HOMEWORK 5

Due: Tuesday, May 8, 2018, 11:55pm

1. **The Split-and-Average Problem.** In class, we discussed the split-and-average approach to subdivide and smooth the surfaces we encounter in many engineering disciplines. In this problem, you will write two separate functions to experiment with this algorithm.

   (a) Write a function that performs the splitting process. It *must use exactly* the function header shown below, where the function name and number of inputs and outputs must be followed exactly.

   ```
   function [ xs ] = splitPts( x )
   % ...
   ```

   As discussed in class, the output of `splitPts` is a vector with twice the number of elements as x, where midpoints are inserted every-other element with a value equal to the average of the neighboring elements. For example:

   $$\begin{aligned} \mathrm{x} &= [1,\ 5,\ 8] \\ \mathrm{xs} &= [1,\ 3,\ 5,\ 6.5,\ 8,\ 4.5] \end{aligned}$$

   Note that the midpoint inserted at `xs(6)` requires data from its left neighbor, `xs(5)`, and right neighbor, `xs(1)`, assuming the vector "wraps around" on itself.

   (b) Write a function that performs the averaging process. It *must use exactly* the function header shown below, where the function name and number and ordering of inputs and outputs must be followed exactly.

   ```
   function [ xa ] = averagePts( xs, w )
   % ...
   ```

   As shown in the lecture slides, the elements of `xa` are formed by calculating the weighted average of the three nearest neighbors, including the current element itself.

   $$\mathrm{xa}(k) = w_1 * \mathrm{xs}(k-1) + w_2 * \mathrm{xs}(k) + w_3 * \mathrm{xs}(k+1) \tag{1}$$

   Where w_1, w_2, and w_3 are the elements of the `[1x3]` input vector w after normalization, i.e., w_1 = `w(1)/sum(w)`. As in the splitting function, we'll assume that the vector "wraps around" on itself, such that `xs(1)` and `xs(N)` are adjacent. An error should be produced from within your `averagePts` function if the sum of the three weights is equal to zero. Negative values are acceptable (and can produce some really interesting fractal results) but the sum of all three cannot be zero.

   (c) Use your `splitPts` and `averagePts` functions to repeatedly split and average a set of x points and y points of your own choosing. Plot the initial distribution of points as unconnected nodes, e.g., `plot(x, y, '.')`. Repeat the process of splitting and averaging until the maximum node displacement after a single split-and-average iteration is less than 1×10^{-3} (i.e.,

$\texttt{max}(\sqrt{dx^2 + dy^2}) < 1 \times 10^{-3}$, where `dx = xa - xs` and `dy = ya - ys`) and plot the final distribution of points on the same axes. You may find it time saving to set a low maximum iteration count while debugging and start with a simple verification test, such as:

```
x = [0, 0, 1, 1];
y = [0, 1, 1, 0];
```

which initializes a square with length 1, before proceeding to more complicated cases.

(d) Using a few of your examples for support, what can you say about the effects of the weight values on the behavior of the final converged shape? Do all weight values lead to convergence? For those that do, how many iterations does it typically take before your method converges (a rough number is acceptable here)?

2. **Runge-Kutta Radioactivity.** Most of us are familiar with carbon-14, the naturally-occurring, radioactive isotope of carbon used in radiocarbon dating, but fewer are familiar with its less-useful cousin, carbon-15. In contrast to the relatively long-lasting carbon-14, which has a half-life of 5,730 years, carbon-15 has a half-life of only 2.45 *seconds*. The amount of carbon-15 over time is given by the following decay equation

$$\frac{dy}{dt} = -\frac{\ln(2)}{t_{1/2}} y \tag{2}$$

where y is the remaining amount of carbon-15 and $t_{1/2}$ is the half-life in seconds. At time $t = 0$, the initial amount of carbon-15, y_0, is 1. The exact solution to this differential equation is

$$y(t) = y_0 \exp\left(-\frac{\ln(2)}{t_{1/2}} t\right) \tag{3}$$

We'll use the exact solution to evaluate and compare the accuracy of three different numerical methods to capture this rapid decay process.

RK1	$c_1 = \Delta t f(t_k, y_k)$ $y_{k+1} = y_k + c_1$
RK2	$c_1 = \Delta t f(t_k, y_k)$ $c_2 = \Delta t f(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_1)$ $y_{k+1} = y_k + c_2$
RK4	$c_1 = \Delta t f(t_k, y_k)$ $c_2 = \Delta t f(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_1)$ $c_3 = \Delta t f(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_2)$ $c_4 = \Delta t f(t_k + \Delta t, y_k + c_3)$ $y_{k+1} = y_k + \frac{1}{6}c_1 + \frac{1}{3}c_2 + \frac{1}{3}c_3 + \frac{1}{6}c_4$

As discussed in class, we'll compare and contrast first-order (RK1, aka explicit Euler), second-order (RK2), and fourth-order (RK4) Runge-Kutta methods, which generally rely on breaking a single timestep down into multiple, incremental calculations in order to reduce overall error.

(a) Write a single function containing all three methods to advance the discretized solution by one timestep. The function must use *exactly* the function header shown below, where the function name and number and order of inputs and outputs must be followed exactly.

```
function [ y ] = advanceRK( y,  dt, method)
% ...
```

where `dt` is the timestep size and `method` is either a 1, 2, or 4 to choose whether to calculate the value at y_{k+1} using the first, second, or fourth-order approximation, respectively. The current amount of carbon-15, `y`, is used to calculate the value at the next timestep, `y = y + ...`, and the updated value is returned as the single output.

Note: Although all three methods are included in a single function, only the result generated by the `method` option will be calculated with a single function call. $t_{1/2}$ is a constant in this problem, not to be confused with the timestep size, Δt, or time, t.

(b) Write a program that repeatedly calls your `advanceRK` function to solve Equation 2 up to $t_{final} = 15$ seconds using each of the three different numerical methods (you'll need to call your function 3 times, specifying a different method each time) using three different timestep values, $\Delta t = [1, 0.1, 0.01]$. Generate 3 plots, one for each value of Δt, containing four curves each (the three numerical solutions plus the actual solution). Additionally, print the average error associated with each method using exactly the format shown below (relax, values shown are for format-illustration purposes only):

```
   dt         RK1         RK2         RK4
1.00:   3.14e-02   3.68e-03   1.64e-05
0.10:   3.24e-03   3.26e-05   1.44e-09
0.01:   3.24e-04   3.21e-07   1.42e-13
```

The values in the table are the mean values of the absolute errors between the numerical approximation and the exact solution given by 3. For example, to calculate the average RK4 error for a given Δt, calculate the vector of differences, `RK4 - actual`, ignore negative signs, and calculate their average. Since time, t, doesn't explicitly show up on the right-hand side of Equation 2, we won't be plugging in any $t_k + \ldots$ terms. The time terms are presented in the Runge-Kutta steps since the value of y is *inextricably* linked with time and the time values serve as a nice indication of "where" the evaluation points occur.

Note: You can open a new plotting window with `figure` to ensure that plots 2 and 3 don't simply overwrite one another.

(c) Using your most accurate numerical method, what percentage of the original amount of carbon-15 remains after just 15 seconds? Just by examining the scaling of the average errors, explain how you can verify that the three different methods incur error that scales like $\mathcal{O}(dt)$, $\mathcal{O}(dt^2)$, and $\mathcal{O}(dt^4)$; that is, first, second, and fourth-order, respectively? What happens to all three solutions (particularly RK1/Euler) when we take a *very* large timestep, e.g., $\Delta t = 5$? Why does this occur?

Using the naming convention presented in the syllabus, submit **two** separate files to the CCLE course website: (1) a .pdf of your written report and (2) a .zip file containing all of the MATLAB files written for the assignment (from this point forward, this will include the "main" scripts along with any functions written). Remember to use good coding practices by keeping your code organized, choosing suitable variable names, and commenting where applicable. Any of your MATLAB .m files should contain a few comment lines at the top to provide the name of the script, a brief description of the function of the script, and your name and UID.