Y1 Computing - Introduction to MATLAB

Getting started with MATLAB:

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualisation, data analysis, and numerical computation. You can use MATLAB in a wide range of applications, including signal and image processing, numerical modelling, etc. One of the many things you will like about MATLAB (and that distinguishes it from many other computer programming systems, such as C++ for instance) is that you can use it interactively. This means you type some commands at the special MATLAB prompt and get results immediately. The problems solved in this way can be very simple, like finding a square root, or very complicated, like finding the solution to a system of differential equations.

In order to start using MATLAB you must become familiar with a few basic techniques.

Start MATLAB from the desktop (Start \rightarrow All Programs \rightarrow School of Physics \rightarrow MATLAB \rightarrow MATLABR2013a). Click on the **Help** menu and select **Examples**, then click on **Getting Started**.

By doing so you can access three video tutorials that will help you familiarising with the MATLAB environment:

- 1. Getting Started with MATLAB
- 2. Working in The Development Environment
- 3. Writing a MATLAB Program

Now connect your earphones to the desktop computer and run the video tutorials. Take your time to get acquainted with the desktop environment. At this stage you are not required to learn all the commands used in the videos, but to be able to execute commands in the command window and to **create a script file**, **save** it and **run it**.

MATLAB has an extremely useful online Help system. At any time you can access the Help system by clicking on the menu bar or by typing doc + [return] in the command window. You can ask help on a specific MATLAB command (e.g. plot) by typing doc plot + [return], or simply by using the search toolbar in the help window.

If you are looking for information on a specific function you can click on the fx symbol near the >> prompt in the command window, and browse through several MATLAB built-in functions.



Where to find the information you need to complete the exercises / additional reading:



1. the short notes describing each exercise will point you to the relevant MATLAB functions and commands,

2. detailed explanations on how to use MATLAB commands can be found in the MATLAB online help (see above). The online help is exhaustive but could be cumbersome at times: do not hesitate to ask a demonstrator for guidance!

3. the University library gives open electronic access to the book "Essential Matlab for Engineers and Scientists", which you can consult/download by connecting to this webpage: <u>http://www.sciencedirect.com/science/book/9780123748836</u> or by searching the book in the main Library search engine <u>http://findit.bham.ac.uk/</u>.

You are encouraged to read through the relevant Chapters as suggested in the notes describing each exercise. For instance, you may find reading Chapter 1 a useful introduction to the main features of MATLAB, and to few basic commands (which are also explained in the video tutorials described above).

You can now start with the exercises, which will show you how to:

- *Exercise 1:* use MATLAB as a calculator, define and handle arrays and matrices 5%
- *Exercise 2:* solve linear systems of equations 5%

Exercise 3: create 2-D plots and export figures, create 3-D plots 5%

- Exercise 4: use arrays and save variables 5%
- Exercise 5: define and use your own functions 5%
- *Exercise 6:* fit functions to data 5%
- Exercise 7: numerical integration 10%
- *Exercise 8:* perform symbolic computations 5%

Exercise 9: animate plots 10%

Exercise 10: solve differential equations 10%

Exercise 11: numerical errors and interpolation 10%

Exercise 12: generate sound with MATLAB 25%

Assessment Deadlines:

The deadlines for getting exercises marked off are:

Exercises 1-3:Week 3Exercises 4-8:Week 7Exercises 9-11:Week 11Exercises 12:submitted through CANVAS by the end of Week 11

To get one of exercises (1-11) marked, save it as a MATLAB script (.m file) and show it to a demonstrator. Exercise 12 has to be submitted through CANVAS by the end of Week 11.

Exercise 1 MATLAB basics

This exercise will test your ability to use MATLAB as a "calculator", to define variables and perform simple operations with arrays and matrices.

Part 1 - MATLAB as a calculator

A. Arithmetic operators and elementary math functions are described in the help window *Matlab, Mathematics, Elementary Maths.*

As explained in the video tutorials, remember that when you work with arrays, you can perform element-by-element operations by using the following notation:

- + Addition
- Subtraction
- .* Element-by-element multiplication
- ./ Element-by-element division
- .^ Element-by-element power

e.g. a=[1 2 3]; a.^2 gives [1 4 9]

Note that in MATLAB i and j represent the imaginary unit, and pi represents π . Additional pre-defined constants are found by searching "*math constants*" in the help search toolbar (have a look at the meaning of Inf, NaN and eps).

Evaluate the following expressions (save the list of commands in a script file):

$$2\pi, \frac{\pi}{2}, \pi^2, \sqrt{\pi},$$

 $i^2, \sqrt{-1}, \log_e 2, \log_{10} 100,$
 $(\sin(\pi/4) + \cos(\pi/4))^2,$
 $e^{i\pi} + 1$

Part 2 - Variables

A. In this exercise, you are required to evaluate the total energy of an object with rest mass m and velocity v using the formula here below:

$$E = \frac{mc^2}{\sqrt{1 - \frac{v^2}{c^2}}}$$

where c is the velocity of light. First, define variables m, v and c with the values 1 Kg, 1 x 10^8 m/s and 3 x 10^8 m/s, then compute E.

B. Now define v as an array containing values from 0 to c in steps of 0.1*c, and compute the corresponding energies as an array.

Part 3 - Matrices

Matrices are two-dimensional arrays of data in the form:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}$$

Note that the notation implies that $a_{i,j}$ is the element of A which is stored in the i-th row and j-th column. The number of rows *n* need not be the same as the number of columns *m*. We say that such a matrix A is of dimension $n \ge m$. When n = 1 we have a column vector and when m = 1 we have a row vector. When n = m = 1, A has a single entry, and is commonly known as a scalar.

Like an array, a matrix can be created in MATLAB using any name starting with a letter. Matrices are created with a syntax similar to the one used to define arrays. For example, the command:

A = [[1,2,3]; [4,5,6]; [7,8,10]];

puts the matrix

 $\left[\begin{array}{rrrr}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 10\end{array}\right]$

into the variable A.

A. Open the help window and go to *Matlab, Getting Started, Matrices and Arrays*. Read the page and familiarise with basic operations with matrices.

Using the function magic (search for *magic square* in the online help), define three "magic squares" of dimension 5, 10 and 15, respectively. Then, for each matrix compute the sum of diagonal elements, the sum of elements in column 1, and of elements in row 5. Verify that the characteristic sum M associated to each matrix of order n is:

$$M = \frac{n(n^2 + 1)}{2}$$

where n is the order of the matrix.

To get your exercise marked, save it as a MATLAB script (.m file) and show it to a demonstrator.

Exercise 2: a plane truss in static equilibrium solving linear equations

One of the problems encountered most frequently in scientific computation is the solution of systems of simultaneous linear equations.

With matrix notation, a system of simultaneous linear equations is written Ax = b.

In the most frequent case, when there are as many equations as unknowns, A is a given square matrix of order n, b is a given column vector of n components, and x is an unknown column vector of n components.

The solution to Ax = b can be written $x = A^{-1}b$, where A^{-1} is the inverse of A. However, in the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute A^{-1} . As an extreme but illustrative example, consider a system consisting of just one equation, such as:

$$7x = 21.$$

The best way to solve such a system is by division:

$$x = \frac{21}{7} = 3.$$

Use of the matrix inverse would lead to

$$x = 7^{-1} \times 21 = 0.142857 \times 21 = 2.99997.$$

The inverse requires more arithmetic—a division and a multiplication instead of just a division—and produces a less accurate answer (In Ex. 11 you will learn more about numerical errors). Similar considerations apply to systems of more than one equation. This is even true in the common situation where there are several systems of equations with the same matrix A but different right-hand sides b. Consequently, we shall concentrate on the direct solution of systems of equations rather than the computation of the inverse.

MATLAB has introduced nonstandard notation using backward slash and forward slash operators, "\" and "/".

If A is a matrix of any size and shape and B is a matrix with as many rows as A, then the solution to the system of simultaneous equations

AX = B

is denoted by

 $\mathsf{X}=\mathsf{A}\backslash\mathsf{B}.$

Think of this as dividing both sides of the equation by the coefficient matrix A. Because matrix multiplication is not commutative and A occurs on the left in the original equation, this is left division.

Similarly, the solution to a system with A on the right and B with as many columns as A, is obtained by right division,

XA = B,

X = B/A.

This notation applies even if A is not square, so that the number of equations is not the same as the number of unknowns.

To illustrate the general linear equation solution algorithm, consider an example of order three:

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ 6 \end{pmatrix}.$$

This, of course, represents the three simultaneous equations

$$10x_1 - 7x_2 = 7,$$

$$-3x_1 + 2x_2 + 6x_3 = 4,$$

$$5x_1 - x_2 + 5x_3 = 6.$$

A. Define the matrices A and B in a script, and solve the system of linear equations.

B. The figure below depicts a plane truss having 13 members (the numbered lines) connecting 8 joints (the numbered circles). The indicated loads, in tons, are applied at joints 2, 5, and 6, and we want to determine the resulting force on each member of the truss.



For the truss to be in static equilibrium, there must be no net force, horizontally or vertically, at any joint. Thus, we can determine the member forces by equating the horizontal forces to the left and right at each joint, and similarly equating the vertical forces upward and downward at each joint. For the eight joints, this would give 16 equations, which is more than the 13 unknown factors to be determined. For the truss to be statically determinate, that is, for there to be a unique solution, we assume that joint 1 is rigidly fixed both horizontally and vertically and that joint 8 is fixed vertically. Resolving the member forces into horizontal and vertical components and defining $\alpha = 1/\sqrt{2}$, we obtain the following system of equations for the member forces f_i :

Joint 2: $f_2 = f_6$, $f_3 = 10;$ Joint 3: $\alpha f_1 = f_4 + \alpha f_5$, $\alpha f_1 + f_3 + \alpha f_5 = 0;$ $f_4 = f_8$, Joint 4: $f_7 = 0;$ Joint 5: $\alpha f_5 + f_6 = \alpha f_9 + f_{10}$, $\alpha f_5 + f_7 + \alpha f_9 = 15;$ $f_{10} = f_{13},$ Joint 6: $f_{11} = 20;$ $f_8 + \alpha f_9 = \alpha f_{12},$ Joint 7: $\alpha f_9 + f_{11} + \alpha f_{12} = 0;$ Joint 8: $f_{13} + \alpha f_{12} = 0.$

Use matrices to solve this system of linear equations to find the vector *f* of member forces.

Tip: you can input large matrices line by line, using the following syntax: A=[1 2 3 4 5 6 7 8; 0 0 0 0 0 0 0; 0 and close the square bracket at the end of the last line, e.g.:

0 0 0 0 0 0 0 0]

When the exercise is complete save your work and show it to a demonstrator.

Exercise 3

2D and 3D plot basics

In the first part of this exercise you will learn how construct and format a 2D (x-y) plots. A description of MATLAB 2D plotting facilities can be found in the *Matlab, Getting Started, 2-D and 3-D plots*, or simply by typing doc plot in the command window. Read the page as well as the documentation about additional useful commands in the "See Also" section (bottom of the page) such as: xlim, subplot, title, ylabel, xlabel, legend, grid.

A. Create a plot defining variable x as an array with values between -50 and 50, with a step of 0.1.

Consider the following functions,

 $y1 = \sin x$

 $y2 = \sin^2 x + 2$

$$y3 = \frac{\sin x}{x} + 4$$

 $y4 = \sin x + \sin 1.2x + 6$

define y1, ..., y4 and plot them using the plot command. To hold several lines on the same figure use the hold on command. Use different colours and styles for the four lines (see the example in the help page).

Add a text title on the plot, as well as the appropriate legend. Change the x limits to be -30 and +30.

Export the figure as a pdf file from the figure "file" menu, or using the print command (e.g. print(gcf, `-dpdf', `myfile.pdf')).

B. Consider now the simple exponential function

$$y5 = 10^{-Ax}$$

and x from 0 to 2 in steps of 0.02.

Create a new figure divided in two horizontal plots (using the subplot command). Plot in the upper subplot $y_5(x)$ for A=0.5 and A=1, and in the lower subplot the same functions but

using a plot with a logarithmic y scale (see semilogy). Add legend, titles, labels, grids, and export as a pdf figure.

In the next two exercises you will learn about arrays and how to construct and format three-dimensional plots. MATLAB has a variety of functions for displaying and visualising data in 3D, either as lines or as various surfaces. Take a look at the following functions in the MATLAB help:

plot3, scatter3

and read the example on how to create and visualise surfaces in *Matlab, Getting Started, 2-D and 3-D plots*

C. The first part of the exercise is simply to plot a curve in 3D. Define z as an array between 0 and 50 with a step of 0.1, and

$$x(z) = e^{-0.03z} \sin z$$
$$y(z) = e^{-0.03z} \cos z$$

then plot the curve using plot3. Remember to add axes labels.

D. Use the command meshgrid to define a X, Y grid between -25 and 25 with a step of 0.2. Plot the surface defined by

$$Z = R^2 \sin R$$

where



$$R = \sqrt{(2X)^2 + Y^2}$$

Use the surf, view, shading, subplot and colormap command to obtain a figure similar to the one shown here.

When the exercise is complete save your work and show it to a demonstrator.

Further reading: see Chapter 7 in the e-book "Essential Matlab for Engineers and Scientists" <u>http://www.sciencedirect.com/science/article/pii/</u> B9780123748836000072#

Exercise 4: Radioactive Decay

arrays, 2D plots and saving variables

Consider the pure isotope A which decays into the unstable isotope B which in turn decays into the stable isotope C as follows:

 $A \rightarrow (T_A=60 \text{ minutes}) \rightarrow B (T_B=30 \text{ minutes}) \rightarrow C$,

where T is the half-life of the isotope.

Decay of A: At time 0 the number of atoms A is a0; at time t the number of atoms A is

$$a(t) = a_0 e^{-\lambda_A t}$$

where λ is the decay constant of A ($\lambda = \ln(2)/T$). The decay constant is defined such that $N(t) = N_0 e^{-\lambda t}$.

Decay of B: At time 0 the number of atoms B is b0=0; at time = t the number of atoms B is

$$b(t) = \frac{a_0 \lambda_A}{\lambda_B - \lambda_A} \left[e^{-\lambda_A t} - e^{-\lambda_B t} \right]$$

for t>>T_B, $b(t) \rightarrow 0$ (all atoms of B will have decayed).

Accumulation of C atoms: At time 0 the number of atoms C is c0=0; at time = t the number of atoms A and B and C is equal to the initial number of atoms a0, i.e. a0=a(t)+b(t) +c(t) and hence c(t) ca be calculated as the other quantities have been determined above. At time t the number of atoms C is

$$c(t) = \frac{a_0}{\lambda_B - \lambda_A} \left[\lambda_B \left(1 - e^{-\lambda_A t} \right) - \lambda_A \left(1 - e^{-\lambda_B t} \right) \right]$$

for t>>T_A or T_B, c(t) \rightarrow a0, and at, b(t) \rightarrow 0 i.e. all atoms of A and B will have decayed.

In this exercise, you will use equations 1,2, and 3 to create radioactive decay plots and and save in a file the variables describing the decay.

Define values for the initial number of atoms (1000) and the two half-lives (A 60 minutes and B 30 minutes). Calculate the two decay constants lambda. Hence, using arrays, calculate and plot the number of atoms of A, B and C and the total number of atoms at 5 minute intervals over a period of 500 minutes. Arrange the script so that the number of data points and the start/finish times (and hence the time interval) can be varied.

You should format your plot to look like the following.



Save in a MATLAB file (see below) the time, the number of atoms of A, B, C, and the total number of atoms.

To save variables in a file use the save command, for instance:

save(`output_var', `varA', `varB'). To load in the workspace the saved variables
simply type load `output_var'. This can be useful if you want to store variables which
are the output e.g. of lengthy calculations.

When the exercise is complete save your work and show it to a demonstrator.

Exercise 5: N slit diffraction defining custom functions in MATLAB

Although MATLAB has numerous built-in functions, you can create your own functions by storing them in separate function files.

Learn how to create functions and how to call them in a script by reading the online documentation (type doc function, or by search "function" in the help window).

A function file is similar to a script file in that it also has an .m extension. However, it differs from a script file as it communicates with the MATLAB workspace only through specially designated input and output arguments.

Functions are indispensable when it comes to breaking a problem down into manageable logical pieces. In this exercise you are requested to define a simple function, save it in a separate .m file, and call it from the main script.

The intensity distribution (I) for N slit diffraction is governed by the relationship

$$I(x) = \frac{\sin^2(\beta)}{\beta^2} \frac{\sin^2(N\gamma)}{\sin^2(\gamma)}$$

where $\beta = \pi w x / \lambda$, $\gamma = \pi s x / \lambda$ and w is the slit width, s is the slit separation, λ is the wavelength and x is the displacement from the origin.

Define a function to compute I using as arguments x, w, λ , s and N. Then, in the main script file, call the function using as input the values given below, and plot the intensity distribution against x.

A suitable set of parameters is N=4, w=1.7 10^{-4} m, s=8.5 10^{-4} m, λ =5 10^{-7} m,, x_{min}=-0.003 m, x_{max}=+0.003 m. Define x as an array of 500 points (consider using the linspace command).

You should find the following distribution:



The worksheet should be set-up so that the various parameters (N, λ , *etc.*) can be varied in order to study the different distributions.

When the exercise is complete, try changing the input parameters and describe the behaviour to a demonstrator.

Further reading: see Chapter 10 in the e-book "Essential Matlab for Engineers and Scientists" <u>http://www.sciencedirect.com/science/article/pii/B9780123748836000102</u>

Exercise 6: Least Squares Fit fitting polynomials and custom functions

The term least squares describes a frequently used approach to solving overdetermined systems of equations in an approximate sense. Instead of solving the equations exactly, we seek only to minimize the sum of the squares of the residuals. The least squares criterion has important statistical interpretations. If appropriate probabilistic assumptions about underlying error distributions are made, least squares produces what is known as the maximum-likelihood estimate of the parameters. Even if the probabilistic assumptions are not satisfied, years of experience have shown that least squares produces useful results.

In the first part of this exercise you will learn how MATLAB can be used to fit a polynomial function to experimental data.

First search MATLAB Help for information on polyfit, polyval, and errorbar.

Then

A. Define two arrays, x: [-0.55, 0.73, 1.98, 2.93, 3.93, 4.85, 6.16] and y: [2.9669, 7.7280, 25.2241, 62.8794, 134.0628, 240.7299, 483.9702].

Fit a third order polynomial to the data using MATLAB <code>polyfit</code> function. Construct a plot showing x against y (as symbols only) and x against **ax**³+**bx**²+**cx**+**d** (as line only). To plot only points consider setting the property: <code>Linestyle</code>, <code>`none'</code> while calling the <code>plot</code> function, or use the <code>scatter</code> function instead. Finally, add a second plot on the figure showing the residuals of your fit, i.e. **y-(ax**³+**bx**²+**cx**+**d**), adding error bars of value 1.5 on the y axis.

Tip: to construct the vector of error bars, you can use the command ones.

B. Least squares with an arbitrary fitting function: In the remainder of this exercise, you will learn how to fit an arbitrary function to a set of data. If you know the form of the required function you can use fminsearch to find the best coefficients. Suppose that you think that some data should conform to a relation of the form:

$$y(x) = ax + b\sin(x) + c$$

To be able to use fminsearch you need to write a MATLAB function that for given values of a, b and c, calculates the square of the disparity (in the example here below the function is called merit). Its first argument is a vector of coefficients, its last argument is a vector of the given data values. The other arguments (in this case, just one) are vectors of free variables. You can put this function into its own file or (as here) make it into a subfunction. If you put the following code into a file called fitting.m, then run it, you should see how good to fit is.

```
function fitting
```

```
% generates artificial noisy data based on the function fun (see below)
[x y]=generate data;
% Calls fminsearch: minimises the function "merit"
% Input:
% - function handle (@) (in this case @merit)
% - [] is the argument for additional options, in this case the empty arguemnt
% "[]" stands for "use default options"
% - array with initial guess of the parameters
% - x: array with independent variable
% - y: array with data
S
% Output:
% - bestparam: array of parameters that minimises the sum of squared
% differences
% - min: sum of squared differences for best-fit parameters
8
% Defines array with initial guess of the parameters
param guess=[5 0.5 50];
% Calls fminsearch
[bestparam min]=fminsearch(@merit,param guess,[],x,y);
% Evaluate the function using the best-fit parameters
yfit=fun(bestparam,x);
% Plots data and fitted function
 scatter(x,y,60,'k');
 hold on;
 plot(x,yfit,'r','LineWidth',2);
 hold off
 figure(gcf);
 title(sprintf('a= %5.3f, b= %5.3f, c= %5.3f, Residuals=%5.3f', bestparam,min ))
 ylabel('Y'); xlabel('X');
 legend('Data','Fit','Location','NorthWest');
 grid on;
end
% Functions
% A. merit: Computes merit function (sum of squared difference between data and
% fitted function)
 function sumsquares=merit(param, X,Y)
     DIFF = fun(param, X) - Y;
     SQ DIFF = DIFF.^2;
     sumsquares = sum(SQ DIFF);
 end
% B. fun: fitting function
 function Y=fun(coeff,X)
     a = coeff(1);
     b = coeff(2);
     c = coeff(3);
     Y = a * X + b * sin(X) + c;
 end
% C. generate data: generates artificial noisy data based on the function fun
  function [x y]=generate data
    x=1:0.2:15;
```

```
a=0.9; b=1.5; c=20;
err_p=0.01;
y=fun([a b c], x)+err_p*randn(1,numel(x)).*fun([a b c], x);
end
```

Note that the choice of starting values of the fitting parameters (param_guess) can affect the speed of the search or, in case of non-linear fits, the convergence.

Now it's time for you to fit a different function to some data!

Load the variables in the file $data_ex6.mat$ in your MATLAB workspace. Fit the following function to the data:

$$y(x) = a e^{-x/50} \sin(x) + b,$$

determine the best fitting parameters a and b, and plot the data and the resulting best fit.

When the exercise is complete save your work and show it to a demonstrator.

Exercise 7: Black Body Radiation

the for loop and numerical integration

Introduction to the for loop:

In the examples we have seen so far the commands are executed consecutively, working down the script from the top to the bottom. In real programs you may want to repeat the same command several times, or perform different calculations in different situations. Statements that tell the computer which command is to be executed next are fundamental building blocks for computer programs, and are called control-flow statements. In this exercise you will learn how to use the "for loop", for repeating a statement or group of statements a fixed number of times,

A simple form of such a loop is

```
for index = 1:n
statements
end
```

The statements are executed n times with index = 1,2,3,...,n. Here n is a number that must be fixed before we enter the loop. More generally, the loop may start with the line for index = j:m:k. Usually j,m and k are integers and k-j is divisible by m. The statements are executed repeatedly starting from index = j with index incremented by m each time and continuing until index = k (which is the last time round the loop). If m = 1, we can replace j:m:k in the first line of the loop by j:k. Even more generally, we can begin with the line for index = v, where v is a vector. The statements inside the loop are repeatedly executed with index set to each of the elements of v in turn. Type help for and read the manual page for the for loop.

Exercise

The radiation intensity per unit area of a black body radiator in the wavelength range λ to λ +d λ is I(λ)d λ and is governed by Planck's radiation formula

$$I(\lambda,T) = \frac{2\pi c^2 h}{\lambda^5} \left(\frac{1}{e^{\frac{h c}{\lambda k T}} - 1}\right)$$

where

h is Planck's constant	6.6260 10 ⁻³⁴ J s
k is Boltzmann's constant	1.3806 10 ⁻²³ J/K
c is the velocity of light	2.9979 10 ⁸ m/s
T is the absolute temperature	К

A. Construct a plot showing the Black Body spectra at a range of temperatures. Use a function to compute the 2-dimensional array $I(\lambda,T)$. A suitable temperature range is 3000 K to 8000 K in steps of, say, 1000 K with a wavelength range of 100 nm to 3000 nm divided into 1000 intervals.

Define an array with temperatures T, and call the function that computes the intensity inside a for loop.

You can then plot I for each value j of temperature inside the for loop (e.g. $plot(\lambda, I(:,j))$) or use a single plot command outside the loop (e.g. $plot(\lambda, I)$).

Plotting I against λ should result in a chart similar to the following:



B. Use the MATLAB function \max to compute the wavelength at the maximum intensity for each temperature. Compare the wavelength of visible light with that corresponding to the maximum radiation intensity of the Sun (whose radiation spectrum can be approximated to that of a black body of T~6000 K).

Tip: The function max(A) gives as output not only the maximum value of A, but also the index of the maximum values of A.

C. Read the online documentation on how to use the numerical integration routine trapz, and use it to estimate the area under each of the curves i.e. the total intensity.

Stefan's Constant (σ_T) can be calculated from the relationship $I_T = \sigma_T T^4$. The accepted value of Stefan's constant (σ) is 5.67 x 10-8 W/m² K⁴. Compare this value with your calculations for various temperatures and wavelength ranges. To improve the value you obtain for σ_T try increasing the upper limit of the wavelength range considered.

When the exercise is complete save your work and show it to a demonstrator.

Further reading: see Chapter 2 (in particular 2.7) in the e-book "Essential Matlab for Engineers and Scientists" <u>http://www.sciencedirect.com/science/article/pii/B9780123748836000023</u>

Exercise 8: Some basic calculus

symbolic computations in MATLAB

The Symbolic Math Toolbox is an additional collection of software routines available with MATLAB that provides tools for solving and manipulating symbolic math expressions, hence performing differentiation, integration, simplification, transforms, ...

For an introduction to this toolbox, open the help window and look in **Symbolic Math Toolbox** \rightarrow **Getting Started**. For the exercise that follows you are required to have understood how to define symbolic variables and expressions, substitute numbers in symbolic expressions, differentiate and compute indefinite and definite integrals.

Hence the topics you should read in the Getting Started Help pages are:

Symbolic Objects:

• Overview, Symbolic Variables

Creating Symbolic Variables and Expressions:

Creating Symbolic Variables, Creating Symbolic Expressions

Performing Symbolic Computations:

- · Simplifying Symbolic Expressions, Substituting in Symbolic Expressions
- · Substituting in Symbolic Expressions: Substituting Symbolic Variables with Numbers
- Differentiating Symbolic Expressions: Derivatives of single-variable expressions)
- Integrating Symbolic Expressions Indefinite and definite integration (only of one-variable expressions).

This is a simple example on how to define and differentiate symbolic function:

A. You should now define and differentiate the following functions using the diff operator in MATLAB (you should also check that you know how to do them yourself!):

(1)
$$\frac{x+1}{x^2+1}$$
 (2) $(\sin x)^4$ (3) $x^2 \sin x$

(4)
$$\sqrt{\frac{1+x}{1-x}}$$
 (5) $x e^{x^2}$

Now simplify the expressions obtained using the command simplify.

B. We can similarly integrate functions using the int. Evaluate the indefinite integrals of the five functions above. The int command can also be used to perform definite integrals. Calculate the definite integrals of the five functions over the appropriate ranges:

(1) [0, 1] (2) $[0, \pi/2]$ (3) [0, 1] (4) [0, 1] (5) [1, 2]

and finally express the numerical values obtained in a decimal format by converting them using double, e.g. double $(pi/2) \rightarrow 1.5708$.

C. For case (1) compute the definite integral by numerical integration (using trapz as in Exercise 7) sampling the function on a vector of N equidistant points. Compare the result with that obtained with symbolic integration. How many points N are needed so that the **relative** difference between the numerically computed integral and the exact definite integral is less than 10^{-4} ?

Tip: to solve exercise C you are encouraged to use the while loop, a control-flow statement alternative to the for loop.

The general form of the while statement is

```
while (condition) statements end
```

The condition is a logical relation and the statements are executed repeatedly while the condition remains true. The condition is tested each time before the statements are repeated. It must eventually become false after a finite number of steps, or the program will never terminate. For further information see MATLAB's help page on the while loop and/or *Chapter 2 (in particular 2.8) in the e-book "Essential Matlab for Engineers and Scientists": http://www.sciencedirect.com/science/article/pii/B9780123748836000023*

When the exercise is complete save your work and show it to a demonstrator.

Exercise 9: Wave Motion animations

In this exercise we will show how to create an animation in MATLAB, and use this feature to visualise wave motion in one and two dimensions.

The displacement of a wave in one dimension is

```
y(x,t) = a \cos\left(2\pi f\left(t - \frac{x}{v}\right)\right)
```

where f is the frequency, v the velocity and a the amplitude.

Setting the values a=1 for the amplitude, f=1 for the frequency, and v=1 for the velocity, we can plot the displacement at successive times using a for loop. By using the getframe and movie commands we can capture movie frames and play the movie. See the online help on getframe and movie, as well as the following example:

```
% define parameters
a=1;
f=1;
v=1;
dt=0.1;
x=linspace(-2,2,100);
j=0;
clear movie1
for t=0:dt:2
    j=j+1;
    plot(x,wave(t,x,f,a,v)); % where wave is a function that
you should define to compute the displacement
    % add text with elapsed time on the plot
    text(0, 0,['t=',num2str(t),' s'])
    % fixes the limits of the y axis
    ylim([-a a])
    % updates the figure
    drawnow
    % save frame j
    movie1(j)=getframe;
end
%% plays the movie once
```

```
movie(movie1,1,1/dt)
```

You should now animate the following situations; clearly explaining in each case what you see:

A. A wave of the same amplitude, frequency and velocity as the above case, but moving in the opposite direction

B. The superposition of the previous two waves.

C. The superposition of two waves of the same amplitude and velocity, but different frequencies travelling in opposite directions: choose as values $a_{1=a_{2=1}, v_{1=v_{2=1}, f_{1=2}, and f_{2=2.2}}$

D. A surface plot (using surf) of a wave in 2 dimensions of amplitude a=1, frequency f=2, and velocity v=1 produced by a source in the origin. The equation for a travelling wave in two dimensions is simply obtained by replacing x by:

$$r = \sqrt{x^2 + y^2}$$

Remember the command meshgrid that you used in Ex. 3.

Finally, draw and animate contour plots using the command imagesc(x, y, z) where **x** and **y** are arrays of size *n* and *m*, and **z** is a matrix of size (*n*,*m*) (see doc imagesc for examples).

E. A contour plot of a wave in 2 dimensions with the same amplitude, frequency and velocity as in the previous case, but produced by sources at the points (1,0) and (-1,0).

F. A contour plot resulting from the interference produced by 5 identical sources equally spaced between the points (-1,0) and (1,0).

When the exercise is complete save your work and show the movies to a demonstrator.

Exercise 10: Simple Harmonic Motion and Pendulums solving differential equations

Many of the equations we meet in physics involve derivatives and hence are differential equations. An important example is Newton's second law which is a second order differential equation since it involves acceleration (the second time derivative of displacement. MATLAB is equipped with several routines to solve differential equations.

The damped Driven Pendulum:

The angular displacement $\theta(t)$ in such a pendulum satisfies the second order differential equation:

$$\frac{d^2}{dt^2}\theta(t) + 2k\frac{d}{dt}\theta(t) + \sin\left(\theta(t)\right) = f\,\cos\left(\omega\,t\right) \quad \text{Eq. 1}$$

where k is called the damping ratio, f is the amplitude and ω the angular frequency of the forcing term. For simplicity we have set g/l=1 in the equation above, where g is the gravitational acceleration and I the length of the pendulum.

For small amplitude motion we can replace $sin(\theta)$ by θ to obtain the equation for damped forced simple harmonic motion:

$$\frac{d^2}{dt^2}\theta(t) + 2k\frac{d}{dt}\theta(t) + \theta(t) = f\cos\left(\omega t\right).$$

In MATLAB we can solve such an equations by using the ode45 routine, which is invoked by the command ode45 (@function,t,u0), where function defines the right side of the differential equation you would like to solve (see example below), t a vector specifying the interval of integration, and u0 a vector of initial conditions.

Many more differential equation solvers are available in MATLAB, and you can find a description of these in the User's Guide \rightarrow Mathematics \rightarrow Calculus. Most of the information you need to know to solve this exercise is however presented in the example below.

As an example, we solve the case of undamped unforced simple harmonic motion.

$$\frac{d^2}{dt^2}\theta(t) + \theta(t) = 0$$

with the initial conditions $\theta(0)=0.1$, $\theta'(0)=0$.

MATLAB Ordinary Differential Equation (ODE) solvers accept only first-order differential equations. To use the solvers with higher-order ODEs, you must rewrite each equation as an equivalent system of first-order differential equations, in this case:

$$\frac{d}{dt}\theta(t) = \theta'(t)$$
$$\frac{d}{dt}\theta'(t) = -\theta(t)$$

Then we need to define a function handle to describe the system of two first-order differential equations in θ and θ '. We do this in a separate function file, which we call e.g. UnUnfSH.m:

```
% Function handle for ODE of an unforced, undamped, simple
% harmonic oscillator
function du = UnUnfSH(t,u)
    theta = u(1);
    theta_prime=u(2);
    du = zeros(2,1);
    % first equation in theta
    du(1) = theta_prime;
    % second equation
    du(2) = - theta;
end
```

Then we define a vector of initial conditions $u0=[0.1 \ 0]$, and a vector specifying the interval of integration t = 0:0.1:100;

Now all we need to do is run the ode45 command, which will give as output the solution of the differential equation in the specified t domain (i.e. $\theta(t)$ and $\theta'(t)$).

```
[T, SOL] = ode45(@UnUnfSH, t, u0);
```

We can then plot $\theta(t)$ by selecting the first column in the solution array using e.g. plot(T,SOL(:,1)), and obtain the following figure:



Try changing initial conditions and domain of integration to familiarise with the command.

You should now use the ode45 command to investigate damped driven simple harmonic motion in the following situations:

A. damping but no forcing term, show both under-, over- and critical damping (k < 1, k > 1, k=1)

B. forcing in each damping case away from resonance (choose f=0.1 and ω =1.2).

C. the damped forced **pendulum** (Eq. 1) with k=0.05, f=1.5, and ω =1.1 with initial conditions $\theta(0)=0.1$, $\theta'(0)=0$ and $\theta(0)=0.101$, $\theta'(0)=0$, both plotted on the same graph. Note the effect on the solution of a small change in the initial conditions. This extreme sensitivity to initial conditions is the signature of chaos.

When the exercise is complete save your work and show it to a demonstrator.

Exercise 11: Numerical Errors and Interpolation

A computer stores floating-point numbers using just a small amount of memory. Typically, single precision (float in C++) allocates 4 bytes (32 bits) for the representation of a number, while double precision (double in C++, MATLAB default precision) uses 8 bytes. A floating point number is represented by its mantissa and exponent (e.g. 1.5e-15, the decimal mantissa is 1.5 and the exponent is -15). The conventional format for double precision uses 53 bits to store the mantissa (including 1 bit for the sign) and 11 bits for the exponent.

The necessarily finite representation of floating-point numbers in terms of bits defines the maximum range and the number of significant digits.

The maximum range is the magnitude of floating-point numbers given the fixed numbers of bits used for the exponent. For single precision a typical value is $2^{\pm 127} = 10^{\pm 38}$; for double precision (default in MATLAB) it is typically $2^{\pm 1023}$ i.e ~ $10^{\pm 308}$. Exceeding the single precision is not difficult. Consider, for example, the evaluation of the Bohr radius in SI units,

$$a_0 = \frac{4\pi\epsilon_0\hbar^2}{m_{\rm e}e^2} \simeq 5.3 \times 10^{-11} \text{ m}$$

While its value lies within the range of a single-precision number, the range is exceeded in the calculation of the numerator and the denominator. The best solution for dealing with this type of range difficulty is to work in a set of units natural to a problem (in the example above, work in units of angstroms, the charge of an electron).

A. Range error

Sometimes range problems arise because the numbers in the problem are inherently large. Let's consider the example of the factorial function:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

In MATLAB you can easily compute n! using the prod command and the colon operator: nFactorial=prod(1:n)

Determine which is the largest n for which you can compute the factorial function in MATLAB. When the result is too large to represent as conventional floating-point value, MATLAB returns Inf, a special floating-point number reserved for representing infinity.

In practice, a common solution to working with very large numbers is to use their logarithm. Write a MATLAB script to compute $log_{10}(n!)$ for n=200.

To print the result, express it as $n!=(mantissa) \times 10^{(exponent)}$, where the exponent is the integer part of $log_{10}(n!)$, and the mantissa is 10^a , where a is the fractional part of $log_{10}(x)$, and use the following command to print it on screen:

sprintf('n!= %6.5fE%i',mantissa, exponent)

Tip: have a look at the function floor to round the number x to the nearest integer less than or equal to x.

B. Round-off error

Suppose we wanted to numerically compute f'(x), the derivative of a known function f(x). The derivative f'(x) can be computed as:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

with $h \rightarrow 0$. Clearly setting h=0 when evaluating f'(x) will not give us anything useful, but what if we set h to a very small value, say h=10⁻³⁰⁰ using double precision?

The answer will still be incorrect due to the second limitation on the representation of floating-point numbers: the finite number of digits in the mantissa. For double precision the number of significant digits is about 16 digits. Thus, in double precision the operation $3+10^{-20}$ returns an answer of 3 because of round off; using h=10⁻³⁰⁰ in Eq. 1 may simply return 0 when evaluating the numerator.

We define the absolute error when computing a derivative as:

$$\Delta(h) = \left| f'(x) - \frac{f(x+h) - f(x)}{h} \right|$$

where f'(x) is the exact analytical derivative of the function f(x).

Write a MATLAB script that computes and plots in a log-log scale $\Delta(h)$ vs h for f(x)=x² and x=1. Use h=1, 10⁻¹, 10⁻², 10⁻³, ..., 10⁻²⁰. Why does $\Delta(h)$ first decreases and then increases as h becomes smaller?

C. Interpolation

Interpolation is the process of defining a function that takes on specified values at specified points. In MATLAB there are several functions that implement various interpolation algorithms. All of them have the calling sequence $v = interp \ function(x, y, u)$

The first two input arguments, x and y, are vectors of the same length that define the interpolating points. The third input argument, u, is a vector of points where the function is to be evaluated. The output v is the same length as u and has elements $v(k) = interp_function(x, y, u(k))$

Make a plot of your hand. Start with:

```
figure('position',get(0,'screensize'))
axes('position',[0 0 1 1])
[x,y] = ginput;
```

Place your hand on the computer screen. Use the mouse to select a few dozen points outlining your hand. Terminate the ginput with a carriage return. You might find it easier to trace your hand on a piece of paper and then put the paper on the computer screen. You should be able to see the ginput cursor through the paper. Save the variables x and y into a MATLAB variable (as in Ex 4) so that you can load them in the workspace later.

Now think of x and y as two functions of an independent variable that goes from one to the number of points you collected. You can interpolate both functions on a finer grid and plot the result with

```
n = length(x);
s = (1:n)';
t = (1:.05:n)';
u = interp_function(s,x,t);
v = interp_function(s,y,t);
clf reset
plot(x,y,'.',u,v,'-');
```

where interp_function is just a placeholder for the interpolation function of your choice.

Try interpolating the points using a piecewise linear interpolation scheme (piecelin) and a piecewise cubic spline named splinetx. You can download these interpolation functions (written by C. Moler) <u>here</u>. If you'd like to learn more about splines, have a look e.g. at <u>http://mathworld.wolfram.com/Spline.html</u>.

When the exercise is complete save your work and show it to a demonstrator.

Exercise 12: Stellar Pulsations and Planetary Transits

generating sound with MATLAB

To be submitted though CANVAS by the end of Week 11.

Stars are spheres of self-gravitating gas and, under certain conditions, they pulsate with resonant, global, acoustic oscillation modes. These (small) oscillations can be detected as periodic perturbations of the star's luminosity. Stars as the Sun oscillate in several oscillation modes at the same time. The spectrum of acoustic oscillation in such stars is characterised by two main frequencies: v_{max} , which represents the frequency with maximum oscillation amplitude, and Δv , which is the typical spacing between consecutive frequencies.



 Δv and v_{max} are directly related to global stellar properties such as mass M, radius R, and temperature T_{eff} :

$$\nu_{\rm max} = 3090 (M/{\rm M}_{\rm sun}) (R/{\rm R}_{\rm sun})^{-2} (T_{\rm eff}/5777)^{-1/2} \ \mu Hz \quad \text{Eq. 1}$$

$$\Delta \nu = 134.9 \ (M/{\rm M}_{\rm sun})^{1/2} \ (R/{\rm R}_{\rm sun})^{-3/2} \ \mu Hz \quad \text{Eq. 2}$$

where R_{sun} and M_{sun} are the radius and mass of the Sun and the temperature is expressed in Kelvin.

In the case of the Sun the frequencies of oscillations are then of the order of 3000 $\mu\text{Hz},$ i.e. about 5 minutes.

A. Stellar sound

In MATLAB you can play sound using the sound command. As an example here we generate and play for 5 seconds a sinusoidal wave of frequency v=1 KHz, with amplitude A=1 and random phase constant ϕ .

```
fs = 44100; % standard sampling rate
T = 1/fs; % sampling period
t = 0:T:5; % time vector
phi=2*pi*rand(1,1); % random phase
nu=1e3; % frequency in Hertz
X=sin(2*pi*nu*t+phi);
sound(X, fs) % play the signal
```

Let's now consider acoustic oscillations in the Sun.

Using the solar values of v_{max} and Δv (from Eq. 1 and 2, assuming the temperature of the Sun to be T_{eff}=5777 K):

1. Consider the superposition of five pulsation modes with frequencies $v_i=v_{max} \pm n \Delta v$, with n=0,1,2. Assume that the perturbation associated to each pulsation is a simple sinusoidal function $f_i(t)=A^*sin(2\pi^*v_i^*t+\varphi_i)$. Assume that all pulsation modes have the same amplitude (A=1), but that the phase at t=0 of each mode is a random number between 0 and 2π (use the MATLAB command rand(1,1) to generate a random number between 0 and 1). Consider the same time vector as in the example above, but convert the frequencies of stellar oscillations into a range of frequencies audible by humans. A convenient conversion factor is:

Play with sound the signal you have generated, i.e. the superposition of five pulsation modes.

- 2. Now let's consider the oscillation frequencies of a star as it evolves in time. By loading the file <u>star_evolving.mat</u> you will find in the workspace two arrays (radius and age) describing how the radius of a 1 M_{sun} star (expressed in R_{sun}) changes with time (expressed in years). Consider the stellar temperature always fixed to the Solar value. For each element of the arrays plot a figure with four subplots showing:
 - a. The perturbation $f_i(t)$ due to each of the five oscillation modes in one plot,
 - b. The total perturbation (sum of the perturbations associated to the five modes)
 - c. The stellar radius as a function of the age of the star,
 - d. A sphere representing a star of a given radius. As in the example here below, use the command sphere and fix the axes limits as the star evolves.

```
[x,y,z] = sphere;
surf(R*x,R*y,R*z) % sphere centred at the origin
axis square
```

2.11

```
rscale=5;
xlim([-rscale rscale])
ylim([-rscale rscale])
zlim([-rscale rscale])
shading interp
```

For each star, generate the signal due to oscillations (as in step 1), and play the sound while displaying the corresponding figure. You should be able to generate a plot similar to the one here below.



QA2: How does the sound of stars change as they get older? and why?

B. The Transit Method of Detecting Extrasolar Planets

When a planet passes in front of a star as viewed from Earth, the event is called a "transit". On Earth, we can observe an occasional Venus or Mercury transit. These events are seen as a small black dot creeping across the Sun-Venus or Mercury blocks sunlight as the planet moves between the Sun and us. Space-borne telescopes, such as CoRoT and

Kepler, can detect planets by looking for tiny dips in the luminosity of a star when a planet crosses in front of it—we say the planet transits the star.

	Т	ransit	Light	Curve	S
	Kepler 4b	Kepler 5b	Kepler 6b	Kepler 7b	Kepler 8b
1.000 0.995				$\overline{\mathbf{b}}$	
0.990	H 8 4 Phase (hours)	-t t t Phase (hours)	* * *	4 0 4 Phase (hours)	+ 0 + Phase (hours)
Orbital Period (days)	3.2 days	3.5 days	3.2 days	4.9 days	3.5 days

A planetary transit in front of a stellar disc causes a decrease of the observed flux which can be approximated as $d=(R_P/R_S)^2$, where R_P and R_S are the radius of the planet and of the star, respectively. For a star like the Sun, the typical relative variations are 10⁻⁴ for Earth-size planets, and 10⁻² for Jupiter-size bodies.

The information we have on the planet depends on our knowledge of the stellar radius, which can be determined with high accuracy if the star shows acoustic oscillations.

Suppose that a planetary transit with a relative depth of d=1.5 10^{-5} is detected when monitoring the luminosity of a star. The star shows pulsations as well, and its oscillation spectrum is characterised by v_{max}=1954 µHz and Δv =95.66 µHz. Moreover, we know that the temperature of the star is the same as that of the Sun.

- 1. Determine the mass and radius of the star by combining Eq. 1 and 2, then determine the radius of the transiting planet in units of Earth radii: which planet in the solar system has a similar radius (QB1)?
- 2. Finally, use a piecewise linear interpolation (see Ex. 11) to estimate the age of the planet (assumed to be the same as the age of the star) using the data provided in part A (<u>star_evolving.mat</u>). Use the stellar radius as the independent variable when interpolating the age.

Useful astrophysical quantities: R_{sun}=6.9599 10⁵ km R_{earth}=6.371 10³ km To get this exercise marked, please submit the script file (.m file) through CANVAS. The deadline for submission is the end of **your** session in Week 11.

Besides having to write a code that produces the required output, you'll need to:

1. Include comments throughout the code: the marker must be able to follow the code easily and to understand its logic and function,

2. Correctly label plot axes,

3. Include answers to questions (QA1, QB2) as part of the comments in the script file.

The marking scheme for this exercise is the following:

A1: 7.5%

A2: 5%

B1: 7.5%

B2: 5%

B. Optional - Non-assessed work

MATLAB can be used to create Graphical User Interfaces (GUIs) which are useful to create more interactive programs. A GUI acts as an interface between the user and the programme and is based on both input and output.

First, read through Chapter 13 in the e-book "Essential Matlab for Engineers and Scientists"

http://www.sciencedirect.com/science/article/pii/ B9780123748836000138

and follow the steps in sections13.2 and 13.3 to familiarise with GUIDE and how to incorporate and use *static* and *edit text controls*, as well as *push buttons*.

Input (using *edit text controls*: v_{max} , Δv , T_{eff} , and transit depth).

Output (using *static text controls* and *push buttons*)

- stellar mass, radius and planetary radius (calculated when clicking on a pushbutton)

- play (clicking on a pushbutton) the sound generated by acoustic modes in a star with the specified v_{max} and Δv (as in 12A).

The final GUI should look like this figure, but you are welcome to include additional features!

Stellar sound and exoplanets Input Delta nu [muHz] 134.9 nu max [muHz] 3090 T_eff [K] 5777 conv 1e5 factor to scale stellar oscillation frequence to the audible range	
Input Input Delta nu (muHz) 134.9 nu max [muHz] 3090 T_eff [K] 5777 conv 1e5 factor to scale stellar oscillation frequent to the audible range	
Input Delta nu [muHz] 134.9 nu max [muHz] 3090 T_eff [K] 5777 conv 1e5 factor to scale stellar oscillation frequence to the audible range	
Delta nu [muHz] 134.9 nu max [muHz] 3090 Transit depth 100 T_eff [K] 5777 conv 1e5 factor to scale stellar oscillation frequent to the audible range	
nu max [muHz] 3090 Transit depth [ppm] 100 Teff [K] 5777 conv 1e5 factor to scale stellar oscillation frequent to the audible range	
T_eff [K] 5777 conv 1e5 factor to scale stellar oscillation frequent to the audible range	
conv 1e5 factor to scale stellar oscillation frequent to the audible range	
conv 1e5 factor to scale stellar oscillation frequent to the audible range	
	cies
Output	
Play sound!	
Compute stellar and planetary properties	
Stellar Mass [solar masses]	
Stellar Radius 1 Planetary 1.09243	
[soiar radii] [earth radii]	