Week 8
Matlab: simulation, functions, control structure
Wright brothers

Invented and built the world’s first powered airplane

Pictures from http://en.wikipedia.org/wiki/Wright_brothers
Crumpled gilder, Oct 1900

Glider (i.e. no power) (1902)
First powered flight (17 Dec 1903)
(Added: Propeller, engine)
Classical engineering design iteration

1. Design
   - This step may use calculations, physical laws, chemistry or biology, experimental data, intuition and guesses
2. Build
3. Test
4. If it doesn’t work, go back to design (Step 1).
Engineering design iteration – with computers

1. Design on computers
   a) Derive **mathematical model** of the design
   b) Perform calculations, **simulations** or optimisation to understand or improve design
   c) Reject designs with poor performance. If none of the designs is good, go back to (a) for a new design or (b) to try to optimise the design.
   d) Choose one or more candidates for prototyping or building the actual design

2. Build
3. Test
4. If it doesn’t work, go back to design (Step 1).

Mathematical model can be derived from science (maths, physics, chemistry, biophysics) or data
Design challenge: Balancing an inverted pendulum

• Can you balance a stick on your finger tip/palm

• An inverted pendulum is sitting on a cart.
• The aim of the design is to balance the inverted pendulum by applying an appropriate force on the cart.

Applications of inverted pendulum

- Segway
- Rocket/spaceship attitude control
  - i.e. orientation control

[http://www.qrg.northwestern.edu/projects/vss/docs/propulsion/2-what-is-attitude-control.html](http://www.qrg.northwestern.edu/projects/vss/docs/propulsion/2-what-is-attitude-control.html)
This week

- Simulation

- Matlab components
  - For
  - Functions
  - Control structures

- Mathematical / physics / chemistry concepts
  - Mathematical modelling
  - Numerical approximation of derivatives
  - Ordinary differential equations
Simulation on paper – the setup

• An object is constrained to move along a straight line
• Time starts at 0 unit. The initial position of the object is \( x(0) = 1 \)
• The velocity \( v(t) \) at time \( t \) is:
  - \( v(t) = 2 \) if \( 0 \leq t < 0.4 \)
  - \( v(t) = -4 \) if \( 0.4 \leq t < 0.8 \)
  - \( v(t) = 1 \) if \( 0.8 \leq t \)
• Determine the position of the object at \( t = 0.1, 0.2, \ldots, 1 \)

At position 1 at time 0
Calculating positions on paper

- Given:
  - Initial position \( x(0) = 1 \)
  - Velocity in time interval \([0,0.1]\) is 2

- Aim: Find the position at time 0.1 = \( x(0.1) \)

\[
x(0.1) = x(0) + 2 \times 0.1 = 1.2
\]

- How about position at time 0.2 = \( x(0.2) \)
  - Velocity in time interval \([0.1,0.2]\) is 2

\[
x(0.2) = x(0.1) + 2 \times 0.1 = 1.4
\]
The aim is to compute the position of the object at time instances 0, 0.1, 0.2, ... , 1

How many time instances are there? 11

Define a Matlab vector called vecTime
- vecTime = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1]
Simulation variable for positions (1)

- \( \text{vecTime} = [0 \ 0.1 \ 0.2 \ 0.3 \ 0.4 \ 0.5 \ 0.6 \ 0.7 \ 0.8 \ 0.9 \ 1] \)

- Define a Matlab vector called \( \text{vecPos} \) (pos is short for position) to store the position of the object at these 11 time instances

- What should \( \text{length(vecPos)} \) be? 11

- \( \text{vecPos}(1) \) = First element in the vector pos
  - Store the position at the first time instance, i.e. time 0
  - \( \text{vecPos}(1) \) = position at time 0
Simulation variable for positions (2)

- time = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1]

- \text{vecPos}(2) = 2^{\text{nd}} \text{ element in the vector vecPos}
  - Store the position at the 2^{\text{nd}} time instance, i.e. time 0.1
  - \text{vecPos}(2) = \text{position at time 0.1}

- \text{vecPos}(5) = \text{position at time 0.4}
  \quad = \text{vecTime}(5)

- Generally:
  - \text{vecPos}(k) = \text{position at time } (k-1) \times 0.1
  - \text{vecPos}(k) = \text{position at time instance vecTime}(k)
Naming convention in the lecture notes (1)

- Mathematical variables \( x(0.4) \) = position at time 0.4

- Matlab vector element \( \text{vecPos}(5) \) stores the position at time 0.4

- Both \( x(0.4) \) and \( \text{vecPos}(5) \) are referring to the position of the object at the same time instance
  - Both have numbers in parentheses
  - But there are some subtle differences
• Mathematical variable: $x(0.4)$

• Matlab variable: `vecPos(5)`

• In the lecture notes:
  - Mathematical variables: always single letter but may have subscripts
  - Matlab variables: Words or abbreviated but always more than one letter
Simulation on paper – the setup (repeat)

- An object is constrained to move along a straight line.
- Time starts at 0 unit. The initial position of the object is $x(0) = 1$.
- The velocity $v(t)$ at time $t$ is:
  - $v(t) = 2$ if $0 \leq t < 0.4$
  - $v(t) = -4$ if $0.4 \leq t < 0.8$
  - $v(t) = 1$ if $0.8 \leq t$
- Determine the position of the object for $t = 0.1, 0.2, \ldots, 1$.
## Simulation on paper

<table>
<thead>
<tr>
<th>Index k</th>
<th>time(k)</th>
<th>Speed in the interval before t</th>
<th>Position pos(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>vecPos(1) = 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>vecPos(2) = vecPos(1) + 2 * 0.1 = 1.2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
<td>vecPos(3) = vecPos(2) + 2 * 0.1 = 1.4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.3</td>
<td>vecPos(4) = vecPos(3) + 2 * 0.1 = 1.6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.4</td>
<td>vecPos(5) = vecPos(4) + 2 * 0.1 = 1.8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>vecPos(6) = vecPos(5) - 4 * 0.1 = 1.4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.6</td>
<td>vecPos(7) = vecPos(6) - 4 * 0.1 = 1.0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.7</td>
<td>vecPos(8) = vecPos(7) - 4 * 0.1 = 0.6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.8</td>
<td>vecPos(9) = vecPos(8) - 4 * 0.1 = 0.2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.9</td>
<td>vecPos(10) = vecPos(9) + 1 * 0.1 = 0.3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1.0</td>
<td>vecPos(11) = vecPos(10) + 1 * 0.1 = 0.4</td>
<td></td>
</tr>
</tbody>
</table>

Matlab implementation in simulateOneD_v1.m
for k = 2:length(vecTime)
    pastTime = vecTime(k-1);
    if pastTime < TIME_LIMIT_1
        velocity = VELOCITY_1;
    elseif pastTime < TIME_LIMIT_2
        velocity = VELOCITY_2;
    else
        velocity = VELOCITY_3;
    end

    vecPos(k) = vecPos(k-1) + velocity * dt;
end
Selection

- Matlab has **if**, **switch** and **try/catch**
  - only **if** is essential for ENGG1811
- Same principle as for OOB but syntax differs slightly 😞

```matlab
if boolexpr_1
    statements_1
elseif boolexpr_2
    statements_2
elseif ...
else
    statements_n
end

% example (max of a and b)

If a < b
    max = b;
else
    max = a;
end
```
Relational and Boolean operators

- **Boolean constants** are `true`, `false`
  - internally represented by 1 (or nonzero) and zero
- **Relational operators** are almost the same, except
  - `==` is the *equality* operator for numbers (OOB: `=`)
  - `~=` is the *inequality* operator for numbers (OOB: `<>`)
- **Boolean operators** also use a different notation
  - `&&` is the *and* operator (OOB: `And`)
  - `||` is the *or* operator (OOB: `Or`)
  - `~` is the *negation* operator (OOB: `Not`)
Iteration – for

- Because Matlab is heavily array-based, the for loop variable is assigned each value in an array in turn:

```matlab
for index = ArrayExpression
    statements
end
```

- The `ArrayExpression` is typically a vector generated with the colon operator
- the index variable is set to each value in turn (or each `column` if it’s a matrix)
- textbooks tend to use `ii` for row index variables and `jj` as column index variables, reflecting the $M_{ij}$ mathematical notation but `i` and `j` are used to define complex numbers
- If you use `i` or `j` as the index, you need to be aware that the default definition of $i = j = \sqrt{-1}$ is overwritten
Quiz: How can we improve `simulateOneD_v1.m`?

Hint: This part can be improved.

What is your suggestion?

Should use a Function!

```matlab
for k = 2:length(time)
    pastTime = time(k-1);
    if pastTime < TIME_LIMIT_1
        velocity = VELOCITY_1;
    elseif pastTime < TIME_LIMIT_2
        velocity = VELOCITY_2;
    else
        velocity = VELOCITY_3;
    end
    pos(k) = pos(k-1) + velocity * dt;
end
```
Functions

- Functions come in different flavours
  - `global` functions in an M-file with matching name
  - `local` functions (or subfunctions), inside a function M-file
  - `nested` functions (not used in ENGG1811)
  - `anonymous` functions (not used in ENGG1811)
  - function `handles` (useful in advanced programming, not in ENGG1811)

- Declaration is:
  ```matlab
  function [outputs...] = functionname(inputs)
  function body, using local variables
  end
  ```

  Tip: keep the [] even when there's only one return variable, avoids mistaking it for the function name.

- inputs are parameters, `passed by value`
- outputs can be any number of variables, including none
simulateOneD_v2.m (simulation loop)

```matlab
for k = 2:length(time)
    pastTime = time(k-1);
    velocity = getVelocity(pastTime);
    pos(k) = pos(k-1) + velocity * dt;
end
```
function velocity = getVelocity(currentTime)

% Comments and constant definitions not shown

    if currentTime < TIME_LIMIT_1
        velocity = VELOCITY_1;
    elseif currentTime < TIME_LIMIT_2
        velocity = VELOCITY_2;
    else
        velocity = VELOCITY_3;
    end % end of if

end % end of function
Local functions (subfunctions)

- If a function has a subtask that’s not general enough for its own file, add a local function to the end
  - accessible only within the parent file
  - regrettably doesn’t apply to scripts 😞 (lazy people don’t want to create extra files, so they tend to bloat the script)

```plaintext
function [retval] = dostuff(args)  % consists of three subtasks
  subtask1
  var1 = subtask2func(arr)
  subtask3(var1)
  ...
end

function [ ] = subtask1()  % no return value ~= OO Basic sub
  ...
end

function [retval] = subtask2func(vec)
  ...
end

function [ ] = subtask3(yyy)
  ...
end
```
simulateOneD_v3.m (uses subfunction)

- Script runSimulateOneD_v3.m
  - Only the line calling the function simulateOneD_v3 is shown

```matlab
% Call the function
[vecPos] = simulateOneD_v3(vecTime,pos0);
```

- Function simulateOneD_v3.m
  - Only the key structures are shown

```matlab
function [pos] = simulateOneD_v3(time,initialPosition)
end % end of function simulateOneD_v3

function speed = getVelocity(currentTime)
end % end of subfunction getVelocity
```
Variable scooping in Matlab

- Matlab functions are passed by value
- By default, all variables are local
- Script and functions have different memory space
- Base space is for scripts
- Each function has the same memory space
- If the same variable name is used in different functions, these variables have nothing to do with each other
Scope of the variables

- Separate memory space for each function
- Variables are local

Code: quadruple.m

```matlab
function y = quadruple(x)
    y = 2*double(x);
end

function y = double(x)
    y = 2*x;
end
```

Memory space for function quadruple:

<table>
<thead>
<tr>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>12</td>
</tr>
</tbody>
</table>

Memory space for function double:

<table>
<thead>
<tr>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>6</td>
</tr>
</tbody>
</table>
### Contrasting two methods to do simulation

<table>
<thead>
<tr>
<th>By increment</th>
<th>By Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>vecPos(k) = vecPos(k-1) + velocity * dt</td>
<td>$v(t) = v_0 e^{-\frac{c_d}{m} t} + \frac{g m}{c_d} (1 - e^{-\frac{c_d}{m} t})$</td>
</tr>
</tbody>
</table>
• The parachutist example is taken from Week 5’s lecture on OO Basic. We didn’t have time to cover this example.

• It’s also an example of simulation but it uses an entirely different approach. It uses a formula to calculate the speed of the parachutist. We will come back to this difference later on. It’s important.
Example: Parachutist Velocity Profile

Velocity of an object of mass $m$ falling under the influence of gravity and subject to drag is

$$v(t) = v_0 e^{-(c_d/m)t} + \frac{g m}{c_d} (1 - e^{-(c_d/m)t})$$

Where $v_0$ is the initial velocity, $g$ is acceleration due to gravity (m s$^{-2}$) and $c_d$ is the drag coefficient (in kg s$^{-1}$)

The exponential factor decays in magnitude, so the velocity asymptotically approaches $g \frac{m}{c_d}$

For a free-falling 70kg parachutist with $c_d = 12.5$, this terminal velocity is $\sim 55$ m s$^{-2}$ (200km/hr)
When the parachutist pulls the rip-cord \((t = t_c)\), the same equation applies subsequently except that...

- \(v_0\) is the velocity at the instant the parachute opens
- there is a new, larger, drag coefficient \(c_{dp}\)
- the time parameter is \(t - t_c\) instead of \(t\)

![Parachutist's Velocity Chart](image-url)
Speed with parachute

If $t \leq t_c$

$$v(t) = v_0 e^{-\frac{c_d}{m} t} + \frac{g m}{c_d} (1 - e^{-\frac{c_d}{m} t})$$

Free fall part

If $t \geq t_c$

$$v(t) = v_{p0} e^{-\frac{c_d}{m} t} + \frac{g m}{c_d} (1 - e^{-\frac{c_d}{m} t})$$

Speed at the moment parachute is deployed

$$v_{p0} = v_0 e^{-\frac{c_d}{m} t_c} + \frac{g m}{c_d} (1 - e^{-\frac{c_d}{m} t_c})$$

With Parachute
Parachutist Simulation

- We can write a function that, given all parameters, calculates the velocity at any time $t$
- The algorithm, expressed in \textit{pseudocode}, is

```plaintext
for t = time vector
  if t < tc % still in free-fall
    Calculate free-fall velocity
  else
    Calculate velocity at time tc
    Store in velocityChute
    Calculate velocity using cdp and velocityChute
  end
end
```

\textbf{Code in parachutistSpeedByFormula.m}
Validating inputs

- To avoid misleading or meaningless results, always ensure that the problem instance parameters are within the limits of the model
  - the `error` function works like `fprintf`, but displays the formatted message and where it occurred, then stops
  - may not be practical to check *all* data values, especially a large externally supplied array

```matlab
if numDataValues < 0
    error('expected non-negative value for data size, \%d was specified\n', numDataValues);
end
```
Checking the number of input/output arguments

- You can use `narginchk` to check whether the number of input arguments is within the specified limit when the function is called
  - For example, if a function expects a minimum of 2 and a maximum of 5 input arguments, then `narginchk(2,5)` returns an error if the number of input argument is not from 2 to 5
- Similarly, `nargoutchk` for checking the number of output arguments
- If you find these names cryptic, this may help:
  - number arguments input check
  - number arguments output check
Comparing object moving in 1D and parachutist

<table>
<thead>
<tr>
<th></th>
<th>By increment</th>
<th>By Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pos(k) = pos(k-1) + velocity * dt</td>
<td>• If 0 ≤ t ≤ 0.4, x(t) = 1 + 2t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If 0.4 ≤ t ≤ 0.8, x(t) = 1.8 − t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If t ≥ 0.8, x(t) = 0.2 + t</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td>v(t) = v₀e^{−\frac{c_d}{m}t} + \frac{gm}{c_d}(1 − e^{−\frac{c_d}{m}t})</td>
</tr>
</tbody>
</table>
An inconvenient truth

• Solving problems by deriving a formula
  – Mathematically elegant; exact solution
  – Formulas may provide insight
  – Convenient to use: simply perform substitution

Bad news!
Most advanced engineering problems do not have an exact solution in the form of a formula

Good news!
You can solve these problems numerically and approximately by computers and programming
The speed of the parachutist obeys the following ordinary differential equations (ODE):

\[ \frac{dv(t)}{dt} = g - \frac{c(t)}{m} v(t) \]

- \( v(t) \) = speed at time \( t \)
- \( c(t) \) = drag coefficient at time \( t \)

We will look at how you can solve this equation numerically and approximately.
Approximating derivatives

• From the definition of derivatives, we know

\[
\frac{dv(t)}{dt} = \lim_{\Delta \to 0} \frac{v(t + \Delta) - v(t)}{\Delta}
\]

• If \( \Delta \) is small enough, then

\[
\frac{dv(t)}{dt} \approx \frac{v(t + \Delta) - v(t)}{\Delta}
\]
Approximating derivatives – numerical illustration

- \( f(x) = x^3 \)
- Derivative of \( f(x) = f'(x) = 3x^2 \)
- At \( x = 2 \), \( f'(2) = 12 \)
- Let us compute the approximate derivative for different values of \( \Delta \)

\[
\frac{(2 + \Delta)^3 - 2^3}{\Delta}
\]

Code: approximateDerivative.m
Solving ODE numerically (1)

1) Starting from the ODE
\[
\frac{dv(t)}{dt} = g - \frac{c(t)}{m} v(t)
\]

2) Replace the derivative by its approximation
\[
\frac{dv(t)}{dt} \approx \frac{v(t + \Delta) - v(t)}{\Delta}
\]

We obtain:
\[
\frac{v(t + \Delta) - v(t)}{\Delta} \approx g - \frac{c(t)}{m} v(t)
\]
Solving ODE numerically (2)

Previous step:

\[
\frac{v(t + \Delta) - v(t)}{\Delta} \approx g - \frac{c(t)}{m} v(t)
\]

3) Make \(v(t + \Delta)\) the subject:

\[
v(t + \Delta) \approx v(t) + \left( g - \frac{c(t)}{m} v(t) \right) \Delta
\]

• For simulation, let us assume velocity is stored in the vector velocity
• Identify \(v(t+\Delta)\) with velocity\((k)\), and \(v(t)\) with velocity\((k-1)\)
Matlab code: approx ODE versus formula

- We will write a Matlab function to solve the ODE numerically for the parachutist problem
  - Solution in the function: parachutistSpeedByODE.m

- Note
  - Formula is exact
  - Numerical solution to ODE is an **approximation**

- Matlab script compareParachutistMethods.m compares the formula against the approximate numerical solution

- We will vary the value of $\Delta$, we expect
  - Small $\Delta$, small difference between the two methods
  - And vice versa
Where did the ODE come from?

ODE we used. Multiply both sides by \( m \).

\[
\frac{dv(t)}{dt} = g - \frac{c(t)}{m} \cdot v(t)
\]

Let us look at what this means.

\[
m \frac{dv(t)}{dt} = mg - c(t) \cdot v(t)
\]
ODEs describe physical laws

\[ m \frac{dv(t)}{dt} = mg - c(t)v(t) \]

mass x acceleration = Net downward force on the parachutist

\[ c(t) \ v(t) = \text{drag force} \]

\[ m \ g = \text{gravitational pull} \]
The big picture

• Physical law gives the ODE

\[ m \frac{dv(t)}{dt} = mg - c(t)v(t) \]

• Computers and algorithms allow you to obtain numerical and approximate solution

• That’s why you need to learn maths, physics, chemistry, your own disciplinary knowledge and COMPUTING!
Solving ODEs

- The method we use for solving ODE is known as **Euler’s forward method**
- Meaning of forward and backward:
  
  \[
  \frac{dv(t)}{dt} \approx \frac{v(t + \Delta) - v(t)}{\Delta}
  \]
  
  Forward:

  \[
  \frac{dv(t)}{dt} \approx \frac{v(t) - v(t - \Delta)}{\Delta}
  \]
  
  Backward:

  Euler’s forward method is simpler to explain but **not the best**. This is so you can focus on learning programming

- You will learn better methods in later years
The extended parachutist problem

- What if you want to determine the height of the parachutist too?
- Let $h(t) = \text{height of the parachutist at time } t$
- How can you compute $h(t + \Delta)$ from $h(t)$?

\[
\frac{dh(t)}{dt} = -v(t)
\]

- You can formally derive this from the following ODE which says: derivative of height = downward speed

\[
h(t + \Delta) \approx h(t) - v(t)\Delta
\]
Matlab implementation

- Essentially, two updates in the for loop

\[
v(t + \Delta) \approx v(t) + \left( g - \frac{c(t)}{m} v(t) \right) \Delta
\]

\[
h(t + \Delta) \approx h(t) - v(t) \Delta
\]

- Matlab function: parachutistSpeedHeightByODE.m
- Matlab script: runParachutistSpeedHeightByODE.m
  - The script also illustrates the function plotyy for using 2 different scales for the y-axis
function [vecSpeed, vecHeight] = parachutistSpeedHeightByODE(vecTime, mass, ...
    speed0, height0, dragAir, tParaDeploy, dragPara)

    vecHeight = zeros(size(vecTime));

    vecHeight(1) = height0;

    % simulation loop
    for k = 2:length(vecTime)
        vecHeight(k) = vecHeight(k-1) – vecSpeed(k-1) * dt;
    end
Summary

- We have introduced the basics of simulation, which is a key tool in modern engineering and science.
- The basic method to do simulation is to set up an iteration step which can be obtained from ordinary differential equations.
- Matlab component covered: control structure, for, functions, error checking.