7 Supplemental Code Files

There are eleven supplemental code files provided. In order to use these files in a script or a Live Script, they must be placed in the same folder as the script file, so that the Current Folder window contains both the file being executed and all of these function files. Another option would be to store all of these function files in a single folder, navigating to that folder in the MATLAB Current Folder window, right-clicking on the folder, and selecting “Add to Path.” The first of these is more recommended, but the second can also work if there is a common repository to store all of the users custom MATLAB functions. The function headers are given below along with a brief description of their use.

```
function quiver244(f, t_min, t_max, y_min, y_max, col)
% quiver244.m
% Author: Matt Charnley
%
% This function draws a quiver plot for the ODE dy/dt = f(t,y) for
% t_min <= t <= t_max and y_min <= y <= y_max. The function f should be
% passed in as an anonymous function, of two variables or as a function
% handle
%
% The function draws this quiver plot in color col and saves it on the
% current figure, and generates a normalized version
% (all vectors are the same length) as the next figure,
% so that it can be accessed outside of this function.
% For this second figure, the magnitude of the arrows does not mean
% anything, but it is easier to see the direction of them.
% so that it can be accessed outside of this function. It will start with
% hold on; and end with hold off;, so the figure needs to be cleared in the
% main file if needed.
```

The main point of this function is to simplify the process of drawing quiver plots. The code here takes care of the difficulties that arise from the built-in `quiver` function in MATLAB and allows the user to input the right-hand side of a first order ODE and generate quiver plots. It will draw a quiver plot in the first figure, and a normalized quiver plot (all vectors the same length) in the second figure. It can sometimes be easier to see the general trajectory of solutions from the normalized figure, so both graphs are provided. All of the plotting commands use the `hold` commands so that they will not overwrite anything on the desired figures. This allows the overlaying of multiple plots, but means that the code calling this method must clear the figure if it needs to be cleared.

This code can be used as

```
f = @(t,y) t - exp(y);
quiver244(f, 0, 5, -6, 6, 'b');
```

```
quiver244(@f2, 0, 5, -6, 6, 'b');

function z = f2(t,y)
    z = t - exp(y);
end
```

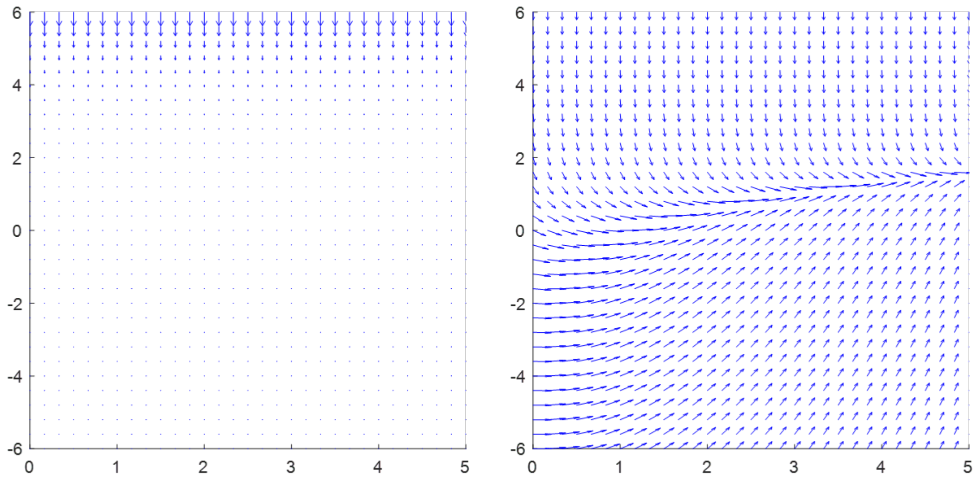

Figure 6: Sample output from the `quiver244` function.

In each case, the ‘`b`’ indicates that the quiver plot will be drawn in blue, and the 1 before that indicates that the two plots will be drawn on figures 1 and 2.

```
function samplePlots244(f, t_min, t_max, y_min, y_max, t_0, y_0, col)
% This function takes the ODE dy/dt = f(t,y) and plots sample solutions
% with initial value (t_0, y_0). It uses ode45 to sketch out the solutions.
% t_0 must be between t_min and t_max. It also truncates the function f so
% that functions will not go off to infinity, causing this to work properly
% on vector inputs for initial conditions in y. The input y_0 can be a
↪  vector
% of initial values, and this function will plot a curve
% for each of those values. If using a vector of initial
% conditions, the function must be written with vector element-wise
% operations.
```

This function follows the same setup as `quiver244`, but draws sample trajectories of the solution instead of the quiver plot. It will take initial conditions as (t_0, y_0). For a single t_0, a vector of initial y_0 values can be passed in and the function will work correctly. This function can be used as

```
f = @(t,y) y.*(y-5).*(y+6);
samplePlots244(f, -1, 6, -7, 6, 0, [-1,0.5,4,5], 'r')
```

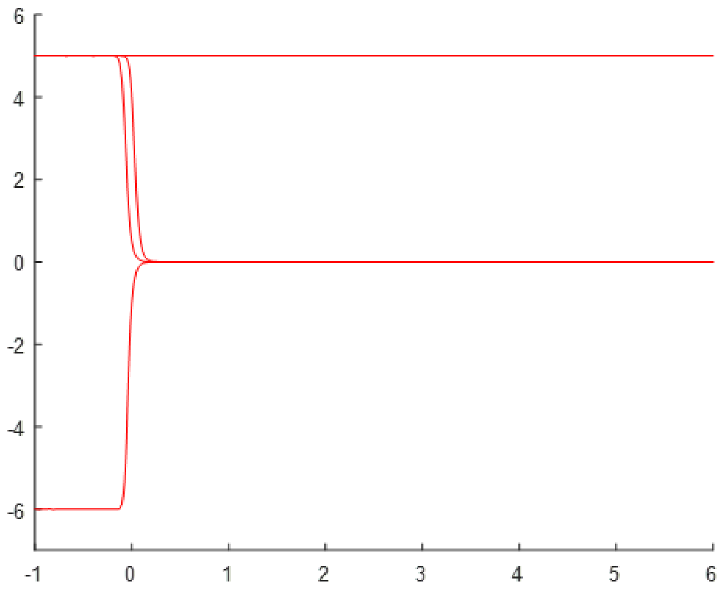

Figure 7: Sample output from the `samplePlots244` function.

The '`r`' here indicates that this plot will be drawn in red and put on figure 2. If this is combined with the `quiver244` method, then it will overlay these red curves on top of the quiver plot drawn on figure 2.

```
function bifDiag244(f, a_min, a_max, y_min, y_max)
% This function draws a bifurcation diagram for the ode dy/dt = f(alpha, y)
% with parameter alpha running from a_min to a_max. The axes are
% constrained to be from a_min to a_max in the horizontal direction and
% y_min to y_max in the vertical direction.
%
% The black marks are for equilibrium solutions, the blue regions are where
% the solution will tend upwards, and the red region is where it will tend
% downwards.
```

This function will draw a bifurcation diagram for the given differential equation. **Note:** This function will need the optimization tool-box add-on for MATLAB in order to run correctly. As with the previous methods, it will not overwrite the figure. Example implementation:

```
f = @(a,y) y.^2 - a.^2;
bifDiag244(f, -3, 3, -5, 5, 3);
```

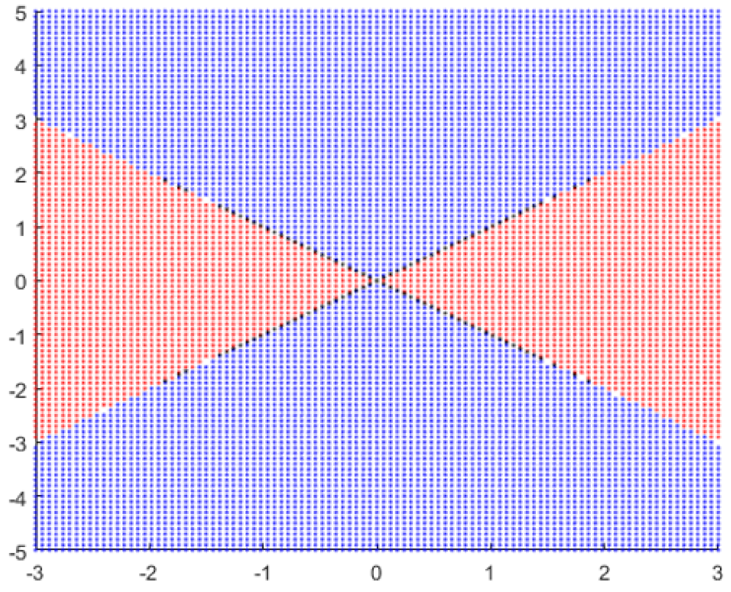

Figure 8: Sample output from the `bifDiag244` function.

```
function quiver2D244(f,g, x_min, x_max, y_min, y_max, col)
% quiver2D244.m
% Author: Matt Charnley
%
% This function draws a quiver plot for the ODE dx/dt = f(x,y), dy/dt =
↪  g(x,y) for
% x_min <= x <= x_max and y_min <= y <= y_max. The functions f and g should
↪  be
% passed in as an anonymous functions, f = @(x,y) ...
%
% The function draws this quiver plot in color col in the current figure
% and generates a normalized version (all vectors are the same length)
% as the next figure, so that it can be accessed outside of this function.
% For this second figure, the magnitude of the arrows does not mean
% anything, but it is easier to see the direction of them.
%
% It will start with
% hold on; and end with hold off;, so the figure needs to be cleared in the
% main file if needed.
```

This function does the same concept as `quiver244` but for the autonomous system of differential equations

$$\frac{dx}{dt} = f(x,y) \qquad \frac{dy}{dt} = g(x,y).$$

Example implementation:

```
f = @(x,y) 3.*x - 2.*x.*y;
g = @(x,y) 2.*y - 3.*x.*y;
quiver2D244(f,g, 0, 5, 0, 5, 'g');
```

```
function phaseLine(f, ymin, ymax)
% This function draws a representation of the phaseline for the
% differential equation dy/dt = f(y). The graph is drawn from ymin to ymax,
% and looks for solutions to f(y) = 0 in that region to find equilibrium
% solutions. This requires the Optimization Toolbox fsolve to run
% correctly.
```

This function draws a representation of the phase line for an autonomous first order differential equation $\frac{dy}{dt} = f(y)$ from y_{min} to y_{max}. Example implementation:

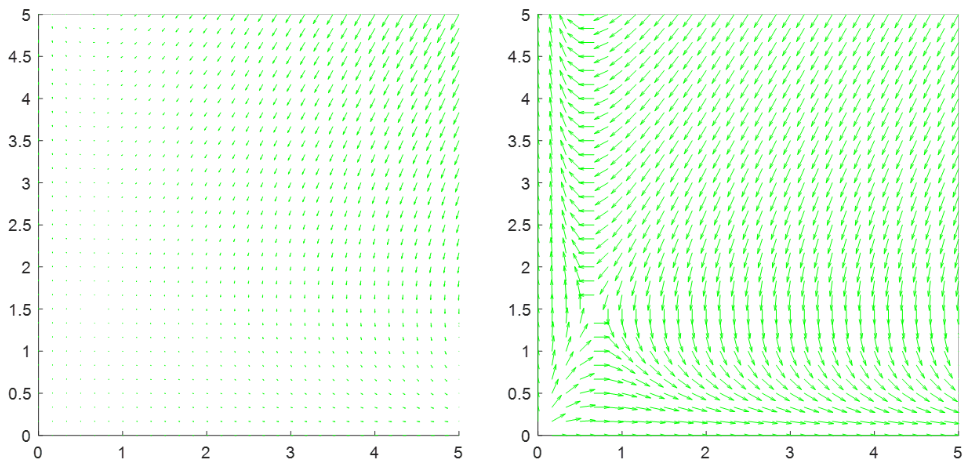

Figure 9: Sample output from the `quiver2D244` function.

```
f = @(y) y.*(y-3).*(y+2);
phaseLine(f, -4, 5);
```

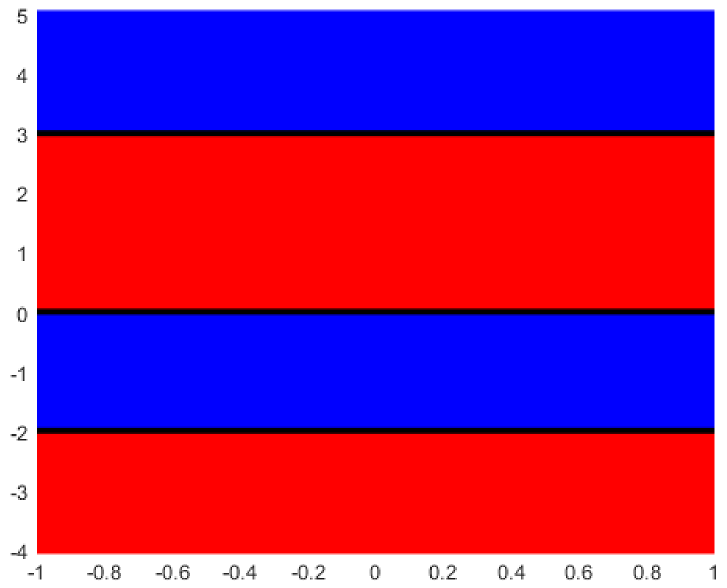

Figure 10: Sample output from the `phaseLine` function.

```
function phasePortrait244(F, G, xmin, xmax, ymin, ymax, tmin, tmax, x0, y0)
% This function draws a 2 dimensional phase portrait for the system dx/dt =
% F(x,y) and dy/dt = G(x,y). The phase portrait will be draw with x bounds
% xmin <= x <= xmax and ymin <= y <= ymax. It is assumed that the initial
% conditions x0 and y0 are at £t=0£, with tmin <= 0 and tmax >=0. x0 and y0
% can be inputted as vectors that are the same length, and a sample curve
% will be drawn for each of them. The black dot will always be plotted at
↪   tmin.
```

This function draws a phase portrait for the two-component autonomous system $\frac{dx}{dt} = F(x, y)$ and $\frac{dy}{dt} = G(x, y)$. The axes are fixed at $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$. Solution curves are drawn starting at the (potential list of) points x_0 and y_0, and will assume these happen at $t = 0$. The curves are drawn from t_min to t_max, and there will be a black dot plotted at t_min to indicate the direction of flow. Example implementation:

```
f = @(x,y) 2.*x - 3.* y;
g = @(x,y) -3.*x + y;
phasePortrait244(f, g, -3, 3, -3, 3, -2, 2, [1, 0, -1, 1, 0, -1],
↪   [1,1,1,-1,-1,-1]);
```

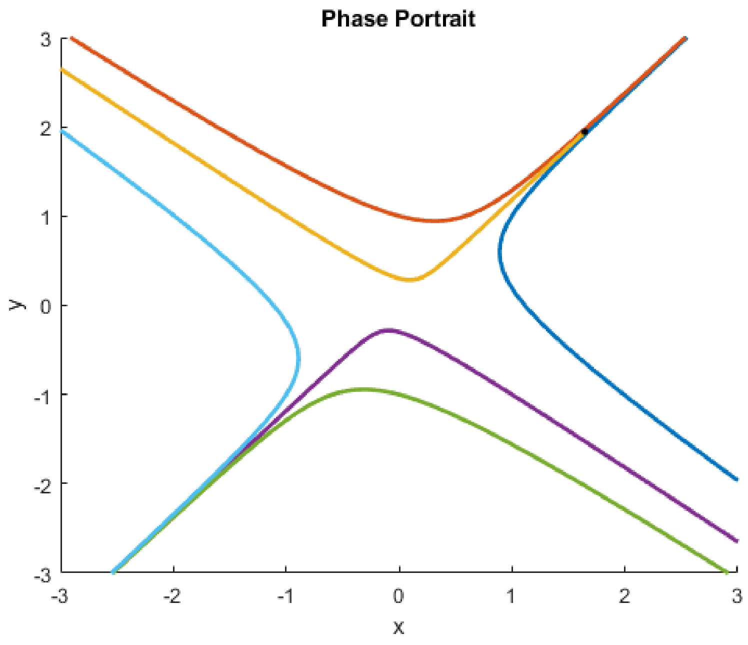

Figure 11: Sample output from the `phasePortrait` function.

```
function [t, y] = rungeKuttaMethod(f, dt, Tf, T0, y0)
% This method solves the ODE dy/dt = f(t, y) using the Runge Kutta method
% from t=T0 to t = Tf with time step dt and initial condition y0 at t = T0.
% In this case, f should be a function of two variables, t
% (time) and y.
```

```
function [t,y] = rungeKuttaSystemMethod(f, T0, Tf, dt, y0)
% This method solves the ODE system dy/dt = f(t, y) using the Runge Kutta
↪  method
% from t=T0 to t = Tf with time step dt and initial condition y0 at t = T0.
% In this case, f should be a vector valued function of two variables, t
% (time) and y (n-dimensional vector of unknowns). The length of the vector
% y0 will determine the size of the system.
```

These two methods use the Runge-Kutta method to numerically solve the differential equation $\frac{dy}{dt} = f(t,y)$ or the system $\frac{d\vec{x}}{dt} = F(t,\vec{x})$. It will return the list of t and y values that are generated by this method.

```
function [S,I,R] = SIRModel_244(r, c, ICs, Tf)
% This code runs an SIR model for disease spread. The system of differential
↪  equations used here is
%   S' = -r*S*I
%   I' = r*S*I - cI
%   R' = c*I
%
% The solution is computed using the RungeKutta method, with the helper
% method rungeKuttaSystemMethod. The system is solved from t=0 to t=Tf,
% with initial conditions ICs given as a 3 component vector.
```

```
function [S,I,Q,R,D] = SIRQModel_244(alpha, beta, gamma, delta, eta, rho,
↪  ICs, Tf)
% This code runs a more complicated SIR model that adds in Q (a quarantined
% population) and D (a deceased population). The system of differential
↪  equations used here is
%   S' = -alpha*S*I
%   I' = alpha*S*I - (beta+gamma+delta)I
%   Q' = beta*I - (eta + rho)Q
%   R' = gamma*I + eta*Q
%   D' = delta*I + rho*Q
%
% The solution is computed using the RungeKutta method, with the helper
% method rungeKuttaSystemMethod. The system is solved from t=0 to t=Tf,
% with initial conditions ICs given as a 5 component vector.
```

```
function [S,I,Q,R,D] = SIRQVModel_244(alpha, beta, gamma, delta, eta, rho,
↪  zeta, ICs, Tf)
% This code runs a more complicated SIR model that adds in Q (a quarantined
% population) and D (a deceased population). The V component adds
% vaccination into the picture, where members are moved from S to R
% directly. The system of differential equations used here is
%   S' = -alpha*S*I - zeta*S
%   I' = alpha*S*I - (beta+gamma+delta)I
%   Q' = beta*I - (eta + rho)Q
%   R' = gamma*I + eta*Q+zeta*S
%   D' = delta*I + rho*Q
%
% The solution is computed using the RungeKutta method, with the helper
% method rungeKuttaSystemMethod. The system is solved from t=0 to t=Tf,
% with initial conditions ICs given as a 5 component vector.
```

Each of these last three methods use the Runge Kutta method to numerical solve a disease modeling problem with their respective equations. The shared arguments are the initial conditions, which are a three or five component vector depending on the problem type, and the final time T_f. The step-size used is one day, and the method will return the list of time-stepped values for each population (every day) from $t = 0$ to $t = T_f$. For SIR, the equations are

$$\frac{dS}{dt} = -rSI \qquad \frac{dI}{dt} = rSI - cI \qquad \frac{dR}{dt} = cI.$$

For SIRQ, the equations are

$$\begin{aligned}
\frac{dS}{dt} &= -\alpha SI \\
\frac{dI}{dt} &= \alpha SI - \beta I - \gamma I - \delta I \\
\frac{dQ}{dt} &= \beta I - \eta Q - \rho Q \\
\frac{dR}{dt} &= \gamma I + \eta Q \\
\frac{dD}{dt} &= \delta I + \rho Q
\end{aligned}$$

and for SIRQV, it is

$$\begin{aligned}
\frac{dS}{dt} &= -\alpha SI - \zeta S \\
\frac{dI}{dt} &= \alpha SI - \beta I - \gamma I - \delta I \\
\frac{dQ}{dt} &= \beta I - \eta Q - \rho Q \\
\frac{dR}{dt} &= \gamma I + \eta Q + \zeta S \\
\frac{dD}{dt} &= \delta I + \rho Q
\end{aligned}$$

An example implementation is

```
[S,I,R] = SIRModel_244(0.1, 0.2, [0.99; 0.01; 0], 400);
[S,I,Q,R,D] = SIRQModel_244(0.15, 0.08, 0.02, 0.03, 0.01, 0.04, [0.95; 0.05;
↪  0; 0; 0], 400);
[S,I,Q,R,D] = SIRQVModel_244(0.15, 0.08, 0.02, 0.03, 0.01, 0.04,0.2, [0.95;
↪  0.05; 0; 0; 0], 400);
```