Math 128A, Fall 2014
Programming Assignment 4
due Wed, Dec 3

Our goal is to solve a stiff, nonlinear system of differential equations using a 4th order diagonally implicit Runge-Kutta method. The first step is to write a multi-dimensional Newton solver. The generalization from the scalar case is quite simple:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \text{becomes} \qquad x_{n+1} = x_n - J(x_n)^{-1} f(x_n), \quad J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

Here is a simple implementation that will be good enough for our purposes:

```
function p = newton(f,J,p0,tol)
for i=1:50
  p = p0 - J(p0)\f(p0);
  if norm(p-p0)<tol, break; end
  p0=p;
end
```

The backslash operator `A\b` gives the solution x of $Ax = b$. The function `norm` gives the 2-norm of a vector.

Part 1. We start by testing the Newton solver on the 4 dimensional nonlinear system of equations $x_i = \exp\left(\cos\left(i \sum_{j=1}^4 x_j\right)\right)$, $i = 1, 2, 3, 4$. First we write this system in the form $f(x) = 0$ by defining

```
f = @(x) x - exp(cos([1:4]'*sum(x))); # x is a 4-component column vector
```

Then we define the Jacobian of f by creating a file called `JJ.m` with the following lines of code:

```
function J = JJ(x)
J = eye(4); % 4x4 identity matrix
u = sum(x);
for i=1:4
  for j=1:4
    J(i,j) = J(i,j) + ???; % replace ??? appropriately
  end
end
```

Finally, we solve the system of equations using Newton's method:

```
p1 = newton(f,@JJ,[2.5;2;1.4;.9],1.0e-12);
```

The starting guess was found by plotting $g(u) = u - \sum_{i=1}^4 \exp(\cos(iu))$, where $u = \sum_{i=1}^4 x_i$, observing that $u = 6.7$ is an approximate root of g, defining $x_i = \exp(\cos(6.7i))$, and rounding to two digits, which gave $x = [2.5; 2.0; 1.4; 0.9]$.

Part 2. Next we consider the following stiff system of ODE's:

$$\begin{aligned} y_1' &= -0.04y_1 + y_2y_3, \\ y_2' &= 400y_1 - 10000y_2y_3 - 3000y_2^2, \\ y_3' &= 0.3y_2^2, \end{aligned} \qquad y(0) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad 0 \le t \le 3.$$

We will solve this using both the Runge-Kutta-Fehlberg method and a 4th order L-stable singly diagonally implicit Runge-Kutta method. The code could be run as follows:

```
f = @(y) [ -.04*y(1) + y(2)*y(3); 400*y(1) - 1e4*y(2)*y(3) - 3000*y(2)^2; .3*y(2)^2 ];
J = @(y) [ ?, ?, ?; ?, ?, ?; ?, ?, ?]; % you fill in the question marks

[t2,y2] = rkf( f, 0, 3, [1;0;0], 0.1, 1e-7, 0.25, 1.0e-3);
plot(t2,y2,'.-');

[t3,y3] = sdirk(f, J, 0, 3, [1;0;0], 0.1, 1e-7, 0.25, 1.0e-3);
plot(t3,y3,'.-');
```

The file `rkf.m` has been provided for you on bCourses. Make a duplicate copy of this file called `sdirk.m`. Change the first line to

```
  function [t,y] = sdirk(f,J,a,b,y0,h,hmin,hmax,tol)
```

Change the definition of `A,bb,b1` to hold the following data:

$$A = \begin{pmatrix} 1/4 & & & & \\ 1/2 & 1/4 & & & \\ 17/50 & -1/25 & 1/4 & & \\ 371/1360 & -137/2720 & 15/544 & 1/4 & \\ 25/24 & -49/48 & 125/16 & -85/12 & 1/4 \end{pmatrix}, \quad b = \begin{pmatrix} 25/24 \\ -49/48 \\ 125/16 \\ -85/12 \\ 1/4 \end{pmatrix}, \quad \tilde{b} = \begin{pmatrix} 59/48 \\ -17/96 \\ 225/32 \\ -85/12 \\ 0 \end{pmatrix}.$$

Make sure `bb` and `b1` are stored as column vectors. (Don't delete the ' in `bb = [...]';`) These coefficients define the Runge-Kutta scheme via

$$k_i = f\left(w_n + h\sum_{j=1}^{i} a_{ij}k_j\right), \qquad i = 1, \dots, s$$

$$w_{n+1} = w_n + h\sum_{j=1}^{s} b_jk_j, \qquad \tilde{w}_{n+1} = w_n + h\sum_{j=1}^{s} \tilde{b}_jk_j,$$

where $s = 5$ in this case and $s = 6$ in the RKF scheme. As discussed in section 5.5 of our book, $\tilde{w}_{n+1}$ is an alternative estimate of the solution which is used together with w_{n+1} to choose the next step size. The main difference between RKF and this

SDIRK scheme is that the diagonal entries of A are not zero in the SDIRK scheme. This means that the k_i equations are of the form

$$k_i = f\left(z + ha_{ii}k_i\right),$$

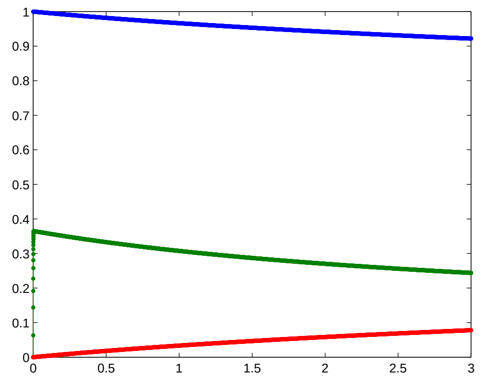
where $z = w_n + h\sum_{j=1}^{i-1} a_{ij}k_j$ is known and $a_{ii} = 1/4$ for this scheme. Thus, if we define $g(k) = k - f(z + ha_{ii}k)$, we can use Newton's method to solve for k. As a starting guess, $f(w_n)$ works well. So, replace the following line in `rkf.m`

```
kk(:,i) = f(z);
```

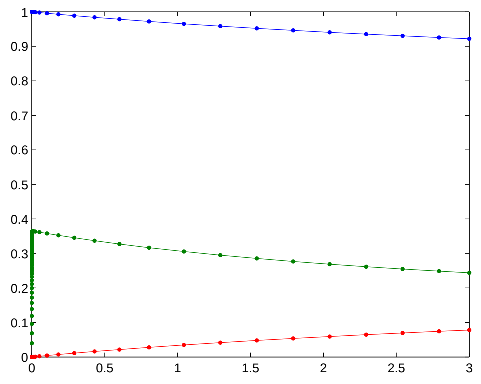
with

```
ah = h*A(i,i);
g = @(k) ???;  % you figure out what goes here (involves f)
dg = @(k) ???; % and here.. (involves J)
kk(:,i) = newton(g,dg,f(y0),1.0e-12);
```

in `sdirk.m`. The stepsize control works the same for SDIRK as it does for RKF, so the rest of the code should not be modified. Plots of the results are as follows:



RKF



SDIRK

The RKF scheme requires 2185 steps while the SDIRK scheme requires only 84 steps. Thus, the extra cost of solving the equations implicitly pays off through a drastic reduction in the number of timesteps required to achieve the same accuracy.

Turn in your result `p1` from part 1 (report each number with 14-16 digits of precision), your version of the above plots, copies of the matlab files you used to generate the results, and a brief report explaining what you did and showing your calculations of `J`, `g`, and `dg`.