# Mathematics for the Life Sciences (MATH1134) - Tutorial Sheet 4

`This tutorial complements the material covered in Lecture 4.`

1. Do the examples in Lecture 4.

2. *(Optional)* It is possible to write vectorised forms of the numerical methods for systems of IVPs, which can be implemented in MATLAB essentially just like the corresponding scalar methods. I will outline how this could be done and then you should do it for Euler's, Heun's and the fourth order Runge-Kutta methods and test your program on the system of IVPs in EXAMPLE 17 of **Lecture 4**.

   For convenience, we will use *column* vectors instead of row vectors as the default vector type.

   (a) Write your $\vec{f}(t, \vec{y})$ so that its output is a (column) vector. For example

   ```
   f = @(t,y) [-4*y(1) - 2*y(2) + cos(t) + 4*sin(t);   3*y(1) + y(2) - 3*sin(t)];
   ```

   (b) Store the approximate solutions in a 2-dimensional array (matrix) $Y$ so that $Y(i,j)$ is the approximate solution of function $y_i$ at timestep $t_j$ - *i.e* $Y(i,j) \approx y_i(t_j)$, for $i = 1, 2, \ldots, n$ and $j = 0, 1, 2, \ldots, N$. In MATLAB notation, when $Y$ is full, then

   - $Y(i,:)$ will be a row vector of length $N+1$ containing all of the approximations to the function $y_i(t)$ at the $N+1$ times $t_0, t_1, \ldots, t_N$.
   - $Y(:,j)$ will be a column vector of length $n$ containing the approximations to all of the solution functions at time $t_j$. So, the first entry of this vector will be the approximation to $y_1(t_j)$, the second entry will be the approximation to $y_2(t_j)$, etc.

   $\hookrightarrow$*(do you see the logic behind this notation choice???)*$\hookleftarrow$

   (c) Obviously then, assuming the initial conditions are in a vector $\vec{y}_0$, these values have to be stored in the matrix $Y$ with a loop such as

   ```
   for k = 1:n
       Y(k,1) = y0(k);
   end
   ```

   (d) It will then be possible to write your main loop in a very similar way to the scalar case, where at each time step *every* approximate function $Y_1, Y_2, \ldots, Y_n$ is updated. For example, for Euler's method this would look like this:

   ```
   for k = 1:N
       Y(:,k+1) = Y(:,k) + h*f(t(k), y(:,k));
   end
   ```

3. See http://terpconnect.umd.edu/~petersd/246/matlabode2.html and follow the instructions there for how to solve a system of ODEs using Matlab's in-built function `ode45`, how to plot trajectories in the phase plane, and how to plot direction fields for systems of 2 ODEs. Note I provide a slightly modified version of vectfield.m on the course Moodle page called

new_vectfieldarrow_funchandle.m (click the *A Direction Field Plotter for Systems of 2 Differential Equations* link to download it) so please use that instead. Once you have downloaded it, type `new_vectfieldarrow_funchandle.m` then hit RETURN or ENTER on the Matlab command window to see detailed instructions of how to use the function, including a full example. If you save that full example into a script M file then you would only need to modify the f and possibly the y1 and y2 vectors to generate a direction field for another system of differential equations.

Note unless you have the Matlab Symbolic Toolkit, you will not be able to use `dsolve` to find symbolic solutions.

4. For EXAMPLES 5, 6, 7, 8, 11, 12, 13, and 14 of **Lecture 4**, use the eigenvalues to classify the zero vector steady states. Verify with appropriate directon field plots.

5. Find and classify all steady states for the nonlinear system

$$\begin{pmatrix} \frac{dy_1}{dt} \\ \\ \frac{dy_2}{dt} \end{pmatrix} = \begin{pmatrix} y_1^2 - y_2^2 - 1 \\ 2y_2 \end{pmatrix}.$$

6. `ode45` *(see Tutorial 3)* and several other related in-built Matlab ODE solvers can also be used to solve SYSTEMS of initial value problems. As an example, this is how it could be used to solve the system of IVPs in EXAMPLES 17, 19, and 20 of **Lecture 4**:

$$\begin{array}{rcl|l} \dfrac{dy_1}{dt} & = & -4y_1 - 2y_2 + \cos(t) + 4\sin(t) & \text{(EXACT SOLUTION)} \\ \dfrac{dy_2}{dt} & = & 3y_1 + y_2 - 3\sin(t) & y_1(t) = 2e^{-t} - 2e^{-2t} + \sin(t) \\ & & t \in [0,2], \quad y_1(0) = 0,\ y_2(0) = -1 & y_2(t) = -3e^{-t} + 2e^{-2t} \end{array}$$

   - Define the function handle `f`:

     ```
     f = @(t,y) [-4*y(1)-2*y(2)+cos(t)+4*sin(t); 3*y(1)+y(2)-3*sin(t)];
     ```

   - Create a vector of $t$ values at which you wish to approximate the solutions to the system of IVPs:

     ```
     tspan = 0:0.1:2;
     ```

   - Create a vector containing the initial conditions:

     ```
     y0 = [0; -1];
     ```

   - Call `ode45` as follows:

     ```
     [t, y] = ode45(f, tspan, y0);
     ```

   The solutions should be stored in matrix $y$ with the first column of $y$ containing the approximations to the solution function $y_1(t)$ and the second column of $y$ containing the approximations to the solution function $y_2(t)$, etc. Also, $t$ will be a time vector containing the same entries as *tspan*. Thus if you wanted to plot the (in this case two) approximate solutions on the same axes you could use the command

   ```
   plot(t, y(:,1), t, y(:,2))
   ```

(or you could use *tspan* instead of *t* in the above command).

NOTE if you are not concerned about getting the approximations at specific timesteps, you can instead replace *tspan* in the `ode45` argument list with just a vector indicating the initial and ending values of the *t* interval. Matlab will then decide at what times to approximate the function and return those times in the vector *t*. For example,

```
[t, y] = ode45(f, [0 2], y0);
```

for me returned approximations at 60 times (hence a *t* vector containing 61 entries).

NOTE again if you store this sequence of commands in a script M file, you can then easily use `ode45` to solve systems of ODEs and plot the solutions (or output it in a table, perhaps using the `disp()` or `fprintf()` command), with only minor modifications required to the M file to get it to work for another system of ODEs