that we might reenter the loop we check if the
%    difference is within .00001 = 10^-5
while abs(ratio - G) >= 10^(-5)    % use the absolute value to check
%    for clarity we save a and b under new names
  previous_a = a;
  previous_b = b;
%
%    we add: ``previous_a'' to ``previous_b'' to obtain ``new_sum''
%    ``previous_b'' becomes the ``a'' for the ratio
%    ``new_sum'' becomes the ``b'' for the ratio
  new_sum = previous_a + previous_b;
%
  a = previous_b; % a for next loop; do before b changes!
  b = new_sum;  % b for next loop;
%    find the new ratio b/a
  ratio = b/a;    % to test the program take off the semi-colon
endwhile
%
  disp('Fibonacci_1 = '), disp(a)
  disp('Fibonacci_2 = '), disp(b)
  format long
  diff = abs(ratio - G);    % check the difference
  disp('difference = '), disp(diff)
```

We run the program:

```
octave:22> fibonacci_while
     Fibonacci_1 = 233
     Fibonacci_2 = 377  % before 2, 3, 5...55, 89; now 144, 233, 377
     difference = 8.23767693347577e-06
```

Try these

1. If we keep taking the square roots of a positive number, the roots will become closer and closer to 1. Write a program that asks for an input value and stops after we are within .001 of 1. Output the number of *iterations* needed.
2. In section 08 we saw that `G` satisfies $x^2 = 1 + x$. Taking the square root of both sides we see that `G` also satisfies:

$$x = \sqrt{1+x} = \sqrt{1+\sqrt{1+x}} = \sqrt{1+\sqrt{1+\sqrt{1+x}}} \quad \ldots$$

   So if we calculate:

$$x = \sqrt{1+1} = \sqrt{1+\sqrt{1+1}} = \sqrt{1+\sqrt{1+\sqrt{1+1}}} \quad \ldots$$

   we will come closer and closer to `G`. How many times do we have to repeat the process so as to come within .0001 of `G`? Repeat for question 1 of section 14.

Section 15A*: Pick a number, part 2

In section 06A we saw how to calculate the probability that if k people pick a number from {1, ,2 ... n}that *at least two* will pick the same number. We did the calculations for the "birthday problem" where n = 365 and for k = 10, 20, ... 60 . In question 1 you were asked to modify the birthday problem for arbitrary n.

All this was nice in theory, but suppose you wanted to try it out on a group of 30. The probability that two people out of the 30 have the same birthday is only .71. This means that about 29% of the time you would look pretty silly. With 25 people the probability is down to .57 and so about 43% of the time all the people in a random test group would have different birthdays.

What we want to look at is the *inverse* "birthday problem", namely for a given number of people k, how large can n be so that we have a degree of certainty that *at least two* will be the same number. There is no direct way of computing this maximum value of n, but this is not a problem; we "guess" n and let the computer check if we have gone below the chosen degree of certainty. If not, we guess again by increasing the value of n by 1.

In practical terms we use a *while-loop* as in the last section. As input values we give the value of k and the tolerance value. The intermediate calculations are the same as in section 06A.

```
% [pick_number_01.m]
%
k = input('how many people will pick a number?  ');
p1 = input('input you tolerance:  .5, .75, .90, .95, .99? ');
%
  p2 = 1 - p1;    maximum allowable value for prob_all_different
%
  n = k;    % starting value for n
  prob_all_different = 0;    % approximate probability when n = k
%
while prob_all_different < p2
%
% increment n by 1 for this new loop
  n = n + 1;    % assignment for new n
  a = [n: -1 : (n-k+1)];    % index vector [n, n-1, ... (n-k+1]
%    divide each element of a by n and multiply all the terms
  prob_all_different = prod((1/n)*a);
%
endwhile
%
%    once prob_all_different >= p2 we have gone too
%    far so the desired answer is n-1
  disp('k ='), disp(k), disp('people')
  disp('tolerance ='), disp(p1)
  disp('maximum n is'), disp(n-1)
```

We now run the program for k = 100 people and with tolerance levels of .99, .95, .90, .75, 50 .

```
tolerance = .99  maximum n = 1108
tolerance = .95  maximum n = 1685
tolerance = .90  maximum n = 2183
tolerance = .75  maximum n = 3603
tolerance = .50  maximum n = 7174
```

Notice how quickly the maximum value of n climbs with *decreasing* tolerance. If we use .95 we can ask 100 people to pick from {1, 2, ... 1685} and be certain 95% of the time that at least two people will pick the same number. If we want virtually absolute certainty (99% of the time) we can still go up to 1108. This is much higher than anyone would guess ahead of time.

Instead of calculating the maximum value for a fixed number (100) of people and different values of the tolerance, we could fix the tolerance and calculate the maximum value for different numbers of people. If we do this for tolerance = .99 (virtual certainly), then for 10, 20, ... 100, we obtain the values: 13, 48, 104, 183, 283, 404, 548, 713, 900, 1108.

The last number, 1108, is the same as above because the calculation is the same.

Try these

1. We saw in section 06A that for 50 people, the probability that at least two people have the same birthday is .970, where as for 60 people we are at .994. We can look at the *inverse problem*: What is the *least number of people* so that the probability that at least two have the same birthday is greater than or equal to .99? Find this number by using *pick_number_01* with tolerance = .99 and k = 50, 51, ... 60 people. [solution: at some point the maximum value will go from *below* 365 to *above* 365. we want to find the the *smallest* k for which we are *above* 365.
2. Plot the maximum values obtained in (1) against k = 50, 51, ... 60 people. Draw the horiziontal line $n = 365$. First draw the points discretely using squares of size 10 and then connect them using a dotted line of thickness 4; see the examples in section 07 (graphing). Label the point where the graph crosses the horizontal line. Give appropriate titles to the x-axis, y-axis and your graph.

Section 16★: Binomial coefficients and probabilities

Rosencrantz: Heads.
(He picks it up and puts it in his bag. The process is repeated.)
Heads.
(Again.)
Rosencrantz: Heads.
(Again.)
Heads.
(Again.)
Guildernstern (flipping a coin): There is an art to the building up of suspense.

— Tom Stoppard, *Rosencrantz & Guildenstern Are Dead*, Act One.

Suppose that a weighted coin has a probability .4 of coming up heads and that we flip the coin 5 times. We want to find the probability that we will have *exactly* 2 heads and 3 tails in the 5 tosses.

There are several ways in which this can happen: we could have the sequence {H H T T T}, or the sequence {H T H T T} etc.. If you list all the ways of having *exactly* 2 heads and 3 tails in 5 tosses you will see that there are indeed 10 possibilities.

Where does the number 10 come from? We can reason as follows: we can think of the tosses as "slots" and the 2 heads as "objects" to place in 2 of the 5 slots. For the first head we have the choice of 5 slots and for the second head we have the choice of 4 slots. That makes 5 x 4 = 20 possibilities. But this is not the correct answer because the 2 objects (i.e. the 2 heads) are identical. So we have to divide 20 by the 2 x 1 = 2! (2 factorial) = 2 duplications. So:

$$\text{number of \textit{distinct} ways of having 2 heads in 5 tosses} = \frac{5 \times 4}{2 \times 1} = 10$$

Because of the way it arises, the calculated quantity is referred to as "5 choose 2". These same numbers also appear in the binomial expansion and so are also referred to in general as the "binomial coefficients". There are different symbols for these including $\binom{5}{2}$ and C(5,2). If we toss n times, then the number of ways in which we can have k heads is given by:

$$\text{n choose k} = \binom{n}{k} = \frac{n \times n-1 \times \ldots (n-k+1)}{k!}$$

Octave has two functions, for the two names, **nchoosek(**`n,k`**)** and **bincoeff(**`n,k`**)** of evaluating the binomial coefficients:

```
octave:7> nchoosek(5,2)
       ans = 10
octave:8> bincoeff(5,2)
       ans = 10
```

Now we want to find the probability of exactly 2 heads in 5 tosses. Since the weighted coin has a probability .4 of coming up heads on any particular toss, the probability of tails is .6. So for any *particular* sequence (e.g. {T H H T T}) the probability is $.4 \times .4 \times .6 \times .6 \times .6$.

Since there are 10 possible sequences involving *exactly* 2 heads and 3 tails we have:

$$\text{probability of } \textit{exactly} \text{ 2 heads in 5 tosses} = 10 \cdot (.4)^2 \cdot (.6)^3 = \binom{5}{2} \cdot (.4)^2 \cdot (.6)^3$$

This is called the *binomial probability*. The general formula, when the probability of heads on *one* toss is p and the probability of tails on *one* toss is $q = 1 - p$ is:

$$\text{probability of k heads in n tosses} = \binom{n}{k} \cdot (p)^{\text{k}} \cdot (q)^{\text{n-k}}$$

Octave has the function **binopdf(**`k,n,p`) to evaluate the binomial probabilities; note that the number of heads `k` comes first, then number of tosses `n`. [**pdf** stands for "probability distribution function"]

```
octave:1> p2 = binopdf(2 , 5, .4)
octave:2> disp(p2)
      0.34560
```

Try these

1. Instead of counting the number of ways we can have 2 heads in 5 tosses, we could count the number of ways we can have 3 tails in 5 tosses. The number of ways must be exactly the same because if we have 2 heads then we have 3 tails and if we have 3 tails we must have 2 heads. If you are convinced by the previous two sentences of this paragraph then what result have you just proved?
2. Above we computed the probability of exactly 2 heads. Call this probability `p2`. Do the same for 0, 1, 3, 4 , 5 heads. Call these probabilities `p0`, `p1`, `p3`, `p4`, `p5`. Add up all the probabilities. What result have you just proved?
3. When people in France come into a room they shake hands with all the other people. If there are 24 Français in a room, how many handshakes will there have been by the time they have finished? Suppose that they change the custom and have three people at a time clasp hands?
4. Use **prod( )** to create our own choice function of *two* variables: **choice(**`n,k`**)**.

Section 17★: Tossing coins on a computer, part 1

We understand intuitively what tossing a coin "at random" means, but defining it precisely is a different story. In fact no one has ever been able to give a completely rigorous mathematical definition of "randomness". Instead one uses a series of statistical tests to check if there is no reason to reject a certain phenomena as being "random".

Random phenomena, such as coin tossings, are *simulated* on computers by means of *random number generators*.[1]

Terminology: Computer generated random numbers are elements of a very large set of numbers $\{x_k\}$such that $0 < x_k < 1$.

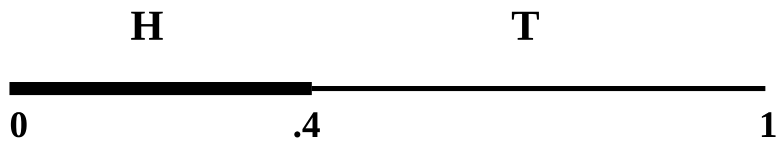
To output random numbers Octave has the command **rand(1, n)** which will generate a random vector of length `n`. Here is the output if we generate 10 random numbers :

```
octave:1> random_01 = rand(1,10);
octave:2> disp(random_01)
    0.700931 0.501059 0.344372 0.565966 0.429773
    0.126794 0.013282 0.684946 0.486618 0.864318
```

If we repeated the same command we would obtain a *new* set of 10 random numbers.

★★ We can reobtain the same set if we reset the seed , e.g. 45834, each time, using the the command: **rand('seed', 45834)**. This feature is important for testing simulations.

In section 06A we discussed coin-tossing with a weighted coin that had a probability .4 of coming up heads. To explain how we simulate tossing this coin, consider the following diagram:



The interval [0 .4] has length .4 and so takes up 40% of the interval [0 1]. If the numbers that we generate *act* as if they were generated at random then we would expect that about 40% of them would lie in the interval [0 .4]. If we generate 1000 random numbers and then count how many (using the vector logic of section 08) of them fall in the interval [0 .4], then we would expect that the answer is somewhere near 400. If we then divide the number by 1000 we will have the *relative frequency* (also called the *empirical probability*) of "heads". In turn the relative frequency should be close to the *theoretical probability* which is .4.

To illustrate, via a short example, we first generate 6 random numbers:

```
octave:1> a=rand(1 , 6);
      0.89997 0.32240 0.82164 0.58678 0.91927 0.28393
```

To test for "heads" (i.e. which random numbers are <= .4), we use vector logic [section 09]. We let vector `b` = [`a` <= .4]. Octave places a 1 in `b` wherever the corresponding random number in vector `a` is <= .4. Check this visually by looking at the random numbers just above and then the 1s just below:

```
octave:2> b = [a <= .4]
        0       1       0       0       1
```

To find the number of "heads", we simply add the 1s.

```
octave:3> c = sum(b)
      2
```

Finally we find the relative frequency of "heads" by dividing c by 6.

```
octave:4> relative_frequency = c/6
      0.33333
```

Now we do exactly the same thing for 1000 "tosses", by generating 1000 random numbers:

```
octave:5> e = rand(1 , 1000);       % Do not forget the semi-colon!
%
octave:6> f = [a <= .4];
octave:7> g = sum( )
      ans = 392
octave:8> relative_frequency = g/100
      relative_frequency = = 0.39200    % this is ‘‘close’’ to .4
```

Notes

1. The usual—new methods are always being developed—technique used to generate the random numbers is "modular arithmetic". This can be illustrated by thinking of a twelve hour clock. We start with 2 as a "seed". Then we use 5 as a "multiplier": $2 \cdot 5 = 10$. We divide by 12: $\frac{10}{12} = 0.83336$ and this is our first random number.

   Now we multiply $10 \cdot 5 = 50 = 4 \cdot 12 + 2$, i.e. we have gone around the clock four times and we are now at 2 o'clock. We again divide by 12: $\frac{2}{12} = 0.1666$ and this is our second random number. But 2 was the original seed so if we continue we again obtain 10! This is certainly *not* random!

   Finding the right values for the seed, multiplier and divisor is not an easy task especially since simulations require millions, billions, even trillions (atomic physics) of random numbers which do not repeat. Finding a good random number generator is both a science and an art. Octave (version 3.6.3) uses a generator with a period of $2^{19937}-1$. [Use logarithms to express $2^{19937}-1$ as a power of 10. If you try to evaluate this number in Octave, the answer will be "infinity".]

Try these

1. Generate a 1000 random numbers and determine what fraction lies between .55 and .82. [Use step 4 of Section 09.]
2. Convince yourself by a few numerical examples that if x is a random number and N is an integer, then the formula:

   $$j = 1 + \textbf{floor}(N \cdot x)$$

   will give a random integer from the sequence $\{1, 2, \dots N\}$.

   Use this formula with N = 6 to simulate tossing a die 1000 times.

Section 18★★: Tossing coins on a computer, part 2

" 'I'll tell you what,' said Mr. Bumblemoose when Joachim had stopped reading, 'he's one of those new-fangled Professors who think you can prove everything with a computer. What do all those rows of figures and statistics mean to me? Nothing at all! These new-look Professors are just witch-doctors, but they do it with a row of figures and make you so dizzy that in the end you believe everything they tell you. But I don't believe it!' "

— Hans Andreus, "Mr. Bumblemoose Quarrels with Joachim", in *Mr. Bumblemoose and the Mumblepuss*, London: Abelard-Schumanan, 1973 (original Dutch version, 1968), p. 29.

In section 17 we tossed *one* coin, with the theoretical probability of heads on *one* toss being .4. When we repeated this experiment 1000 times the *relative frequency of heads* (or *empirical probability*) was close to the *theoretical* probability. Now we return to section 15 where we tossed the same coin 5 times, with the probability of *2 heads in 5 tosses* being given by the binomial distribution.

The idea is the same as in section 17, but now instead of generating *one* random number at a time, we generate *five* random numbers at a time. This will be referred to as the *basic experiment*. Next we repeat the *basic experiment* 1000 times. Instead of a 1×1000 *vector*, we now are going to have a 5×1000 *matrix*. The Octave logic remains the same for matrices as for vectors.

To illustrate, here is a 6 time repetition of the *basic experiment*. What makes this simulation more complicated is that we first have to add the number of 1s in each column (i.e. count the number of heads in 5 tosses). This results in a *vector* `c`. Then we have to use Octave logic a second time to find out which elements of the *vector* `c` are equal to 2 (i.e. which of the 6 repetitions resulted in exactly 2 heads).

Check this visually by looking at the random numbers in `A` and then the 1s in `B`:

```
octave:1> A = rand(5 , 6)     capital letters for matrices
    0.763913 0.253032 0.357587 0.091995 0.120525 0.638336
    0.933027 0.764294 0.532767 0.178103 0.944216 0.299117
    0.794544 0.332849 0.814172 0.594680 0.025912 0.644570
    0.838468 0.488249 0.449008 0.250096 0.533096 0.052626
    0.077102 0.439889 0.798099 0.583340 0.955996 0.466741
%
octave:2> B = [A <= .4]
    0 1 1 1 1 0
    0 0 0 1 0 1
    0 1 0 0 1 0
    0 0 0 1 0 1
    1 0 0 0 0 0
```

```
%    now we add the columns to see how many heads
%    check the column sums by hand
octave:3> c = sum(B)
      1 2 1 3 2 2
%
%    which elements of c equal 2?
octave:4> d = [c == 2]    % note the double equal sign
%    check that the 1s in vector d match up with the 2s in vector c
      0 1 0 0 1 1
%
%    now add up the ones in c to find out
%    how many times we had a 2 in c
octave:5> e = sum(d)
      3
%
% finally the relative frequency
octave:6> f = e/6
      0.50000
```

We do exactly the same thing for 1000 repetitions and compare with the binomial distribution of section 15.

```
octave:7> A = rand(5 , 1000);
octave:8> B = [A <= .4];
octave:9> c = sum(B);
octave:10> d = [c == 2];
octave:11> e = sum(d);
octave:12> f = e/1000
      0.34300
octave:2> binopdf(2 , 5, .4)
      0.34560
```

Try this

1. Reconsider problem 2 of section 17. Suppose that we toss the die 7 times. What is the theorical probability of *exactly* one four? Use Octave to simulate this basic experiment 500 times and find the empirical probability of *exactly* one four in 7 tosses.

Section 19: Statistics, part 3; Histograms

A wide variety of data appears to follow the *normal distribution* (often loosely referred to as the "bell-shaped curve") . For the purpose of providing data we want to generate *normal random numbers* with mean = 0 and standard deviation =1. To obtain these numbers we use the function **randn**(n,1). Here n is the number of normal random numbers that we want to generate. To have a large sample, we generate 8261 random normal numbers [be sure to put a semi-colon, to suppress printing, after each statement]:

```
octave:1> x = randn(8261,1);
% we check the mean and standard deviation (section 05)
octave:2> mean(x)
ans = 0.011881  % theoretical value = 0
octave:3> std(x)
ans = 0.99582  % theoretical value = 1
```

The next step is to use x to obtain values with a mean of 1.42 and a standard deviation of .31. To do this we multiply all values in x by the desired standard deviation and then add the desired mean:

```
octave:4> y = .31*x + 1.42;
% we check the mean and standard deviation (section 05)
octave:5> mean(y)
ans = 1.4237  theoretical value = 1.42
octave:6 > std(y)
ans = 0.30870  theoretical value = .31
```

We are now ready to find the **histogram**. The command is **hist**(y groupings). Here y is the data that we want to analyse and groupings is the number of divisions of the data that we wish. Deciding the number of groupings is partially a question of trial-and-error; here we try 15 groupings with the vector obtained in statement 4:

```
octave:  7> hist(y 15)
```

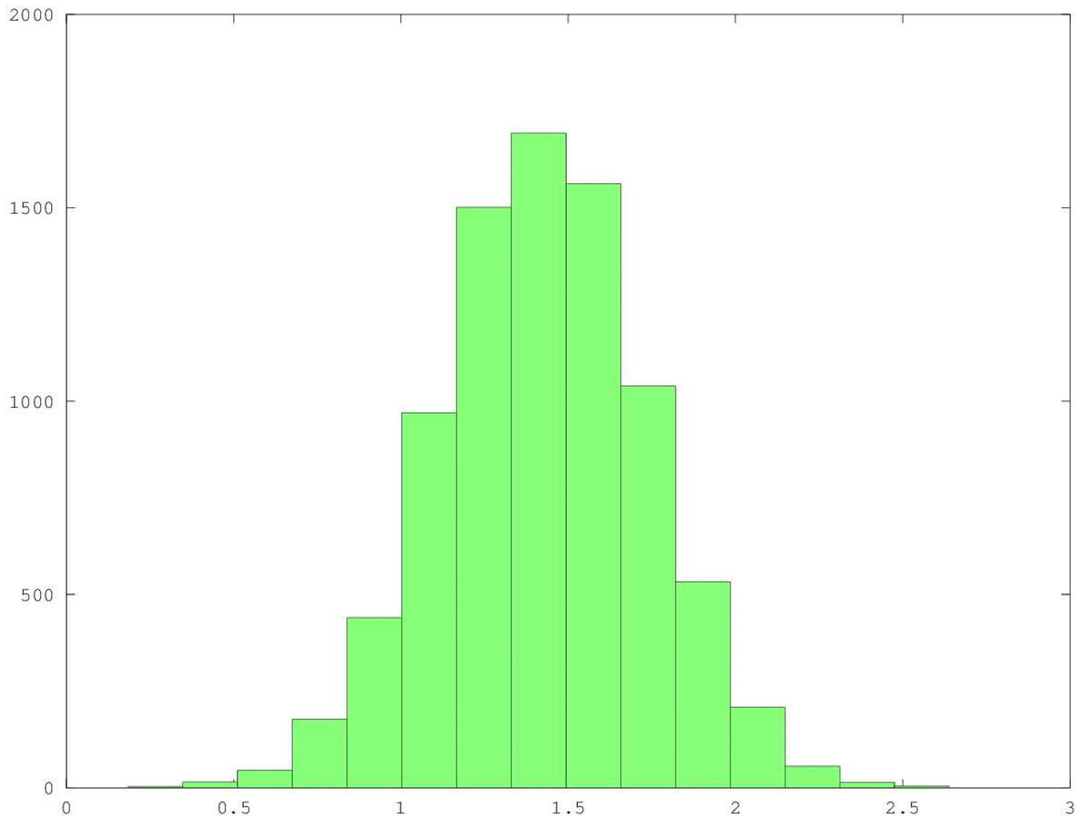
Now we are ready to obtain the graph (section 07; we could have also printed directly to a PDF file):

```
octave:8 > print -dpng normal_[mean-1.42--std-.31]_[15-bins].png
```

Notice how the graph is more or less centred about the desired mean = 1.42. If we had used a much larger data set and a larger value for the number of groupings we would be closer to the normal curve.

For the normal curve about 95% of the area is within 2×std of the mean. In our case this would mean between approximately 1.4 + .6 = 2 and 1.4 – .6 = 0.8 and visually this does not seem unreasonable. Only a more sophisticated test (e.g. the chi-squared test) would determine how close the histogram is to the normal curve

We obtain the following graph:



Try these

1. Repeat the above with a mean of -2.6 and a standard deviation of 4.9. Generate 9,416 normal random numbers. Try with several different values for the number of groupings.
2. Use the ordinary random number generator of section 17 and exercise 2 of that section to simulate 5000 tosses of a die. Obtain a histogram with 10 divisions.

Section 20: Matrices, part 2

In section 10 we saw how to enter data for matrices and how to build matrices from vectors. Then in section 11 we discussed the relationship between matrices and equations. Here we continue the discussion and introduce new operations on matrices. In section 10 we spoke of a 2 by 3 matrix. We also introduce the notation, 2×3 matrix and use the terminology, the *dimensions* of a matrix.

Matrices of ones, constant matrices

In section 06 we used the function **ones** to generate a *vector* all of whose elments are equal to 1. The same function can generate a *matrix* of ones. Suppose that we want a 2×3 matrix of 1s:

```
octave:1> R = ones(2,3)
     R =
     1 1 1
     1 1 1
```

Next we want to multiply *each* element of `R` by 5. The symbol for multiplication is `*`, the same as for the multiplication of numbers:

```
octave:2> S = 5*R
     S=
     5 5 5
     5 5 5
```

Addition of matrices

Suppose we have two 2×3 matrices

$$\mathtt{U} = \begin{bmatrix} 7 & -1 & 8 \\ 0 & 5 & 1 \end{bmatrix} \qquad \mathtt{V} = \begin{bmatrix} 2 & 1 & -3 \\ 4 & 3 & -2 \end{bmatrix}$$

First we want to add 6 to each element of `U`. The symbol for addition is `+`, the same as for the addition of numbers:

```
octave:3> U1= 6+U
     U1 =
     13 5 14
     6 11 7
```

Next, since `U` and `V` have the same dimensions we can add them, by adding each element of `U` to the corresponding element of `V`. Once again the symbol is `+`:

```
octave:4> W = U+V
     W =
     9 0  5
     4 3 -2
```

Multiplication of matrices

First we create a 2×3 matrix `M` and a 3×2 matrix `N`.

Section 20: Matrices, part 2

$$\texttt{M} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \texttt{N} = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Whereas the addition of matrices and the multiplication of a matrix by a *number* are similar to what we do for numbers, the same is not true for the multiplication of two *matrices*. The symbol is once again `*`, but in order for the multiplication to make sense, one *condition* must be satisfied:

> The number of *columns* of the matrix on the *left* must be the same as the number of *rows* of the matrix on the *right*. The new matrix will have the same number of *rows* as the matrix on the *left* and will have the same number of *columns* as the matrix on the *right*

Since `M` has 3 columns and `N` has 3 rows the condition is satisfied. Also because `M` has 2 rows and `N` has 2 columns, `M*N` will have 2 rows and 2 columns.

The actual proceedure consists of multiplying the entries in each *row* of the matrix on the left with the entries in each *column* of the matrix on the right. For example the *second row* of `M` = [4 5 6] and the *first column* of `N` = [7 9 11]. We multiply and add as follows: $4\times7+5\times9+6\times11 = 139$. This becomes the element in the *second row* and *first column* of `M*N`.

```
octave:5> P = M*N
    P =
    58   64
    139 154
```

Since `P` is a *square* matrix the condition required for multiplying `P*P` is automatically satisfied and the product will have the same dimensions as `P`. The next step would be to calculate `P*P*P`. But there is no need to write `P` three times, for just as we write `a*a*a` = a^3, we can simply write $\texttt{P}^3$.

```
octave:6> P^3
    2597032 2874112
    6242212 6908200
```

Notice that `N*M` is *meaningless* because the condition is no longer satisfied.

Even if `A` and `B` are square matrices—so that again the condition for multiplication is satisfied—`A*B` is *not* equal to `B*A`. This is another feature that differs from the multiplication of numbers where $a\times b = b\times a$.

The identity matrix, inverses

The function **eye( )** produces a *square* matrix `I` with 1s on the diagonal and 0 elsewhere; this is called the *identity* matrix:

```
octave:7> I = eyes(2,2)
    1 0
    0 1
```

I plays the same role with matrices as 1 does with numbers, i.e. 1*5 = 5; 5*1 = 5 and I*S = S; S*I = S.

```
octave:8> S1= I*S
    S1 =
    58   64
    139 154    % this is S again so I*S = S
```

The inverse of the number a = 2 is $a^{-1} = \frac{1}{2}$. If we multiply a number and its inverse we always obtain 1 again; i.e. $a \times a^{-1} = 2 \times \frac{1}{2} = 1$.

Most, but not all, *square* matrices have inverses. The role played by 1 with numbers is replaced by the identity matrix I. The inverse of a matrix S is denoted by S^{-1} and we have: $S^{-1} * S = I$. Aside from the 2×2 case, finding the inverse requires a lot of computation. Even in the 3×3 case it is very easy to make mistakes when doing it by hand. The same is of course true for solving equations and in fact the algorithmic method for finding the inverse of a matrix is *directly* related to the row reduction method discussed in section 11. Fortunately Octave performs all these operations very efficiently. For the inverse of S, we simply write **inv(**S):

```
octave:9> S2 = inv(S)
    S2 =
    4.2778 -1.7778
    -3.8611 1.6111
```

We check:

```
octave:10> S*S2
    1 0
    0 1
```

which is the 2×2 identity matrix. We could also check S2*****S.

Solving equations using inverses

If we have the equation:

$$2x = 1$$

then we can solve by multiplying both sides by the *inverse* of 2: $2^{-1} = \frac{1}{2}$:

$$\frac{1}{2} \times 2x = \frac{1}{2} \times 1$$

which gives

$$x = \frac{1}{2}$$

The same idea applies to equations. Suppose we have the same 3 equations in 3 unknowns as in section 11, exercise 1:

$$\begin{aligned} 2x - y + z &= 2 \\ x + y - z &= 7 \\ x + y + 2z &= 4 \end{aligned}$$

We first look at C, the matrix of the *coefficients* on the left hand side of the equation:

$$\mathtt{C} = \begin{bmatrix} 2 & -1 & 1 \\ 1 & 1 & -1 \\ 1 & 1 & 2 \end{bmatrix}$$

Next we find D, the *inverse* matrix of C, and then check:

```
octave:11> D = inv(C)
     D =
      0.33333  0.33333 -0.00000
     -0.33333  0.33333  0.33333
      0.00000 -0.33333  0.33333
```

```
octave:12> D*C
     1.00000 0.00000 0.00000
     0.00000 1.00000 0.00000
     0.00000 0.00000 1.00000
```

Now we obtain the 1×3 *column vector* of the *constants* on the right side of the equations (note how semi-colons are used to put each number on a new line).

```
octave:13> constants = [2; 7; 4]
     constants =
     2
     7
     4
```

Finally we obtain the solution to the equations by following the lead of what we did to solve $2x = 1$; we multiply the vector of *constants* by D the *inverse* matrix of C:

$$\mathtt{solution} = \mathtt{D*constants}$$

```
octave:14 > solution = D*constants
     solution =
      3.00000
      3.00000
     -1.00000
```

Determinants

As with inverses, the concept of a *determinant* only applies to *square matrices*. Aside from the 2×2 case, which corresponds to 2 equations in 2 unknowns, determinants are now used only in theoretical discussions. If the determinant of a square matrix is 0, then the matrix does not have an inverse, and if a matrix does not have an inverse then the determinant equals 0. This corresponds to the situation with numbers where a non-zero number has an inverse, but the number 0 does not have an inverse.

```
octave:15> det(C)
     ans = 9
```

Try these

1. With `U` and `V` as above, evaluate `5*U - 3*V`
2. With `N` and `M` as above try calculating `N*M` and see what Octave says.
3. Make up a 3×3 matrix `P` such that all the entries are positive and each row adds to 1. Keep on taking higher and higher powers of `P`. What do you notice? Try the same thing with a 4×4 matrix. A matrix with non-zero entries, whose rows add to 1, is called a *Markov* chain. They have many interesting practical applications; see `http://en.wikipedia.org/wiki/Examples_of_Markov_chains`.
4. Make up two 2×2 matrices `A` and `B`. Calculate `A*B` and `B*A`. Are the two products equal? Do the same with two 3×3 matrices.
5. In the above solution of equations, multipy `C`, the *coefficient* matrix, by the *solution* vector and show that that we obtain the vector of *constants* back again. In words, we have checked the solution. Before you do the multiplication do a mental check of the dimensions of `C` and `solution` so as to make sure that the multiplication make sense.
6. With `D` = `C`$^{-1}$, find the inverse of `D` and show that we obtain `C` back again. What result does this correspond to in the algebra of numbers?
7. Make up your own set of four equations in 4 unknowns and solve by the inverse method shown here. Do not make up a "textbook" equation such as $2x + 3y \ldots$; try something such as $\pi x + \sqrt{3}y + -1.270z + 27.2w = 2^{1.12}$ etc. How long would it take you to solve these equations with pencil and paper, using the row reduction method of section 11?
8. Create the 5×5 identity matrix. What is its determinant?
9. Above we calculated `D`, the *inverse* matrx of `C`? What is the determinant of `D`? Compare it to the determinant of `C`. What result, relating inverses and determinants, have you just "proved"?
10. Make up two 3×3 matrices `A` and `B`. Find the determinants of `A` and `B` and then the determinant of `A*B`. What result have you just "proved"? Why is question (9) a special case of this result?

Table of Contents

A ★, here and in the text, indicates a special topic. As indicated by the double star, section 18 involves a more complicated situation.