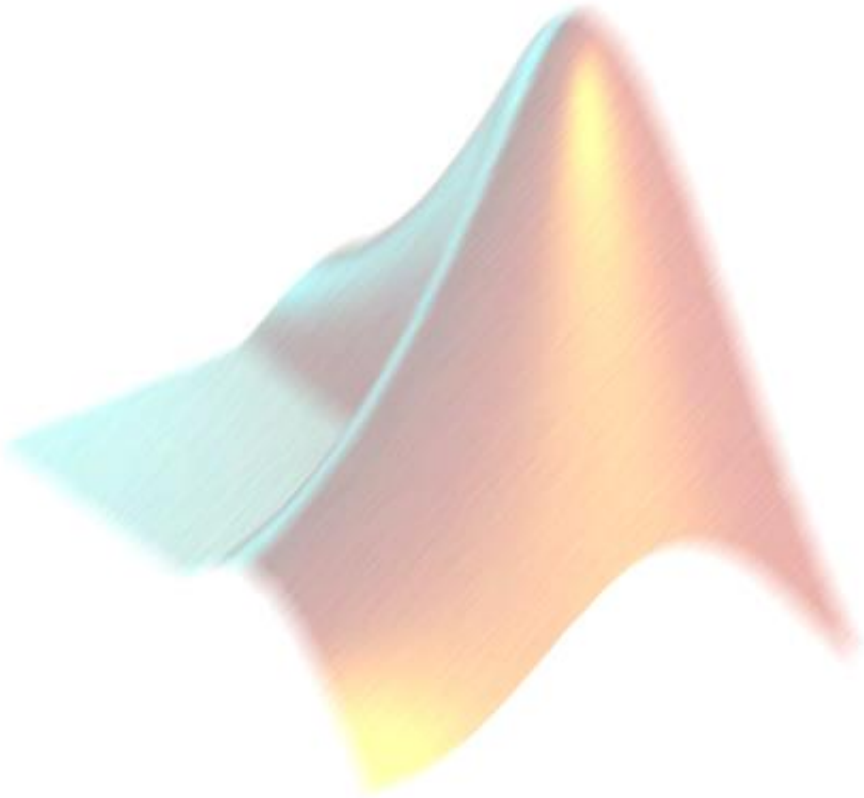
ENGGEN 131
Engineering Computation and Software Development

Laboratory 6: Linear equations and differential equations



The Department of Engineering Science
The University of Auckland

Laboratory 6

TASK 6.0

Download the file "Lab_6.zip" from Canvas and extract the contents to an appropriate file directory. This contains files you may need for later tasks.

Then read the post "Lab 6 Notes" on the class Piazza discussion forum.

LINEAR EQUATIONS

Systems of linear equations appear in many different branches of engineering. It is easy to solve a system of linear equations using MATLAB and only requires writing a few lines of code once the system has been written in matrix form.

Consider the problem of finding the intersection of the following two lines:

$$y = \frac{1}{2}x + 1$$
$$y = -x + 4$$

Replacing x with x_1 and y with x_2 we can rearrange these two lines to get the following system

$$-x_1 + 2x_2 = 2$$
$$x_1 + x_2 = 4$$

This system of linear equations can be written in matrix form as:

$$\begin{bmatrix} -1 & 2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

which is of the general form: $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} = \begin{bmatrix} -1 & 2 \\ 1 & 1 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$

In MM1 you will have learnt how to solve matrix equations of the form $\mathbf{Ax} = \mathbf{b}$.

Recall that if **A** has an inverse you can multiply both sides of the equation by the inverse of **A** to get:

$$\mathbf{A^{-1}Ax} = \mathbf{A^{-1}b}$$
$$\mathbf{Ix} = \mathbf{A^{-1}b} \qquad \text{That is, } \textit{the solution is the inverse of A multiplied by b}.$$
$$\mathbf{x} = \mathbf{A^{-1}b}$$

In MATLAB it is easy to find the inverse of square matrices and to perform matrix multiplication. To find the solution we can simply type the following:

```
A = [-1 2;1 1]
b = [2;4]
x = inv(A) * b
```

Check the accuracy of your solution by typing `A*x` and verifying that we get `b` (or close to it)

Once the matrix **A** and the column vector **b** have been assigned values it only took one line of code to solve the system. MATLAB also supplies the left division method, which is equivalent to Gaussian elimination and **generally gives a more accurate solution**. Try typing the following:

```
x = A\b
```

Verify that you get similar solution values.

Check the accuracy of your solution by typing `A*x` and verifying that we get `b` (or close to it)

Both of those methods ONLY work if **A** has an inverse. This can be checked before hand. Recall that a matrix has an inverse if, and only if, the determinant of the matrix is nonzero. To find the determinant of a matrix we can use the `det` command. Find the determinant of our matrix **A** by typing:

```
det(A)
```

Verify that it is nonzero. Recall that if your determinant is zero then the system of equations has no solution. Such a matrix is called a singular matrix.

If your matrix is singular, this also means that the `inv` command would not generate a valid inverse, as the inverse does not exist.

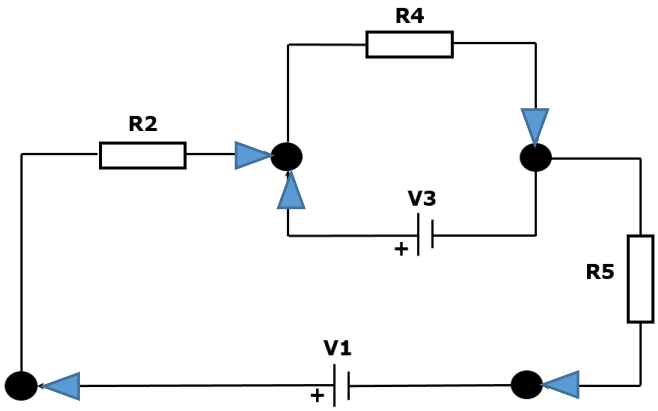
Try the following and see what happens:

```
A = [2 2;1 1]
b = [4;2]
x = inv(A) * b
```

Is this what you expected? Check the value of `det(A)`.
Does `A` have an inverse?

TASK 6.1 Solving a system of linear equations

Consider the following electrical circuit, where we are interested in determining the current flowing along each branch:



Applying Kirchoff's current law at each of the nodes (black dots), and using the current directions indicated by the arrows, we get the following system of equations:

$$\begin{aligned} i_1 - i_2 &= 0 \\ i_2 + i_3 - i_4 &= 0 \\ i_4 - i_3 - i_5 &= 0 \\ i_5 - i_1 &= 0 \end{aligned}$$

Applying Kirchoff's voltage law around the three loops in the circuit, we get the equations:

$$\begin{aligned} V_1 - i_2 R_2 - i_4 R_4 - i_5 R_5 &= 0 \\ V_1 - i_2 R_2 - V_3 - i_5 R_5 &= 0 \\ V_3 - i_4 R_4 &= 0 \end{aligned}$$

This gives 7 equations for 5 unknowns (the currents). This is two more equations than we need. From this point on, we **drop the final equation from each group.**

With a little work we can rewrite the five remaining equations as follows:

$$\begin{aligned} \mathbf{1}\cdot i_1 - \mathbf{1}\cdot i_2 + \mathbf{0}\cdot i_3 + \mathbf{0}\cdot i_4 + \mathbf{0}\cdot i_5 &= \mathbf{0} \\ \mathbf{0}\cdot i_1 + \mathbf{1}\cdot i_2 + \mathbf{1}\cdot i_3 - \mathbf{1}\cdot i_4 + \mathbf{0}\cdot i_5 &= \mathbf{0} \\ \mathbf{0}\cdot i_1 + \mathbf{0}\cdot i_2 - \mathbf{1}\cdot i_3 + \mathbf{1}\cdot i_4 - \mathbf{1}\cdot i_5 &= \mathbf{0} \\ \mathbf{0}\cdot i_1 + \boldsymbol{R_2}\cdot i_2 + \mathbf{0}\cdot i_3 + \boldsymbol{R_4}\cdot i_4 + \boldsymbol{R_5}\cdot i_5 &= \boldsymbol{V_1} \\ \mathbf{0}\cdot i_1 + \boldsymbol{R_2}\cdot i_2 + \mathbf{0}\cdot i_3 + \mathbf{0}\cdot i_4 + \boldsymbol{R_5}\cdot i_5 &= \boldsymbol{V_1 - V_3} \end{aligned}$$

Write a function called `ElectricalCircuitCurrent` that determines the current flowing along each branch of the electrical circuit displayed above.

Input:

an array of circuit parameters `[V1 R2 V3 R4 R5]` in this order.

Output:

an array of currents in each branch `[I1 I2 I3 I4 I5]` in this order.

The function `ElectricalCircuitCurrent` should encode the equations in matrix form, then solve the equations to find the current through each branch.

Test your function by completing the first section of the script `TestMaxV1.m` (also see Task 6.2) for the following parameters:

$$V_1 = 10V, R_2 = 3\Omega, \mathrm{V}_3 = 4V, R_4 = 4\Omega, R_5 = 3\Omega$$

and

$$V_1 = 2.6V, R_2 = 2.25\Omega, \mathrm{V}_3 = 7.7V, R_4 = 2.25\Omega, R_5 = 7.3\Omega$$

Submit the completed function `ElectricalCircuitCurrent` to MATLAB Grader.

TASK 6.2 Solving a system of linear equations

We now want to find by how much we can increase the voltage at V_1 in the **positivive direction** before melting a wire (which occurs when more than 5A of current passes through any of them).

Write a function called `MaxV1` that finds maximum voltage V_1 before the maximum current flowing through any of the wires exceeds 5A.

Input:

an array of circuit parameters `[V1 R2 V3 R4 R5]` in this order.

Output:

the maximum voltage V_1.

It should use an appropriate type of loop with an step size of 0.1V that increases the voltage V_1 in the **positivive direction** until the maximum current flowing through any of the wires exceeds 5A. Remember you need to find the maximum voltage **before the wire melts.**

Recall that both voltage and current can also be **negative**, indicating that the directions initially assumed are incorrect and the actual direction is the opposite to the indicated direction.

Your function should use the function `ElectricalCircuitCurrent` from Task 6.1. You can assume that the first solve will not result in any current already exceeding 5A.

The supplied script `TestMaxV1.m` (with some small modifications) will print to the screen, in the form:

```
The maximum voltage that can be applied is ? volts.
```

Where `?` is replaced by the number of volts.

Submit the completed function `MaxV1` to MATLAB Grader.

SOLVING ODES

Ordinary differential equations arise in many branches of engineering. It is relatively easy to find numerical solutions for ODEs using MATLAB if they can be written in the following form:

$$\frac{dy}{dt} = f(y,t)$$

i.e. the derivative can be written as some function of the dependent variable and independent variable. In many cases the derivative will only depend on one of the variables but MATLAB can handle functions which depend on both.

Consider the following differential equation.

$\frac{dy}{dt} = \cos \omega t$ with initial condition $y(0) = 0$

This can easily be solved by direct integration to give $y = \frac{1}{\omega} \sin \omega t$

Investigate the graph of this solution for time 0 to 1 seconds and an angular frequency of 2π by using the following script (available from Canvas):

```
% calculate the analytic solution of the ODE
% dy/dt = cos(omega * t);

% omega is the angular frequency
omega = 2*pi;

% our time range is from 0 to 1
t = linspace(0,1,100);

yAnalytic = 1 / omega * sin(omega * t);

plot(t,yAnalytic)
```

Now we will solve the same equation in MATLAB and compare it against the numerical solution. To calculate a numerical solution the solver needs three things:

- A formula for the derivative (in the form of a function)
- A time interval (start time and finish time stored in an array)
- An initial value at the start time (initial condition)

First we need to write a MATLAB function that calculates the derivative for any given values of *t* and *y*. This function will then be called many times by our solver to find our numerical solution. We can choose any valid function name for our derivative but as always it is a good idea to give the function a meaningful name.

Note that the solvers require the derivative function to take **both** the independent and dependent variables as inputs (even if one or the other may not be used). The independent variable must be the first

input and the dependent the second. The output for the function must be the derivative for the given inputs.

We can now write this function as follows:

```
function [dydt] = SinusoidDerivative(t,y)
% calculate the derivative dy/dt for the equation
% dy/dt = cos(omega * t)
% inputs: t, the independent variable representing time
%         y, the dependent variable representing displacement
% output: dydt, the derivative of y with respect to t

% omega is the angular frequency
omega = 2 * pi;
dydt = cos(omega * t);

end
```

Download this function from Canvas and save it as SinusoidDerivative.m.

Try testing the function with a few values of *y* and *t* and verify that it gives you the correct derivative value for the following inputs:

$t = 0, \quad y = 0$

$t = 0.25, \quad y = 1$

$t = 0.5, \quad y = 0$

$t = 1, \quad y = 1$

Now we are ready to write a script file to solve our ODE:

```
% calculate the numerical solution of the ODE
% dydt = cos(omega * t);

% set up an array containing the start time and finish time
% we will calculate solution values for this time range
% we only need to specify TWO values (the start and finish)
timeInterval = [0 1];

% our initial condition is y(0)=0
yinit = 0;

% solve our ODE, the solver expects three arguments in the
% the following order
% - the name of the derivative function (in quotes)
% - the time interval to solve for (a two element row vector)
% - the value at the start time
[t,y] = ode45('SinusoidDerivative', timeInterval, yinit);
plot(t,y)
```

Download this file from Canvas and run it. Compare your plot with the analytical solution.

A ROCKET-PROPELLED SLED

The following problem is a dapted from an example on page 535 of "Introduction to MATLAB 7 for Engineers", William J. Palm III. It forms the basis for tasks 6.3, 6.4 and 6.5.

Newton's law gives the equation of motion for a rocket-propelled sled as:

$$\text{(sled ODE)} \qquad m\frac{dv}{dt} = -cv$$

where m is the sled mass (kg) and c is the air resistance coefficient (N s/m). We know the initial velocity $v(0)$ of the sled and want to find velocity $v(t)$.

We want to plot the motion for the rocket-propelled sled described above using a few different methods including an analytic solution (Task 6.3) and a numberical solution (Task 6.4 + Task 6.5)

Our sled has a mass of 1000 kg, an initial velocity of 5 m/s and air resistance coefficient of 500 N s/m, we want to plot the rocket-propelled sled's velocity over the time interval $0 \le t \le 10$ with the different solution methods.

Finding v(t) Analytically

Using our MM1 knowledge we know we can solve this ODE using *Separation of Variables*.

Solve $m\frac{dv}{dt} = -cv$ given $v(0)$ is known.

Step 0 : Separate the variables to different sides

$$m\frac{dv}{v} = -c\ dt$$

Step 1: Integrate

$$\int m\frac{dv}{v} = \int -c\ dt$$
$$m\ln v = -ct + k$$

Note that k is a constant.

Step 2: Make v the subject

$$m\ln v = -ct + k$$
$$\ln v = \frac{-ct + k}{m}$$
$$v = e^{\frac{-ct+k}{m}}$$

Step 3: Rework the constant

$$v = Ae^{\frac{-c}{m}t}$$
$$\text{where } A = e^{\frac{k}{m}}$$

Step 4: Evaluate the constant

$$v(0) = Ae^{\frac{-c}{m}0}$$
$$A = v(0)$$

Step 5: Answer the question

$$v(t) = v(0)e^{\frac{-c}{m}t}$$

TASK 6.3 Analytic Solution

Create a function called `SledAnalyticSolution` that calculates the analytic solution for the rocket-propelled sled velocity.

Input:

an array of time t,
the initial velocity $v(0)$.

Output:

an array of the calculated velocity v.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledAnalyticSolution`.

Submit the completed function `SledAnalyticSolution` to MATLAB Grader.

TASK 6.4 Representing the ODE as a Matlab function

Write a function called `SledVelocityDerivative` that returns the derivative $\frac{dv}{dt}$, similar to that of the function `SinusoidDerivative` defined in the example above.

Input:

the time t,
the velocity v.

Output:

the derivative $\frac{dv}{dt}$.

Submit the completed `SledVelocityDerivative` to MATLAB Grader.

TASK 6.5 Numerical Solution using ode45

Write a function called `SledNumericalSolutionOde45` that calculates the numerical solution to the sled ODE using MATLAB's function `ode45`.

Input:

an array of the time span i.e. just the start and end time.
an initial velocity $v(0)$.

Output:

an array of time values t.
an array of the calculated velocity v.

It should call the function `ode45` and the function `SledVelocityDerivative` to numerically solve the sled ODE.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledNumericalSolutionOde45`.

Hint: Examine the section SOLVING ODES on the previous pages.

Submit the completed function `SledNumericalSolutionOde45` to MATLAB Grader.

LAB 6 Lab Task Submission Summary

□ Task 6.1
Submit the completed function `ElectricalCircuitCurrent` to MATLAB Grader.

□ Task 6.2
Submit the completed function `MaxV1` to MATLAB Grader.

□ Task 6.3
Submit the completed function `SledAnalyticSolution` to MATLAB Grader.

□ Task 6.4
Submit the completed function `SledVelocityDerivative` to MATLAB Grader.

□ Task 6.5
Submit the completed function `SledNumericalSolutionOde45` to MATLAB Grader

COMPULSORY LAB TASKS END HERE

EXAM PRACTICE – USING EULER'S METHOD (DEBUGGING)

Another way we can solve ODEs of the form:

$$\frac{dy}{dt} = f(y,t)$$

is by using Euler's method. Recall that in Euler's method subsequent *y* values are given by:

$y_{n+1} = y_n + h\ f(y_n, t_n)$ where *h* is the step size.

If you need more information about Euler's method see your MM1 notes or Wikipedia.

You are provided with a function `SledNumericalSolutionEuler` that should solve the ODE problem from Task 6.3 and Task 6.4 but by using Euler's method and `feval`. However, the function `SledNumericalSolutionEuler` has bugs.

Note the function `SledNumericalSolutionEuler` references the function `SledVelocityDerivative` from Task 6.4.

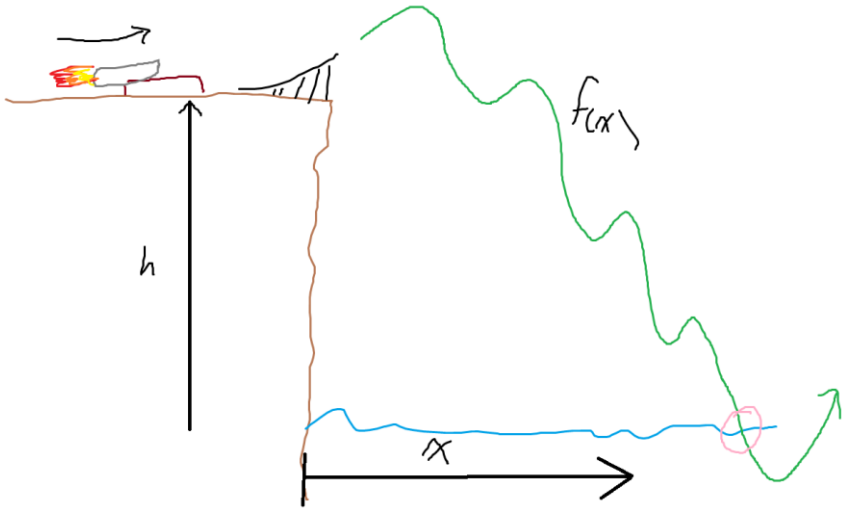
Open the function `SledNumericalSolutionEuler` in the MATLAB Editor, read the comments to understand what the function should do and then fix any bugs so that it runs correctly.

Remember you can use the debugger to step through line by line, so that you can see how each line of code changes the variables in the workspace.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledNumericalSolutionEuler`.

EXAM PRACTICE - PREDICTING A CRASH SITE (DEBUGGING)

The rocket-propelled sled is now sent off a ramp located on a cliff with elevation `h` above sea level. A few seconds after the rocket-propelled sled leaves the ramp, the rocket explodes. The explosion causes the onboard computers to send its distressed flight path elevation (relelative to the cliff top elevation) as a mathematical function `f(x)`, which is a function of distance `x` from the ramp. To determine where the crash site location is, we must determine when the flight path elevation is first `-h`.



That is, we want to find the first value of x such that:

```
-h = f(x)
0 = f(x) + h
```

You are provided with a function `SledCrashSiteLocation` that should be able to determine the crash site location `x` using the function `fzero`. However, the function `SledCrashSiteLocation` has bugs.

Open the function `SledCrashSiteLocation` in the MATLAB Editor, read the comments to understand what the function should do and then fix any bugs so that it runs correctly.

Remember you can use the debugger to step through line by line, so that you can see how each line of code changes the variables in the workspace.

Use the script `TestSledCrashSiteLocation.m` to test and view the output of the function `SledCrashSiteLocation`.

Debug the function `SledCrashSiteLocation` so that it correctly calculates the crash site.

EXAM PRACTICE - USING VECTOR ALGEBRA TO ESTIMATE PI

The following algorithm calculates increasingly accurate approximations to pi using vector algebra:

Set a to [0 1 0] and b to [1 0 0]
For n = 1 to the number of iterations

Set b to $\frac{1}{2}(a+b)$

Normalise b: $b = \frac{b}{|b|}$

Calculate an approximation of pi using: $pi = 2^{n+1}|a \times b|$

end

Write a MATLAB script that asks the user to specify the number of iterations to use and then calculates a sequence of approximations to pi using the algorithm described above. For every iteration your script should display the iteration number and the value of the approximation to 8 decimal places as shown in the sample output below:

```
Value of pi for iteration 1 is 2.82842712
Value of pi for iteration 2 is 3.06146746
Value of pi for iteration 3 is 3.12144515
Value of pi for iteration 4 is 3.13654849
Value of pi for iteration 5 is 3.14033116
Value of pi for iteration 6 is 3.14127725
Value of pi for iteration 7 is 3.14151380
```

IMPORTANT: Remember to comment your file.