A BRIEF INTRODUCTION TO MATLAB

BRAD BAXTER

Version: 202210201136

Abstract. This is a short introduction to scientific computation in MATLAB. It is designed for self-study. This tutorial is also suitable for Octave. All computing classes will be online only.

Contents

1. Introduction

These notes, and much else, can be obtained from

`http://econ109.econ.bbk.ac.uk/brad/teaching/Methods/`

The book *Numerical Methods in Finance and Economics: A MATLAB-based Introduction*, by P. Brandimarte, contains many excellent examples, and is strongly recommended, particularly for MSc Mathematical Finance. I also recommend the

undergraduate-level textbook *An Introduction to Financial Option Valuation*, by D. Higham, which is particularly clear.

All of the programs in this note also work with Octave, which is a free quasi-clone of MATLAB, and can be found here:

`http://octave.org/`

Another good quasi-clone is

`http://freemat.sourceforge.net/`

2. MATLAB Basics

2.1. **Matrices and Vectors.** MATLAB (i.e. MATrix LABoratory) was designed for numerical linear algebra.

Notation: a $p \times q$ matrix has p rows and q columns; its entries are usually real numbers in these notes, but they can also be complex numbers. A $p \times 1$ matrix is also called a *column vector*, and a $1 \times q$ matrix is called a *row vector*. If $p = q = 1$, then it's called a scalar.

Entering commands: You can enter commands at the command prompt. However, there are more than one or two lines of commands, it is usually better to create a script, or `.m` file. A `.m` file is simply a textfile whose filename extension is `.m`, to indicate that it is a MATLAB program. I will describe script creation in the MATLAB class.

We can easily enter matrices:

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
```

In this example, the semi-colon tells MATLAB the row is complete.

The transpose A^T of a matrix A is formed by swapping the rows and columns:

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
AT = A'
```

Sometimes we don't want to see intermediate results. If we add a semi-colon to the end of the line, the MATLAB computes silently:

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
AT = A'
```

Matrix multiplication is also easy. In this example, we compute AA^T and A^TA.

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
AT = A'
M1 = A*AT
M2 = AT*A
```

In general, matrix multiplication is non-commutative, as seen in Example 2.1.

Example 2.1. *As another example, let's take two column vectors* $\mathbf{u}$ *and* $\mathbf{v}$ *in* $\mathbb{R}^4$ *and compute the matrix products* $\mathbf{u}'\mathbf{v}$ *and* $\mathbf{u}\mathbf{v}'$. *The result might surprise you at first glance.*

```
u = [1; 2; 3; 4]
v = [5; 6; 7; 8]
u'*v
u*v'
```

Exercise 2.1. *Use Example 2.1 to find* $\mathbf{u}'\mathbf{v}$ *and* $\mathbf{u}\mathbf{v}'$ *when*

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \quad \text{and} \quad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}.$$

Exercise 2.2. *What's the general formula for* $A = \mathbf{u}\mathbf{v}'$ *and* $B = \mathbf{u}'\mathbf{v}$ *when* $\mathbf{u}$ *and* $\mathbf{v}$ *are column vectors in* $\mathbb{R}^n$*? In other words, find formulae for the components of* A *and* B*.*

2.2. **The `sum` function.** It's often very useful to be able to sum all of the elements in a vector, which is very easy in MATLAB:

```
u = [1 2 3 4]
sum(u)
```

The `sum` is also useful when dealing with matrices:

```
A = [1 2; 3 4]
sum(A)
```

You will see that MATLAB has summed each column of the matrix.

2.3. **Solving Linear Equations.** MATLAB can also solve linear equations painlessly. Specifically, to solve $A\mathbf{u} = \mathbf{v}$, we use

```
u = A\v;
```

This might seem to be a typographical error at first, but the backslash is correct. Here is a longer example, best created as a script. The lines beginning with percentage signs are comment lines, but you should include them as good programming practice.

Example 2.2.
```
n = 10
% M is a random n x n matrix
M = randn(n);
% y is a random n x 1 matrix, or column vector.
y = randn(n,1);
% solve M x = y
x = M\y
% check the solution
y - M*x
```

We shall need to measure the length, or *norm*, of a vector, and this is defined by

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2},$$

where $v_1, \ldots, v_n \in \mathbb{R}$ are the components of the vector $\mathbf{v}$; the corresponding MATLAB function is `norm(v)`. For example, to check the accuracy of the numerical solution of $M\mathbf{x} = \mathbf{y}$, we type `norm(y - M*x)`.

It's rarely necessary to compute the inverse M^{-1} of a matrix, because it's usually better to solve the corresponding linear system $M\mathbf{x} = \mathbf{y}$ using

```
x = M\y
```

as we did above. However, the corresponding MATLAB function is `inv(M)`.

2.4. **The MATLAB Colon Notation.** MATLAB has a very useful *Colon notation* for generating lists of equally-spaced numbers:

```
1:5
```

will generate the integers $1, 2, \ldots, 5$, while

```
1:0.5:4
```

will generate $1, 1.5, 2, 2.5, \ldots, 3.5, 4$, i.e. the middle number is the step-size.

Example 2.3. *This example illustrates a negative step-length and their use to generate a vector.*

```
v = [10:-1:1]';
w = 2*v
```

We can easily extract parts of a matrix using the MATLAB colon notation.

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
M = A(2:3, 2:3)
```

The same procedure works for a vector:

```
v = [1; 2; 3; 4]
u = v(2:3)
```

The following example illustrates the MATLAB **dot notation**

Example 2.4. *Consider the following MATLAB code.*

```
A = [1 2; 3 4]
A^2
A.^2
```

The first command uses matrix multiplication to multiply `A` *by itself, whilst the second creates a new matrix by squaring every component of* `A`.

Exercise 2.3. *What does the following do?*

```
sum([1:100].^2)
```

Exercise 2.4. *What does the following do?*

```
y = exp(-0.5 * [0:0.1:1].^2);
```

2.5. **Graphics.** Let's learn more about graphics.

Example 2.5. *Plotting a sine curve:*

```
t = 0: pi/200: 2*pi;
y = sin(3*t);
plot(t,y)
```

Exercise 2.5. *Replace* `plot(t,y)` *by* `plot(t,y,'o')` *in the last example.*

Example 2.6. *See if you can predict the result of this code before typing it:*

```
t = 0: pi/200: 2*pi;
y = sin(3*t).^2;
plot(t,y)
```

Exercise 2.6. *Use the* `plot` *function to plot the graph of the quadratic* $p(x) = 2x^2 - 5x + 2$, *for* $-3 \leq x \leq 3$.

Exercise 2.7. *Use the* `plot` *function to plot the graph of the cubic* $q(x) = x^3 - 3x^2 + 3x - 1$, *for* $-10 \leq x \leq 10$.

Example 2.7. *Here's a more substantial code fragment producing a cardioid as the envelope of certain circles. You'll also see this curve formed by light reflection on the surface of tea or coffee if there's a point source nearby (halogen bulbs are good for this). It also introduces the* `axis` *command, which makes sure that circles are displayed as proper circles (otherwise MATLAB rescales, turning circles into ellipses). This longer example should, of course, have its own* `.m` *file.*

```
hold off
clf
t=0:pi/2000:2*pi;
plot(cos(t)+1,sin(t))
axis([-1 5 -3 3])
axis('square')
hold on

M=10;
for alpha=0:pi/M:2*pi
    c=[cos(alpha)+1; sin(alpha)];
    r = norm(c);
    plot(c(1)+r*cos(t),c(2)+r*sin(t));
end
```

Exercise 2.8. *Generate a graph of the Gaussian* $N(0,1)$ *probability density function*

$$p(s) = (2\pi)^{-1/2} e^{-s^2/2}, \quad s \in \mathbb{R},$$

over the interval $[-5, 5]$.

2.6. **Getting help.** You can type

```
help polar
```

to learn more about any particular command. MATLAB has extensive documentation built-in, and there's lots of information available online.

3. Generating random numbers

Computers generate *pseudorandom numbers*, i.e. deterministic (entirely predictable) sequences which mimic the statistical properties of random numbers. Speaking informally, however, I shall often refer to "random numbers" when, strictly speaking, "pseudorandom numbers" would be the correct term. At a deeper level, one might question whether anything is truly random, but these (unsolved) philosophical problems need not concern at this stage.

We shall first introduce the extremely important `rand` and `randn` functions. Despite their almost identical names, they are **very** different, as we shall see, and should **not** be confused. The `rand` function generates uniformly distributed numbers on the interval $[0, 1]$, whilst the `randn` function generates *normally distributed*, or *Gaussian* random numbers. In financial applications, `randn` is extremely important.

Our first code generating random numbers can be typed in as a program, using the `create script` facility, or just entered in the command window.

The function we shall use is `rand(m,n)`, which produces an $m \times n$ matrix of pseudorandom numbers, uniformly distributed in the interval $[0, 1]$.

Example 3.1. *Uniformly distributed random numbers and histograms:*

```
N = 10000;
v=rand(N,1);
nbins = 20;
hist(v,nbins);
```

Here MATLAB has divided the interval $[0,1]$ *into* 20 *equal subintervals, i.e.*

$$[0,0.05],[0.05,0.1],[0.1,0.15],\ldots,[0.90,0.95],[0.95,1],$$

and has simply drawn a bar chart: the height of the bar for the interval $[0,0.05]$ *is the number of elements of the vector* `v` *which lie in the interval* $[0.0.05]$*, and similarly for the other sub-intervals.*

Exercise 3.1. *Now experiment with this code: change* `N` *and* `nbins`*.*

Example 3.2. *Gaussian random numbers and histograms:*

```
N = 100000;
v=randn(N,1);
nbins = 50;
hist(v,nbins);
```

Observe the obvious difference between this bell-shaped distribution and the histogram for the uniform distribution.

Exercise 3.2. *Now experiment with this code: change* `N` *and* `nbins`*. What happens for large* `N` *and* `nbins`*?*

As we have seen, MATLAB can easily construct histograms for Gaussian (i.e. normal) pseudorandom numbers. As `N` and `nbins` tend to infinity, the histogram converges to a curve, which is called the *probability density function* (PDF). The formula for this curve is

$$p(s) = (2\pi)^{-1/2} e^{-s^2/2}, \qquad \text{for } s \in \mathbb{R},$$

and the crucial properties of the PDF are

$$\mathbb{P}(a < Z < b) = \int_a^b p(s)\, ds$$

and

$$\mathbb{E}f(Z) = \int_{-\infty}^{\infty} f(s)p(s)\, ds.$$

4. Brownian Motion

The mathematics of Brownian motion is covered in my Mathematical Methods lectures. However, it is possible to obtain a good feel for Brownian motion using some simple MATLAB examples.

Our first example generates discrete Brownian motion. Mathematically, we're generating a *random function* $W : [0,\infty) \to \mathbb{R}$ using the equation

$$W(kh) = \sqrt{h}\,(Z_1 + Z_2 + \cdots + Z_k)\,, \qquad \text{for } k = 1,2,\ldots,$$

where $h > 0$ is a positive time step and $Z_1, Z_2, \ldots, Z_k$ are independent $N(0,1)$ random variables. Another way to write this is that

$$W(kh) = V_1 + V_2 + \cdots + V_k,$$

where $V_1, V_2, \ldots$ are independent $N(0,h)$ random variables. The MATLAB function `cumsum` calculates this cumulative sum in one line:

Example 4.1. `T=1; N=10000; dt = T/N; dW=sqrt(dt)*randn(1,N); plot(cumsum(dW))`
Now play with this code, changing `T` *and* `N`.

Example 4.2. *In this example we illustrate Itô's rules computationally. It's almost exactly the same code as before, but this time calculates the cumulative sum*

$$V_1^2 + V_2^2 + \cdots + V_k^2,$$

where $V_1, V_2, \ldots$ *are independent* $N(0,h)$ *random variables.*

```
T=1; N=10000; dt = T/N; dW=sqrt(dt)*randn(1,N); plot(cumsum(dW.^2))
```

Again, play with this code, changing `T` *and* `N`.

Exercise 4.1. *Now modify the last example:*

```
T=1; N=100000; dt = T/N; dW=sqrt(dt)*randn(1,N); plot(cumsum(dW.^3))
```

followed by

```
T=1; N=100000; dt = T/N; dW=sqrt(dt)*randn(1,N); plot(cumsum(dW.^4))
```

and

```
T=1; N=100000; dt = T/N; dW=sqrt(dt)*randn(1,N); plot(cumsum(dW.^5))
```

Try to explain this behaviour once we have covered stochastic calculus in lectures.

Those of you who have been reading notes carefully will remember the axiom that $W(0) = 0$. This is slightly more fiddly to incorporate:

Example 4.3.
```
T = 1; N = 10000; dt = T/N;
dW = sqrt(dt)*randn(1,N); plot([0:dt:T],[0,cumsum(dW)])
```

Example 4.4. *We can also use* `cumsum` *to generate many Brownian sample paths. Note the use of* `hold on`*, which keeps the previous graph on screen.*

```
T = 1; N = 500; dt = T/N;
nsamples = 10;
hold on
for k=1:nsamples
  dW = sqrt(dt)*randn(1,N); plot([0:dt:T],[0,cumsum(dW)])
end
```

Exercise 4.2. *Increase* `nsamples` *in the last example. What do you see?*

4.1. **Simple Random Walk.** A simple random walk is given by

$$S_n = X_1 + X_2 + \cdots + X_n,$$

where $X_1, X_2, \ldots$ is a sequence of independent (Bernoulli) random variables satisfying $\mathbb{P}(X_k = \pm 1) = 1/2$, for all k. This is the discrete time analogue of Brownian motion and is also easily generated using MATLAB.

```
n = 10^5;
a = rand(n,1);
b=(a > 0.5);
du=2*b-1;
plot(cumsum(du/sqrt(n)))
```

Exercise 4.3. *Explain how the previous example works.*

To make repeated experiments slightly easier, we can condense this code into one line:

```
n = 10^5; du=2*(rand(n,1)>0.5)-1; plot(cumsum(du/sqrt(n)))
```

You will see that simple random walk looks very like Brownian motion, so it should not surprise you to learn that Brownian motion is the limit of simple random walk as the number of steps tends to infinity: this is the Central Limit Theorem once again.

Exercise 4.4. *What happens if we replace the final plot command as follows?*

```
plot(cumsum(du.^2))
```

4.2. **Geometric Brownian Motion (GBM).** The idea that it can be useful to model asset prices using random functions was both surprising and implausible when Louis Bachelier first suggested Brownian motion in his thesis in 1900. There is an excellent translation of his pioneering work in *Louis Bachelier's Theory of Speculation: The Origins of Modern Finance*, by M. Davis and A. Etheridge. However, as you have already seen, a Brownian motion can be both positive and negative, whilst a share price can only be positive, so Brownian motion isn't quite suitable as a mathematical model for share prices. Its modern replacement is to take the exponential, and the result is called *Geometric Brownian Motion* (GBM), although exponential Brownian motion would be a better name. In other words, the most common mathematical model in modern finance is given by

$$S(t) = S(0)e^{\mu t + \sigma W(t)}, \qquad \text{for } t > 0,$$

where $\mu \in \mathbb{R}$ is called the *drift* and σ is called the *volatility*.

Example 4.5. *Generating GBM the quick way:*

```
T = 1; N = 500; dt = T/N;
t = dt:dt:T;
dW = sqrt(dt)*randn(1,N);
mu = 0.1; sigma = 0.01;
plot(t,exp(mu*t + sigma*cumsum(dW)))
```

Example 4.6. *Generating GBM, being careful to ensure $W(0) = 0$:*

```
T = 1; N = 500; dt = T/N;
t = 0:dt:T;
dW = sqrt(dt)*randn(1,N);
mu = 0.1; sigma = 0.01;
plot(t,exp(mu*t + sigma*[0,cumsum(dW)]))
```

Exercise 4.5. *Now experiment by increasing and decreasing the volatility* `sigma`.

In mathematical finance, we cannot predict the future, but we estimate general future behaviour, albeit approximately. For this we need to generate several Brownian motion sample paths, i.e. several possible futures for our share. The key command will be `randn(M,N)`, which generates an $M \times N$ matrix of independent Gaussian random numbers, all of which are $N(0,1)$. We now need to tell the `cumsum` function to cumulatively sum along each row, and this is slightly more tricky.

Example 4.7. *Generating several GBM sample paths:*

```
T = 1; N = 500; dt = T/N;
t = 0:dt:T;
M=10;

dW = sqrt(dt)*randn(M,N);

mu = 0.1; sigma = 0.01;
S = exp(mu*ones(M,1)*t + sigma*[zeros(M,1), cumsum(dW,2)]);
plot(t,S)
```

Here the MATLAB function `ones(p,q)` creates a $p \times q$ matrix of ones, whilst `zeros(p,q)` creates a $p \times q$ matrix of zeros. The matrix product `ones(M,1)*t` is a simple way to create an $M \times N$ matrix whose every row is a copy of `t`.

Exercise 4.6. *Experiment with various values of the drift and volatility.*

Exercise 4.7. *Copy the use of the* `cumsum` *function in Example 4.7 to avoid the* `for` *loop in Example 4.4.*

4.3. **The Central Limit Theorem.** Where does the Gaussian distribution come from? Why does it occur in so many statistical applications? It turns out that averages of random variables are often well approximated by Gaussian random variables, if the random variables are not too wild, and this important theorem is called the *Central Limit Theorem*. The next example illustrates the Central Limit Theorem, and shows that averages of independent, uniformly distributed random variables converge to the Gaussian distribution.

Example 4.8. *This program illustrates the Central Limit Theorem: suitably scaled averages of uniformly distributed random variables look Gaussian, or normally distributed. First we create a* 20×10000 *matrix of pseudorandom numbers uniformly distributed on the interval* $[0, 1]$*, using the* `rand` *functions. We then sum every column of this matrix and divide by* $\sqrt{20}$*.*

```
m = 20;
n = 10000;
v = rand(m,n);
%
% We now sum each column of this matrix, divide by sqrt(m)
% and histogram the new sequence
%
nbins = 20
w = sum(v)/sqrt(m);
hist(w,nbins);
```

Exercise 4.8. *Play with the constants* `m` *and* `n` *in the last example.*

4.4. **Gaussian Details.** The MATLAB `randn` command generates Gaussian pseudorandom numbers with mean zero and variance one; we write this $N(0, 1)$, and such random variables are said to be *normalized Gaussian*, or *standard normal*. If Z is a normalized Gaussian random variable, then the standard notation to indicate this is $Z \sim N(0, 1)$, where "$\sim$" means "is distributed as". We can easily demonstrate these properties in MATLAB:

Example 4.9. *Here we generate n normalized Gaussian pseudorandom numbers $Z_1, \ldots, Z_n$, and then calculate their* sample mean

$$\widehat{\mu} = \frac{1}{n} \sum_{k=1}^{n} Z_k$$

and their sample variance

$$\widehat{\sigma}^2 = \frac{1}{n} \sum_{k=1}^{n} Z_k^2,$$

as follows.

```
n=10000;
Z=randn(n,1);
mean(Z)
mean(Z.^2)
```

Experiment with this code, increasing n to 10^6, say.

Obviously not all random variables have mean zero and unit variance, but it's simple to generate Gaussian random variables with any given mean μ and variance σ^2. Specifically, if $Z \sim N(0,1)$, then $W = \mu + \sigma Z \sim N(\mu, \sigma^2)$. It's easy to illustrate this in MATLAB.

Example 4.10. *Here we generate n normalized Gaussian pseudorandom numbers $Z_1, \ldots, Z_n$, to represent a normalized Gaussian random variable $Z \sim N(0,1)$. We then define $W = \mu + \sigma Z$, and generate the corresponding pseudorandom $W_1, \ldots, W_n$, finding their sample mean and variance*

$$\widehat{\mu} = \frac{1}{n} \sum_{k=1}^{n} Z_k$$

and their sample variance

$$\widehat{\sigma}^2 = \frac{1}{n} \sum_{k=1}^{n} (Z_k - \widehat{\mu})^2 ,$$

as follows.

```
n=10000;
Z=randn(n,1);
mu = 1; sigma = 0.2;
W = mu + sigma*Z;
mu_hat = mean(W)
sigma_hat = sqrt(mean((W-mu_hat).^2))
```

Experiment with this code, increasing n to 10^6, say.

For reference, the PDF for a $N(0, \sigma^2)$ random variable is given by

$$p(s) = (2\pi\sigma^2)^{-1/2} e^{-s^2/(2\sigma^2)}, \qquad s \in \mathbb{R}.$$

Exercise 4.9. *What the PDF for a $N(\mu, \sigma^2)$ random variable?*

Here is a much longer exercise on the use of simulation. This is obviously not a financial example, but the underlying phenomenon (i.e. random variables with identical means but different variances) is highly relevant to portfolio management.

Exercise 4.10. *The pupils of two groups of* 10000 *students, A and B say, compete for a new mathematical scholarship by taking an examination. The marks for pupils in Group A are approximately normally distributed* $N(\mu, \sigma^2)$*, whilst the marks for pupils in Group B are approximately normally distributed* $N(\mu, (1+\delta)^2\sigma^2)$*, where* δ *is positive. The scholarship providers decide that to award scholarships to all students whose marks exceed* $\mu + T\sigma$*. Further, they provide special "star scholarships" to those whose marks exceed* $\mu + (T + 0.5)\sigma$*.*

(a) Suppose $\mu = 100$*,* $\sigma = 10$*,* $\delta = 0.1$ *and* $T = 2.5$*. It transpires that* 60 *of the* 10000 *pupils of Group A win scholarships, compared to* 110 *of the* 10000 *pupils of Group B. Moreover,* 13 *Group A pupils win star scholarships, compared to* 32 *Group B pupils. A truculent local politician is angered by this result, stating "This is clearly discrimination! Since the average marks of the groups are identical, the same proportion should win a scholarship. It's even more egregious discrimination for star scholarships!" Is the politician justified? Justify your answer by using the MATLAB* `randn` *function to simulate behaviour. [Hint: you don't need to use a relatively small sample of* 10^4 *pupils in your simulation.]*

(b) The scholarship provider deems any mark lower than $\mu - T\sigma$ *to be a failure. The truculent politician makes a new discovery: "I have found that* 62 *pupils in Group A failed the examination, whilst* 120 *pupils in Group B failed. This proves that Group B is no better than Group A! How, therefore, can we trust the claim that Group B should be awarded double the proportion of scholarships?" Is the politician justified? Once again, justify your answer by using the MATLAB* `randn` *function to simulate behaviour.*

(c) Use MATLAB to discover the behaviour of

$$\frac{\mathbb{P}\left(\textit{Pupil in Group B wins scholarship}\right)}{\mathbb{P}\left(\textit{Pupil in Group A wins scholarship}\right)}$$

as T *grows.*

5. Some Finance

Now let's discuss a financial application. We shall use *Monte Carlo simulation.*[1] You can find a full mathematical treatment in my notes for Mathematical Methods, but we really only need the basics here. We shall assume that our share price $S(t)$ is a random variable given by the following formula

$$S(t) = S(0)e^{(r-\sigma^2/2)t+\sigma\sqrt{t}Z}, \qquad \text{for } t > 0,$$

where Z is a standard Gaussian random variable, $S(0) = 42$, $r = 0.1$ and $\sigma = 0.2$. These parameters were fairly typical for the NYSE in the 1990s, and this example was taken from *Options, Futures and Other Derivative Securities*, by J. C. Hull.

We cannot predict the future price $S(T)$ of our stock at time T, but we can approximate the *distribution* of its possible values. In other words, we can predict the likely behaviour of our asset in many possible futures, although its value in our future sadly remains a mystery.

Example 5.1. *Predicting many possible futures at expiry time T:*

```
S0 = 42;
r = 0.1;
T = 0.5;
sigma = 0.2;
N = 100000;
%
% generate asset prices at expiry
%
Z = randn(N,1);
ST = S0*exp( (r-(sigma^2)/2)*T + sigma*sqrt(T)*Z );
%
% display histogram of possible prices at expiry
%
nbins=40;
hist(ST,nbins);
```

Exercise 5.1. *Try various values of* `N`, `sigma`, `T` *and* `nbins` *in the previous example. What happens for, say,* `sigma=20`*?*

Once we know how to generate likely future prices in this way, we can actually price a Euro put option: let us suppose we own the share already and wish to insure ourselves against a decrease in its value over the next 6 months. Specifically, we wish to buy the right, but not the obligation, to sell the share at the *exercise price* $K = \$40$ at $T = 0.5$. Such a contract is called a European put option, with exercise price (or *strike price*) K and expiry time $T = 0.5$. Obviously we want to

[1]This name originated with the brilliant mathematician John von Neumann, during his work on the Manhattan Project, the secret project building the first American atomic bombs during World War II. In the first Monte Carlo simulation, the sample paths were those of neutrons passing through Uranium, the aim being to estimate the mass of the Uranium isotope U235 required for a successful fission bomb. The American team used some of the first computers (more like programmable calculators, by our standards) to estimate some 64 kg of U235 would be sufficient, which was achievable using the cumbersome technology required to separate the 0.07% of U235 from common Uranium ore; they were correct in their estimates. The German team, led by Werner Heisenberg, had neither computers nor simulation. Heisenberg estimated a 1000 kg of U235 would be required, and therefore gave up, ending the German atomic bomb project.

compute the fair price of such a contract. Now, if $S(T) \geq K$, then the option is worth nothing at expiry; there is no value in being able to sell a share for K if it's actually worth more! In contrast, if $S(T) < K$, then we have made a profit of $K - S(T)$. If we take the view that the fair price at expiry should be the average value of $\max\{K - S(T), 0\}$, then we can easily compute this using the vector `ST` of possible expiry prices calculated in the previous example. Specifically, we compute the average

```
tput = sum(max(K-ST,0.0)/N;
```

To complete the pricing of this option, we need to understand the time value of money: we shall assume that we can borrow and save at the *risk-free* rate r. Thus, to obtain 1 at a time t in the future, we need only invest $\$\exp(-rt)$ now. In other words, the *discounted future expected price* of the European put option is given by

```
fput = exp(-r*T)*sum(max(K-ST,0.0)/N;
```

Finally, here is a summary of all of the above.

Example 5.2. *Using Monte Carlo simulation to approximate the value of the European put option of Example 11.6 of Hull:*

```
%
% These are the parameters chosen in Example 11.6 of
% OPTIONS, FUTURES AND OTHER DERIVATIVES,
% by John C. Hull (Prentice Hall, 4th edn, 2000)
%
%% initial stock price
S0 = 42;
% unit of time = year
% continuous compounding risk-free rate
%
r = 0.1;
% exercise price
K = 40;
% time to expiration in years
T = 0.5;
% volatility
sigma = 0.2;
% generate asset prices at expiry
N=10000;
Z = randn(N,1);
ST = S0*exp( (r-(sigma^2)/2)*T + sigma*sqrt(T)*Z );
% calculate put contract values at expiry
fput = max(K - ST,0.0);
% average put values at expiry and discount to present
mc_put = exp(-r*T)*sum(fput)/N
```

Exercise 5.2. *Modify this example to calculate the Monte Carlo approximation for a European call, for which the contract value at expiry is given by*

```
max(ST - K, 0)
```

Exercise 5.3. *How can we check our Monte Carlo approximations to the prices of European calls and puts? For this, you will need to create a script to calculate the*

cumulative distribution function for the normalized Gaussian distribution, that is,

$$\Phi(x) = \mathbb{P}(Z \leq x) = \int_{-\infty}^{x} (2\pi)^{-1/2} e^{-s^2/2}\, ds, \qquad \text{for } x \in \mathbb{R}, \tag{1}$$

where $Z \sim N(0,1)$. The following code will produce a suitable script, which you should save in a file called `phi.m`.

```
function Y = Phi(t)
Y = 0.5*(1.0 + erf(t/sqrt(2)));
```

To calculate analytic values of a put, you can use the code:

```
% calculate analytic value of put contract
wK = (log(K/S0) - (r - (sigma^2)/2)*T)/(sigma*sqrt(T));
a_put = K*exp(-r*T)*Phi(wK) - S0*Phi(wK - sigma*sqrt(T))
```

For the European call, you should use put–call parity: given European Put and Call options, each with exercise price K and expiry time T, their prices satisfy

$$f_C(S,t) - f_P(S,t) = S - Ke^{-r\tau}, \qquad \text{for } S \in \mathbb{R} \text{ and } 0 \leq t \leq T, \tag{2}$$

where $\tau = T - t$, the time-to-expiry. Use put–call parity to calculate the analytic call price from the put price.

Example 5.3. *There is a very useful MATLAB trick when dealing with digital options. Type the following code.*

```
v = [-3 -2 -1 0 1 2 3]
(v >= 0);
```

Thus (...) generates a vector (or matrix) of zeros and ones, with ones where the condition in brackets is true.

Example 5.4. *In this example, we first generate a matrix of random numbers uniformly distributed on the interval $[0,1]$, using rand, following which we transform this into a matrix of random ± 1 entries.*

```
%
% m = 10, n = 5
%
A = rand(m, n)
B = (A > 0.5)
C = 2*B-1
```

Exercise 5.4. *Modify the code to calculate the Monte Carlo approximation to a digital call, for which the contract value at expiry is given by*

```
(ST > K);
```

6. Least Squares fitting

Suppose we are given N points (x_k, y_k), for $k = 1, 2, \ldots, N$, which lie approximately on a line. How should we compute the line?

If the points lay exactly on a line, then we simply solve the linear system

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix} \mathbf{c} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix},$$

to give the coefficients $\mathbf{c} = (c_1, c_2)^T$. Since the points are exactly on the line, we can even solve the linear system

$$A\mathbf{c} = \mathbf{y}$$

where A is the $N \times 2$ matrix

$$A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_N & 1 \end{pmatrix}$$

and $\mathbf{y} = (y_1, y_2, \ldots, y_N)^T \in \mathbb{R}^N$. However, when the points lie approximately, but not exactly, on a line, we cannot solve $A\mathbf{c} = \mathbf{y}$, because these N equations in 2 variables will be inconsistent.

We can however solve them approximately, and the *least squares* solution finds that vector $\mathbf{c} \in \mathbb{R}^2$ minimizing the norm $\|\mathbf{y} - A\mathbf{z}\|$, for all $\mathbf{z} \in \mathbb{R}^2$. MATLAB can handle this equally easily: we simply type

```
c = A \ y;
```

Mathematically, we first define the *Euclidean norm* by

$$\|\mathbf{v}\| = \left(\sum_{k=1}^{n} v_k^2 \right)^{1/2}, \tag{3}$$

for any vector $\mathbf{v} \in \mathbb{R}^n$. The least squares solution to the overdetermined[2] linear system $A\mathbf{x} \approx \mathbf{y}$ is that vector $\mathbf{x}^*$ minimizing the Euclidean norm $\|\mathbf{y} - A\mathbf{x}\|$. It can be shown that $\mathbf{x}^*$ satisfies the so called *normal equations*:

$$A^T A\mathbf{x}^* = A^T\mathbf{y}, \tag{4}$$

but it turns out that solving the normal equations is extremely bad in floating point arithmetic. Fortunately, MATLAB uses a far superior algorithm.

Example 6.1. *This MATLAB example generates lots of points on a line, and then perturbs them by adding some Gaussian noise, to simulate the imperfections of real data. It then computes the least squares line of best fit.*

```
%
% We first generate some
% points on a line and add some noise
%
a0=1; b0=0;
n=100; sigma=0.1;
x=randn(n,1);
y=a0*x + b0 + sigma*randn(n,1);
```

[2] I.e. more equations than unknowns.

```
%
% Here's the least squares linear fit
% to our simulated noisy data
%
A=[x ones(n,1)];
c = A\y;
%
% Now we plot the points and the fitted line.
%
plot(x,y,'o');
hold on
xx = -2.5:.01:2.5;
yy=a0*xx+b0;
zz=c(1)*xx+c(2);
plot(xx,yy,'r')
plot(xx,zz,'b')
```

Exercise 6.1. *What happens when we increase the parameter* `sigma`*?*

Exercise 6.2. *Least Squares fitting is an extremely useful technique, but it is extremely sensitive to* outliers. *Here is a MATLAB code fragment to illustrate this:*

```
%
% Now let's massively perturb one data value.
%
y(n/2)=100;
cnew=A\y;
%
% Exercise: display the new fitted line. What happens when we vary the
% value and location of the outlier?
%
```

7. General Least Squares

There is no reason to restrict ourselves to linear fits. If we wanted to fit a quadratic $p(x) = p_0 + p_1 x + p_2 x^2$ to the data $(x_1, y_1), \ldots, (x_N, y_N)$, then we can still compute the least squares solution to the overdetermined linear system

$$A\mathbf{p} \approx \mathbf{y},$$

where $\mathbf{p} = (p_0, p_1, p_2)^T \in \mathbb{R}^3$ and A is now the $N \times 3$ matrix given by

$$A = \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots \\ x_N^2 & x_N & 1 \end{pmatrix}.$$

This requires a minor modification to Example 6.1.

Example 7.1. *Generalizing Example 6.1, we generate a quadratic, perturb the quadratic by adding some Gaussian noise, and then fit a quadratic to the noisy data.*

```
%
% We first generate some
% points using the quadratic x^2 - 2x + 1 and add some noise
%
a0=1; b0=-2; c0=1;
n=100; sigma=0.1;
x=randn(n,1);
y=a0*(x.^2) + b0*x + c0 + sigma*randn(n,1);
%
% Here's the least squares quadratic fit
% to our simulated noisy data
%
A=[x.^2 x ones(n,1)];
c = A\y;
%
% Now we plot the points and the fitted quadratic
%
plot(x,y,'o');
hold on
xx = -2.5:.01:2.5;
yy=a0*(xx.^2)+b0*xx + c0;
zz=c(1)*(xx.^2)+c(2)*xx + c(3);
plot(xx,yy,'r')
plot(xx,zz,'b')
```

Exercise 7.1. *Increase* `sigma` *in the previous example, as for Example 6.1. Further, explore the effect of choosing a large negative outlier by adding the line* `y(n/2)=-10000;` *before solving for* `c`.

There is absolutely no need to restrict ourselves to polynomials. Suppose we believe that our data $(x_1, y_1), \ldots, (x_N, y_N)$ are best modelled by a function of the form

$$s(x) = c_0 \exp(-x) + c_1 \exp(-2x) + c_2 \exp(-3x).$$

We now compute the least squares solution to the overdetermined linear system $A\mathbf{p} \approx \mathbf{y}$, where $\mathbf{p} = (p_0, p_1, p_2)^T \in \mathbb{R}^3$ and

$$A = \begin{pmatrix} e^{-x_1} & e^{-2*x_1} & e^{-3x_1} \\ e^{-x_2} & e^{-2*x_2} & e^{-3x_2} \\ \vdots & \vdots & \vdots \\ e^{-x_N} & e^{-2*x_N} & e^{-3x_N} \end{pmatrix}.$$

Example 7.2. `%`

```
% We first generate some
% points using the function
%   a0*exp(-x) + b0*exp(-2*x) + c0*exp(-3*x)
% and add some noise
%
a0=1; b0=-2; c0=1;
n=100; sigma=0.1;
x=randn(n,1);
```

```
y=a0*exp(-x) + b0*exp(-2*x) + c0*exp(-3*x) + sigma*randn(n,1);
%
% Here's the least squares fit
% to our simulated noisy data
%
A=[exp(-x) exp(-2*x) exp(-3*x)];
c = A\y;
%
% Now we plot the points and the fitted quadratic
%
plot(x,y,'o');
hold on
xx = -2.5:.01:2.5;
yy=a0*exp(-xx)+b0*exp(-2*xx) + c0*exp(-3*xx);
zz=c(1)*exp(-xx)+c(2)*exp(-2*xx) + c(3)*exp(-3*xx);
plot(xx,yy,'r')
plot(xx,zz,'b')
```

8. Warning Examples

In the 1960s, mainframe computers became much more widely available in universities and industry, and it rapidly became obvious that it was necessary to provide software libraries to solve common numerical problems, such as the least squares solution of linear systems. This was a golden age for the new discipline of Numerical Analysis, straddling the boundaries of pure mathematics, applied mathematics and computer science. Universities and national research centres provided this software, and three of the pioneering groups were here in Britain: the National Physical Laboratory, in Teddington, the Atomic Energy Research Establishment, near Oxford, and the Numerical Algorithms Group (NAG), in Oxford. In the late 1980s, all of this code was incorporated into MATLAB. The great advantage of this is that the numerical methods chosen by MATLAB are excellent and extremely well tested. However any method can be defeated by a sufficiently nasty problem, so you should not become complacent. The following matrix is usually called the *Hilbert matrix*, and seems quite harmless on first contact: it is the $n \times n$ matrix $H^{(n)}$ whose elements are given by the simple formula

$$H^{(n)}_{jk} = \frac{1}{j+k+1}, \qquad 1 \le j,k \le n.$$

MATLAB knows about the Hilbert matrix: you can generate the 20×20 Hilbert matrix using the command `A = hilb(20);`. The Hilbert matrix is notoriously *ill-conditioned*, and the practical consequence of this property is shown here:

Example 8.1. `%`

```
% A is the n x n Hilbert matrix
%
n = 15;
A = hilb(n);
%
%
%
v = [1:n]';
w = A * v;
%
% If we now solve w = A*vnew using vnew = A \ w,
% then we should find that vnew is the vector v.
% Unfortunately this is NOT so . . .
%
vnew = A \ w
```

Exercise 8.1. *Try increasing* `n` *in the previous example.*

8.1. **Floating Point Warnings.** Computers use *floating point arithmetic*. You shouldn't worry about this too much, because the relative error in any arithmetic operation is roughly 10^{-16}, and we shall make this more precise below. However, it is *not* the same as real arithmetic. In particular, errors can be greatly magnified and the order of evaluation can affect results. For example, floating point addition is commutative, but not associative: $a + (b + c) \neq (a + b) + c$, in general.

In this section, we want to see the full form of numbers, and not just the first few decimal places. To do this, use the MATLAB command `format long`.

Example 8.2. *Prove that*

$$\frac{1-\cos x}{x^2} = \frac{\sin^2 x}{x^2\,(1+\cos x)}.$$

Let's check this identity in MATLAB:

```
for k=1:8, x=10^(-k); x^(-2)*(1-cos(x)), end
for k=1:8, x=10^(-k); x^(-2)*sin(x)^2/(1+cos(x)), end
```

Explain these calculations. Which is closer to the truth?

We can also avoid using loops using MATLAB's dot notation for pointwise operations. I have omitted colons in the next example to illustrate this:

```
x=10.^(-[1:8])
1-cos(x)
(sin(x).^2) ./ (1+cos(x))
```

Example 8.3. *Prove that*

$$\sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1}+\sqrt{x}},$$

for $x > 0$. Now explain what happens when we try these algebraically equal expressions in MATLAB:

```
x=123456789012345;
a=sqrt(x+1)-sqrt(x)
a =  4.65661287307739e-08
b=1/(sqrt(x+1) + sqrt(x))
b =  4.50000002025000e-08
```

Which is correct?

Example 8.4. *You should know from calculus that*

$$\exp(z) = \sum_{k=0}^{\infty} \frac{z^k}{k!},$$

for any $z \in \mathbb{C}$. Let's test this.

```
x=2; S=1; N=20; for k=1:N, S=S+(x^k)/factorial(k); end
exp(x)
S
```

Now replace `x=2` *by* `x=-20`*. What has happened? What happens if we increase* `N`*?*

Example 8.5. *The roots of the quadratic equation*

$$x^2 + bx + c = 0$$

are given by

$$x_1 = \frac{-b+\sqrt{b^2-4c}}{2} \quad \textit{and} \quad x_2 = \frac{-b-\sqrt{b^2-4c}}{2}.$$

Use these expressions to find the roots when `b = 1111111; c=1`*. Now the identity*

$$x^2 + bx + c = (x-x_1)(x-x_2)$$

implies that $c = x_1 x_2$. Is c/x_2 equal to your computed value of x_1? What has occurred?

8.2. **Machine Precision.** The smallest number ϵ such that the computer can distinguish $1+\epsilon$ from 1 is called the *machine epsilon*. Computers use base 2, so finding ϵ reduces to finding the largest positive integer k for which the computer can distinguish between $1+2^{-k}$ and 1. We can easily find this using the following MATLAB code:

Example 8.6. *The following code generates a* 55×4 *matrix for which row* k *contains* k, 2^{-k}, $1+2^{-k}$, *and finally a true/false value (i.e.* 1 *or* 0*) depending on whether MATLAB believes that* $1+2^{-k}$ *exceeds* 1.

```
x=zeros(55,0);
for k=1:55
    x(k,1)=k; x(k,2)=2^(-k); x(k,3) = 1+x(k,2);
    x(k,4) = (x(k,3) > 1); % equals 1 if x(k,3) > 1 else 0
end
```

You should find that $\epsilon = 2^{-52} = 16^{-13} = 2.22044604925031 \times 10^{-16}$. *This will be the case on almost all computers, which now follow the IEEE 754 standard specifying the details of floating point arithmetic, without which our machines' computations would be far more dubious.*

Exercise 8.2. *Construct values of* a, b *and* c *for which* $a+(b+c) \neq (a+b)+c$, *implying that floating point arithmetic is not associative.*

9. Recursion and Sudoku

This section is really just for fun, but it also gives me a chance to display some other features of the MATLAB language, of which the most important is recursion: a MATLAB function can call itself.

Sudoku is a popular puzzle in which a 9×9 matrix is further sub-divided into 9 3×3 submatrices. The matrix can only contain the integers $1, \ldots, 9$, but each row, each column, and each of the 9 3×3 submatrices, must contain all 9 digits. Initially, the solver is faced with some given values, the remainder being blank. Here's a simple example:

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Here's a harder example:

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5

It's not too difficult to write a MATLAB program which can solve any Sudoku. You can download a simple Sudoku solver (`sud.m`) from my office machine:

`http://econ109.econ.bbk.ac.uk/brad/CTFE/matlab_code/sudoku/`

Here's the MATLAB code for the solver:

```
function A = sud(A)

global cc
cc = cc+1;

% find all empty cells
[yy xx]=find(A==0);
if length(xx)==0
disp('solution')
disp(A);
    return
end
x=xx(1);
y=yy(1);
for i=1:9 % try 1 to 9
    % compute the 3 x 3 block containing this element
    y1=1+3*floor((y-1)/3); % find 3x3 block
    x1=1+3*floor((x-1)/3);
    % check if i is in this element's row, column or 3 x 3 block
```

```
    if ~( any(A(y,: )==i) | any(A(:,x)==i) | any(any(A(y1:y1+2,x1:x1+2)==i)) )
        Atemp=A;
        Atemp(y,x)=i;
        % recursively call this function
        Atemp=sud(Atemp);
        if all(all(Atemp))
            A=Atemp; % ... the solution
            return;  % and that's it
        end
    end
end
```

Download and save this file as `sud.m`. You can try the solver with the following example:

```
%
% Here's the initial Sudoku; zeros indicate blanks.
%
M0 = [
0 4 0 0 0 0 0 6 8
7 0 0 0 0 5 3 0 0
0 0 9 0 2 0 0 0 0
3 0 0 5 0 0 0 0 7
0 0 1 2 6 4 9 0 0
2 0 0 0 0 7 0 0 6
0 0 0 0 5 0 7 0 0
0 0 6 3 0 0 0 0 1
4 8 0 0 0 0 0 3 0];

M0
M = M0;
%
% cc counts the number of calls to sud, so it is one measure
% of Sudoku difficulty.
%
global cc = 0;
sud(M);
cc
```

Exercise 9.1. *Solve the first two Sudokus using* `sud.m`.

Exercise 9.2. *How does* `sud.m` *work?*

Department of Economics, Mathematics and Statistics, Birkbeck College, University of London, Malet Street, London WC1E 7HX, England
Email address: b.baxter@bbk.ac.uk