3 DBT Part A: Simulating a Rocket Launch

3.1 Introduction

In the remaining sessions 3-5, you will reinforce and apply what you have learnt in the sessions 1-2 in a more open-ended **Design, Build and Test** project. Your tasks are to build simulators for the launch of a Saturn V rocket shown in Figure 3.1, and for the landing of a lunar module.

With careful incremental software design and build you will be able to tackle a complicated problem in an incremental, progressive fashion. In the early stages, example code will be provided which you should use as a template.

You will be required to undertake sets of tasks as you proceed. These tasks will, of course, help you in the project, by making you think about software design and good coding habits in general, and about the modular use of functions and structures in particular.

You will be assessed on

- how well your code works
- how clearly the code is written,
- how clearly you can talk about your design
- how well designed your code is
- the neatness of your filestore, and

3.1.1 Good Coding

You should keep the following points in mind: they will serve you well in both the DBT exercise and in the rest of your computing career.

Practice. You will only become reasonably skilled at programming by practising, which requires time spent at the keyboard trying things out. Unlike other areas of practical engineering, you cannot break anything by experimentation! Matlab is an ideal language for trying things out because (i) it is interpreted at the prompt and (ii) it has many powerful methods built-in. A little practice therefore brings substantial reward.

Design. As with all areas of engineering, design plays a crucial rôle. There comes a point when, rather than typing and hoping, you must think carefully, plan on paper, draw a diagram, and so on. (The bouncing ball was a good example where things get too complicated to make up on the spot!) If you find yourself trying to fix syntax bugs at the same time as how to solve the larger problem, you need to put in some design effort.

A good way to design code is to start with very high level pseudo-code — statements written in plain language. Then burrow down into the next level of detail, writing pseudo code as you go, and so on until you reach the core tasks — ones that cannot be broken down further. At this point you will (i) know what those core tasks are, and (ii) have a clear picture about how all the parts fit together. Then, starting from those lowest level core tasks and working back upwards, translate the pseudo code to valid code.

Remember **top-down design, bottom-up build**.

Interestingly, if you design in this way, you will find your code very easy to modify if the overall task

changes somewhat. Also because the core tasks do fundamental indivisible things they are often useful elsewhere. (As an example, in Lecture 2 we saw how we could write a Matlab function for the transfer function of a bridge circuit which called (twice) a more basic function describing a potential divider.)

Build. The logical organization of your filestore (directory structure, directory and filenames) and the neatness and clarity of your code are much more than optional "window dressing". Code that is difficult to read is likely to be the product of a confused programmer and is more likely to harbour errors.

Add comments to your code. Indent lines of code to make for loops, etc, stand out. Adopting a consistent style will clarify your thinking, and will make demonstrators more willing to invest time in your code.

- Use variable names which convey some meaning to the reader: $a1$, $a2$, $a3$ are rarely the best choices of name! (But don't use overlong names either.)
- Code indentation makes code easy to read. Matlab will even format it for you — press ctrl-a then ctrl-i.
- Insert white space to break up blocks of code. Use comments to indicate the purpose of a block of code, and add a detailed to a particular line if it is noteworthy.
- If you find yourself repeating code it should probably be put into in a function.
- Don't bury "magic numbers" in your code.

  If your code needs constants declare them at the top of the file and assign them to variables names. This way you won't have constant numbers appearing all over your code-base which if you need to change will undoubtably lead to misery (you'll forget to change all the places it mentioned).

```
finalspeedofvehicle=initialspeedofvehicle + ...
accelerationduetogravity*timeelapsedsincethestart;
b = a + 1;              % b becomes a plus one
E = m*(299792458)^2;
Gpoint6 = ((w^2-w0^2)^2+4*w^2*w0^2*0.6^2)^(-0.5);
Gpoint7 = ((w^2-w0^2)^2+4*w^2*w0^2*0.7^2)^(-0.5);
```

Testing. You should test code as you go along. When you have written a function, check that it behaves as expected.

When all your code appears to "work", stress-test it by changing input parameters, checking that the output changes as expected. Regard the code not as an end in itself, but rather as a piece of apparatus that allows you now to perform experiments.

Archiving at milestones. Your filestore is backed-up regularly by the Department, so your code should be secure against fire, flood and so on.

However, when developing a project you should saved snapshots of your code as you progress. When you reach a milestone, copy your working files to a backup and comment to the file describing where you had got to. This will save you pain later.

Suppose you are developing in a directory call DBT. A simple but effective way of taking a snapshot would be to

```
cp -r DBT  DBT.FridayWeek5
```

Then carry on working in DBT.

But if the major rewrite you make on Monday Week 6 turns out to be a big mistake ...

```
mv DBT  DBT.DisasterMondayWeek6
cp -r DBT.FridayWeek5 DBT
```

Notice that Monday's disaster is not immediately deleted. Never delete in haste.

Initial Mass	2900 tonnes
Total Thrust at Launch	34000 kN
Stage 1 Burn Time	150 s
Stage 1 Propellant Mass	2,150,000 kg
Max Diameter	10 m
Length	110m

Figure 3.1: The Saturn V rocket used in the Apollo 11 Mission, and some flight parameters. You'll build a flight simulation in software.

3.2 Simulating a Rocket Launch

3.2.1 Solving a Differential Equation with Software

The flight of a rocket can be modelled mathematically using a 2nd order differential equation involving position, and its time derivatives, velocity and acceleration.

One approach to solving higher order differential equations numerically is to split the equation into a number of 1st order differential equations which can be solved by approximating gradients. (If you want to read more, look up Euler's forward method in any good applied mathematics textbook — or see the lecture slides.) The presentation here is intuitive rather than formal.

In one dimension, assume at time k we know values the instantaneous acceleration a_k, velocity v_k, and position x_k of the rocket.

Using "forward differencing" involving the velocity v_{k+1} of the rocket at timestep $k+1$ a small time

Δt later, the acceleration can be written as

$$a_k = \frac{dv_k}{dt} \approx \frac{v_{k+1} - v_k}{\Delta t} \ .$$

so that

$$v_{k+1} = v_k + \Delta t \, a_k \ . \tag{3.1}$$

Similarly, using velocity and position,

$$x_{k+1} = x_k + \Delta t \, v_k \ . \tag{3.2}$$

Hence, provided you know the position and velocity of the rocket at any timestep k, *and you have an independent way of working out the acceleration at that timestep*, these equations form a scheme to compute position and velocity at the next timestep $k + 1$, and so on over all time.

Rather than an analytical solution $x(t)$ to the original differential equation, the numerical solution is a list (or Matlab vector) of x values corresponding to another list of time values. Table 3.1 gives an example for $t = 0.1$. (x is initially -2 for this rocket, which is a bit ominous!)

k	1	2	3	4	5	6	7	8	···
t	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	···
x	-2	1.8	2.3	2.1	4.7	6.0	6.1	7.4	···

Table 3.1: A discrete time solution: k is a integer time index and x_k is the numerical value of x at time t_k

Merely as a reminder of formatted output, If the x direction was the vertical, we could imagine producing formatted output using

```
fprintf('Time=%f Altitude=%f\n', t(k), x(k) );
```

3.2.2 Applying the method

Before we can proceed we need some information about the rocket. The table in Figure 3.1 gives some specifications of a typical Saturn V rocket (read these carefully — it was a monster).

In Listing 3.2 is skeleton code for a simple simulator that iterates, each time step calculating an acceleration, then a velocity and then a new position. Have a look at it and make sure it makes sense to you. If you are unclear about any parts, talk to a demonstrator.

Listing 3.1: An iteration loop - we need to write function GetAcceleration()

```
 v      = 0;                % initial velocity
 h      = 0;                % initial height
 tstart = 0;                % start time
 dt     = 0.1;              % time step
 tstop  = 300;              % how long to simulate for
 for t = tstart:dt:tstop
  a    = GetAcceleration(); % get current acceleration
% find the new values
```

```
  h     = h  + dt * v;        % update height
  v     = v  + dt * a;        % update velocity
 end;
```

For plotting, it is useful to capture these values into vectors. We could convert all the `a`, `v`, `h`, and `t` values into vectors (for example, the first line becomes `v(1) = 0;` and the update becomes `v(k+1) = v(k) + ...` However on an actual rocket flight the telemetry would not be stored in the program itself (why not?). So here is a simple fix.

Note that we don't store the final set of updated velocity and height – but these actually belong to time `t=tstop+dt`, so no matter. Note too that the height is updated before the velocity. Can you see why?

Listing 3.2: Capturing the values in vectors

```
  v       = 0;                  % initial velocity
  h       = 0;                  % initial height
  tstart  = 0;                  % start time
  dt      = 0.1;                % time step
  tstop   = 300;                % how long to simulate
  k       = 0;
  for t = tstart:dt:tstop     % vector of times
    a     = GetAcceleration(); % get current acceleration
% capture telemetry for plotting
    k=k+1; A(k)=a; V(k)=v; H(k)=h; T(k)=t;
% find the new values
    h     = h  + dt * v;          % update height
    v     = v  + dt * a;          % update velocity
 end
```

3.2.3 At first GetAcceleration() is GetConstAcceleration()

Recall the earlier remark *"and you have an independent way of working out the acceleration at each timestep"*.

There is a gap in your solution, which has been patched over using a function `GetAcceleration()`. This is not laziness, but actually an example of good code design mentioned in the introduction. Design broadly at first, and abstract the difficult detail into functions to be written later.

It will be no surprise that Newton's laws of motion are your first port of call when specifying the function.

For now you should assume that the thrust and mass of the rocket are constants. If the rocket produces a thrust F and has mass M:

$$Ma = F - Mg \tag{3.3}$$

where g is the acceleration due to gravity at the earth's surface. So

$$a = (F/M - g)\ . \tag{3.4}$$

You now have enough to build a basic simulator of rocket flight.

3.3 Exercise Set 3a

Assuming constant thrust and constant mass and using the parameters tabulated in Figure 3.1

1. Design, build and test Matlab code which simulates a rocket launch, using the constant acceleration model of Equation 3.4. Write a script, `launchEx3.m`, specifying the initial conditions at the top so that they can be easily changed. Write a function `GetConstAcceleration()` Remember that the function has to be put in a file with the name `GetConstAcceleration.m`
2. Add code to plot a graph of height verses time for the rocket.
3. Now move that graphics code into its own function `PlotTelemetry()` in file `PlotTelemetry.m`. The function will have parameters, of course, but no return value.
4. Modify the function to produce a single figure with two subplots, one above the other (use the subplot command), plotting position in red and velocity in blue.
5. Give the figure a title and label all axes
6. Using the Matlab "text" command (use "help text") have your code automatically annotate the velocity/time graph with the instant the rocket exceeds Mach1 — assumed a constant 320 ms^{-1}.
7. I notice that NASA fires the motors at negative t and lifts off at $t = 0$, while the Russians fire the motors at $t = 0$ and lift off at positive t. Have the vehicle launch at $t = 5$s rather than at $t = 0$.
8. Copy the `launchEx3.m` script into `launchH.m`, and `GetConstAcceleration` into `GetHAcceleration.m`, and modify them to determine the horizontal displacement at which the rocket exceeds the speed of sound if, instead of launching vertically, the rocket launched horizontally.

3.4 Increasing Simulation Complexity

Now return to vertical flight! The code you have just written can be made more realistic by considering:

- A rocket's mass does not remain constant.
- After a time, it runs out of fuel.
- The faster a rocket moves, the greater the air resistance.
- The density of the air changes with height.

An analytic approach would be daunting — you could write down the differential equation but would have no hope of solving it.

However, the numerical approach taken so far can very easily account for these. Assume the rocket is flying vertically once more, and begin by writing down the force balance on the rocket (neglect the change in acceleration due to gravity):

$$M(t)\,a(t) = F(t) - D(t, v, h) - M(t)\,g \tag{3.5}$$

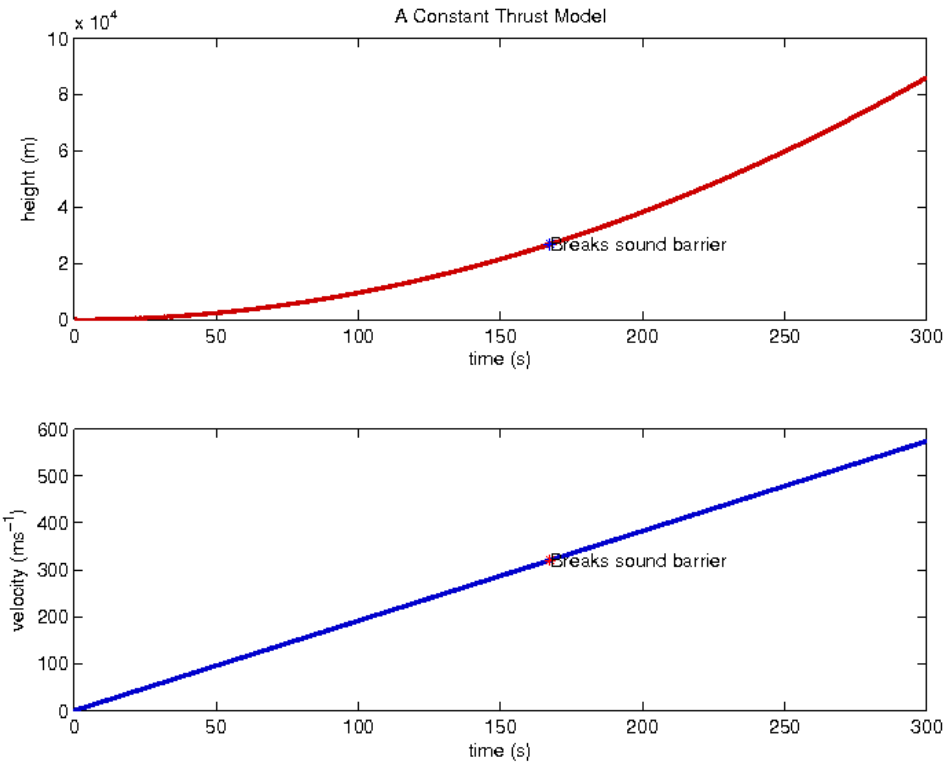

Figure 3.2: Expected output at the end of task 5 of Exercise Set 3a.

where $F(t)$ is the thrust at time t, $M(t)$ is the mass, and $D(t, v, h)$ is the drag on the rocket at time t when travelling at velocity v at height h. Now just as before we can solve for the instantaneous acceleration:

$$a(t) = \frac{F(t) - D(t, v, h) - M(t)g}{M(t)} \ . \tag{3.6}$$

The acceleration at a time t (or time step k) must be some some complicated function of time. This is no problem — you simply evaluate the numerical value of each of the terms on the right hand side at each step of the iteration.

Instead of `GetConstAcceleration()`, you need a more general `GetAcceleration()` function which now takes three parameters — time, height and velocity.

Note that the graphics code is assumed to be in a function `PlotTelemetry(T,H,V,A)`.

Listing 3.3: A modified iteration loop, script launchEx4.m

```
% Initial conditions
  v      = 0;
  h      = 0;
% Times and seasons
  tstart= 0;
  dt     = 0.1;
  tstop  = 300;
  k      = 0;     % Clock tick
% loop through all times
```

```
for t = tstart:dt:tstop
  a = GetAcceleration(t,h,v);  % get acceleration
  % capture telemetry for plotting
  k=k+1; A(k)=a; V(k)=v;  H(k)=h;  T(k)=t;
  % update velocity and height
  h = h + dt * v;
  v = v + dt * a;
end
% graphics
  PlotTelemetry(T,H,V,A);
% save plot as colour postscript
  print('-depsc', 'launchEx4.eps');
```

3.4.1 GetAcceleration()

Now you can turn to filling in the details of the more sophisticated `GetAcceleration()` function. Write this in a separate file and use Matlab's ability to declare "helper" functions in the same file to keep things neat. Listing 3.4 shows some skeleton code which you'll soon be asked to fill out.

Listing 3.4: File *GetAcceleration.m*. The first function declared is a high level function which calls other functions declared locally in the file to evaluate instantaneous Mass, Thrust and Drag. Finally it returns the instantaneous acceleration. You will need to fill in the details.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function a = GetAcceleration(t,h,v);
  g = GetGravity(h);
  f = GetThrust(t);
  m = GetMass(t);
  d = GetDrag(t,h,v);
  a = (f - m*g -d)/m;    % Newton Second Law

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function thrust = GetThrust(t)
  burnTime = 150;
  if( t>0 & t<burnTime )
    thrust = ?
  else
    thrust = ?
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function mass = GetMass(t)
  initialMass     = ?;    % kg
  initialFuelMass = ?;    % kg
  burnTime        = ?;
  burnRate        = ?;
```

```
  if ( t<=0 )
    mass = ?;
  elsif (t>0 & t<burnTime)
    mass = ?;
  else
    mass=  ?;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function dragcoeff = GetDragCoeff(t,h)
% Assume a constant drag coefficient
% (but we could make it a function of height etc)
  dragcoeff = 0.6;  % Nm^2s^-2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function drag = GetDrag(t,h,v)
% calculate a quadratic drag
% NB! When v is +ve (up) drag should be +ve (down)
  k    = GetDragCoeff(t,h);
  drag = k * ?

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function g = GetGravity(height)
% ignore variation with height!
  g = 9.81;
```

The function GetDrag() implements a very simple drag model — the resistance is proportional to the square of the velocity. The constant of proportionality is called a drag coefficient and in the above example is set to 0.6. In the future this function leaves space for us to alter the constant of proportionality as a function of height (as the air thins the resistance drops).

3.5 Exercise Set 3b

1. Using the code in Listings 3.3 and 3.4 as templates, implement a more sophisticated simulation that has
   - The rocket mass decreasing with time but with constant thrust, until
   - the thrust becomes zero when the fuel runs out.
   - A drag term that is quadratic in velocity.
2. Use your PlotTelemetry function to produce on a single figure (using the subplot command) graphs of Height, Velocity and Acceleration against time. As always, make sure your figures have a title and labelled axes!
3. Try runs with different drag coefficients.

4. Modify PlotTelemetry to produce two additional plots of rocket mass and thrust against time. You will have to create extra vectors to store the information. But take care! Would this work if the `GetMass()` and `GetThrust()` functions are in the `GetAcceleration()` function file?

```
k=k+1;
A(k)=a; V(k)=v;  H(k)=h;  T(k)=t;
M(k) = GetMass(t);
Th(k) = GetThrust(t);
```

   The answer is, of course, no! How are you going to fix this problem?

5. This is a good point to step through your code, in and out of functions using the IDE (integrated development environment).

   Make sure you can reconcile what appears as you type 'whos' at the command prompt or equivalently what appears in /vanishes from the variables window with your understanding of functions and variable scoping.

   Remember within a function, only local variables and parameters passed to the function are visible).

3.6 Tidying the code using Structures

You probably found that that last task required you to embed important magic numbers deep within your code — like initial vehicle mass for example. This is always bad news. What if you wanted to simulate a different rocket — you would have to remember to delve into the `GetAcceleration.m` file and edit the `GetMass()` function — which is profoundly messy.

What you would prefer is to supply more parameters to the function `GetAcceleration()`, so that at run time you pass it all the parameters it could ever need. You would then have a function that could be applied to other simulations. That is, something like

```
function acc = GetAcceleration(t,v,h,M0,MFinal,DragCoeff,Area,....)
```

However, that soon starts to look cumbersome.

There is a better way — you can group the parameters into a **structure**, from which you can access an individual variable by using the '.' operator. For example you might choose to put all the static properties of the rocket into a single structure called `SaturnV`. A modified version of the top level iterating function shown in Listing 3.3 is shown in Listing 3.5.

Listing 3.5: A modified iteration loop. Note the initialisation of a `SaturnV` structure and its use in making the end time for the simulation. It this collection of parameters is also passed to a modified version of `GetAcceleration()`.

```
% Saturn V rocket information
  SaturnV.initialMass    = 2.900e6;
  SaturnV.initialFuelMass= 2.150e6;
```

```
  SaturnV.burnTime       = 150;
  SaturnV.initialThrust  = 34.00e6;
% Initial conditions
  v      = 0;
  h      = 0;
% Times and seasons
  tstart= 0;
  dt     = 0.1;
  tstop = 2.0*SaturnV.burnTime;
  k      = 0;    % Clock tick
% loop through all times
for t = tstart:dt:tstop
  a = GetAcceleration(t,h,v,SaturnV);  % get acceleration
  % capture telemetry for plotting
  k=k+1; A(k)=a; V(k)=v;  H(k)=h;  T(k)=t;
  % update velocity and height
  h = h + dt * v;
  v = v + dt * a;
end
% graphics
  PlotTelemetry(T,H,V,A);
% save plot as colour postscript
  print('-depsc', 'launchEx5.eps');
```

Listing 3.6 shows how this structure can be used within the `GetAcceleration.m` file. The parameter structure is passed to the top level `GetAcceleration()` function from your main script.

NOTE 1: At this risk of labouring the point, notice how the parameter name used to refer to a structure can change — we don't have to call it `SaturnV` because each function is passed a *copy* of the structure which it can call what it likes. **If the the use of parameters and scope of local variables isn't crystal clear by now you MUST talk to a demonstrator.**

NOTE 2: Not all the functions which need to be modified are shown below.

Listing 3.6: Example code to show how a structure of parameters can be passed between functions, just like any other parameter.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function a = GetAcceleration(t,h,v,rocket);
  g = GetGravity(h);
  f = GetThrust(t,rocket);
  m = GetMass(t,rocket);
  d = GetDrag(t,h,v,rocket);
  a = (f - m*g -d)/m;    % Newton Second Law

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  function thrust = GetThrust(t,R)
  if( t>0 & t<R.burnTime )
```

```
    thrust = R.initialThrust; % Newtons
  else
    thrust = 0;
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  function ... etc ...
```

3.7 Exercise Set 3c

1. Modify your simulation to use a structure of parameters. No constants (magic numbers) should appear anywhere apart from at the top of your top level simulation script.

2. By changing **just one number** run the simulation for three different values total rocket mass. Check your results make sense by looking at the heights achieved at burn out. Lighter rockets reach higher...

3. Modify your code and relevant functions by introducing a "State" structure which will contain a snap-shot of time varying properties of the rocket for example, current mission time, instantaneous acceleration, velocity and height. So instead of passing t, v, h around you'll pass a single structure which contains these fields.

   (You could of course quite reasonably put t, v, a, h as members of the SaturnV structure. Perhaps having another structure provides more practice — but up to you!)

4. Ask a demonstrator to come round and comment on your code — both on how well it works and on its style.

5. Now modify the simulation so it takes account of the change in density of the air as height increases. Let the drag coefficient vary linearly with air *density*, and work out, or find out, how density varies with height.

6. After the lab, spent time to write down in your log book the overall differential equation you are solving at this point. It is complicated, but the method you have used can be applied to every more involved simulations.

   For example, you have assumed a constant burn-rate and thrust before the motors cut out completely. Suppose that the burn rate varied, but that you knew what it was at each time. Without writing code, think what would change in the problem, and how you might tackle it.

   (In Shuttle launches, for example, the main engines – not the boosters – are throttled back to around 65% of maximum thrust at around $T + 30$s, the point of maximum dynamic pressure on the vehicle, and then throttled up again at $T + 50$s.)