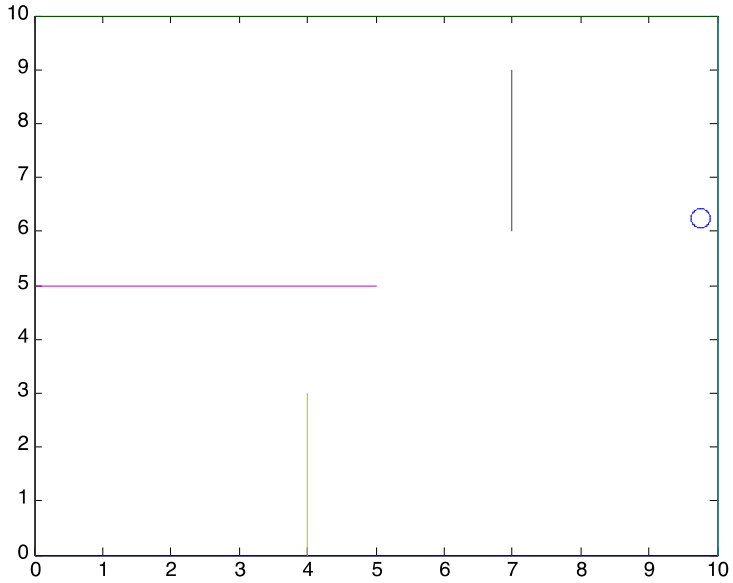
Introduction to Scientific Computing, Homework #4
Due by 12 pm on Thursday, May 26, 2022

Problem 1 (20 pts): Billiards
In this problem you will develop a simulation of a billiard ball as it moves and collides with a set of walls as shown below.



In this simulation the state of the ball is represented by a four-element row vector which encodes its position and velocity `[x y vx vy]`. The locations of the walls are stored in an array with one row for every wall each row encodes the coordinates of the endpoints of the wall segment in the following format `[x1 y1 x2 y2]`. So the array `[1 3 1 5; 7 5 10 5]` would represent two walls one with end points (1,3) and (1,5) and the other with endpoints (7,5) and (10,5). Note that the walls will either be purely horizontal or purely vertical you can tell which by checking whether the x or y coordinates of the endpoints are constant. In this simulation whenever the ball collides with a wall its speed after the collision is related to the speed before the collision by the coefficient of restitution. For example a coefficient of restitution of 0.9 would imply that the speed after the collision would be 90% of what it was before the collision.

You are provided with a script called `billiardScript.m` which drives the simulation. Your job is to write two functions that will complete the simulation. Feel free to play around and create new scripts, adding walls, changing the initial conditions, the coefficient of restitution etc.

- `function [t, collisionState] = findCollision (ballState, wall, coefficient_of_restitution)`

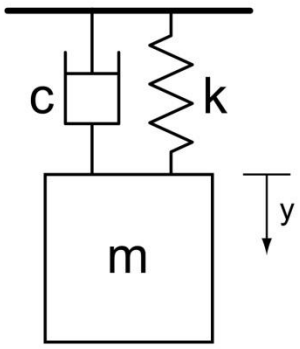
  This function will determine when the ball will collide with a given wall. The `ballState` input is a four-element vector denoting the current position and velocity of the ball. The `wall` input is a four-element row vector storing the coordinates of the wall endpoints in the following format `[x1 y1 x2 y2]`. The `t` output is used to return the time at which the collision will occur, and the `collisionState` output is used to return the position and velocity of the ball after the collision. If the ball does not collide with the wall the `t` output should be `Inf` and the `collisionState`

output should be an empty array. (Hint: you will probably find it easiest to handle horizontal and vertical walls as separate cases. Start with pencil and paper and work out expressions to compute when a ball at position x, y moving with velocity vx, vy would impact the given wall, if at all.)

- `function newBallState = updateBallState (ballState, dt, walls, coefficient_of_restitution)`

  This function computes where the ball will be after `dt` seconds taking into account collisions with the walls. (Hint: you should use the `findCollision` function here and determine which wall, if any, the ball will collide with **first** during the given simulation interval. You will probably want to write this as a recursive function which calls itself to simulate the balls motion for the time remaining in the time step after the first collision. For inspiration, you should definitely look at the code that was used to run the baseball simulation discussed in class which also involved simulating collisions between the ball and a ground plane.)

Problem 2 [20 pts]: Damped harmonic oscillators described by second order ordinary differential equations are often observed in engineering problems. You might encounter such systems when studying diffusion, Newtonian particle physics/atomistics, and the dynamics of many mechanical systems (*e.g.* shock absorbers on cars or robotics). Consider the example of a block of weight *m*, subject to gravity, and connected to a parallel spring and damper.



This system can be described by the following second order differential equation

$$m\ddot{y} + c\dot{y} + ky = 0$$

where *c* is the damping coefficient (units: kg/s), *k* is the spring constant (units: kg/s^2 = N/m), and the derivatives are with respect to time.

It is often useful to know the position and velocity of such a system as a function of time. You are to create a function that solves this differential equation using `ode45` – a differential equation solver built into MATLAB. You will look observe the position and velocity of this block subject to a set of initial conditions using the following function declaration.

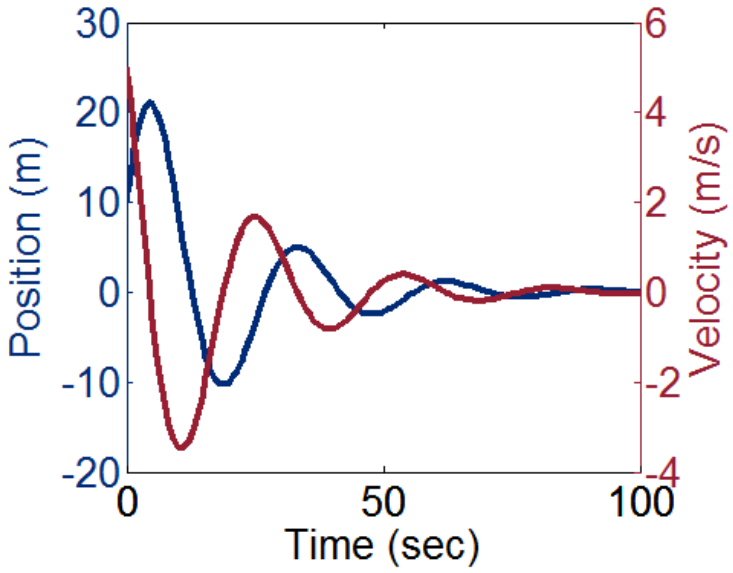
```
function msd_posVel(c,k,m,yi,vi,t_beg,t_end)
```

where `yi` is the initial position, `vi` is the initial velocity, `t_beg` is the start time of the simulation (most often taken to be 0), and `t_end` is the end time of the simulation.

The function that stores the differential equations and is called by ode45 should be of the form

```
function dy = msd(t,y,c,k,m)
```

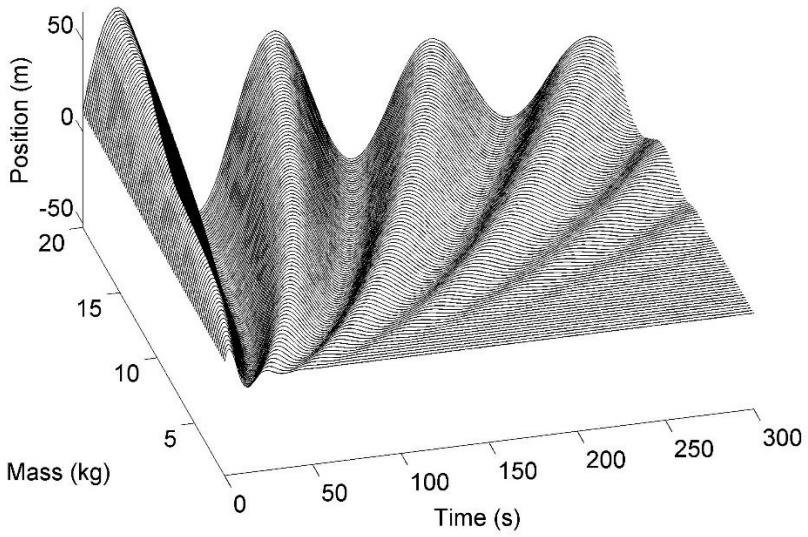
Your function `msd_posVel` should produce a plot with two y axes using `plotyy` with the position of the block on the left axis and velocity of the block on the right axis. This plot should be pleasing and include labels on the x axis and both y axes. Include a plot for 0 ≤ `t` ≤ 100 s, `m` = 2 kg, `c` = 0.2 kg/s, `k` = 0.1 N/m, `yi` = 10 m, and `vi` = 5 m/s with your .docx submission. See the following for an example of the solution plot.



You will then visually investigate the effect of varying mass on the system position as a function of time using a function with the following function declaration.

```
function msd_water(c,k,m,yi,vi,t_beg,t_end)
```

where input *m* is a vector of masses to be modeled (0≤*m*≤20 kg by divisions of 0.1 kg). All other variables are scalars with the same magnitude of those used to produce the plot for `msd_posVel`. Your plot should look similar to that below. Note that plot3() is a more suitable choice than surf() for this plot.



Problem 3 (20 pts): Predator-Prey Model

The Volterra-Lotka equation provides a model that describes the evolution of the population of two species, one predator, one prey.

$$\frac{dx}{dt} = x \times (a - b \times y)$$

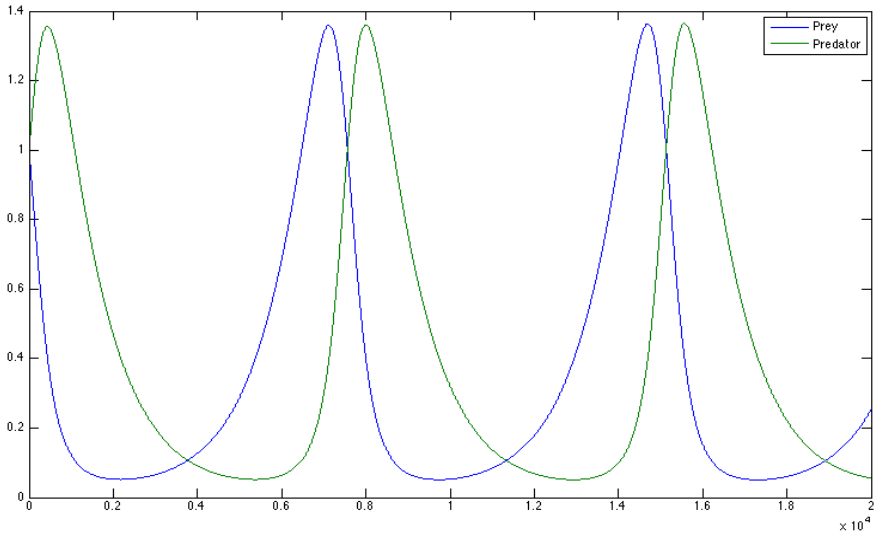
$$\frac{dy}{dt} = -y \times (g - d \times x)$$

where *x* and *y* are populations for the prey and the predator, respectively, α is the growth rate of prey, β is the rate at which predators destroy prey, γ is the death rate of predators, and δ is the rate at which predators increase by consuming prey. Your job is to write a function with the following signature that will simulate this dynamical system.

```
function state = lotkaVolterra (initial_state, alpha, beta, gamma,
delta, dt, ntimesteps)
```

This function returns an array with 2 rows and n timesteps columns where the first row corresponds to the prey population (for example, rabbits), the second row to the predator population (for example, wolves), and the columns correspond to the time steps as in the examples worked in class. The initial state is a two vector giving the initial state of prey and predator populations `[x0,y0]`, the parameters `alpha`, `beta`, `delta` and `gamma` are constants representing the interaction of the two species, `dt` denotes the duration of each simulation time step and `ntimesteps` denotes the total number of time steps.

To test the function you should use the following parameters
`Initial_state = [1,1], alpha=1, beta=2.5, gamma=1, delta=2.5, dt =0.001, ntimesteps=20000`. Your output should look like this.



For fun, you can experiment with different parameter values to see how the system evolves.

Problem 4 (20 pts) Grassfire/PacMan

In this assignment you will write code to implement the Grassfire/PacMan algorithm to find the shortest path through a grid. You will write a function that takes three inputs, the first input is a 2D logical array, `occupancy`, whose entries correspond to cells in a grid, `true` values correspond to empty traversable cells and `false` values correspond to occupied cells that one can't pass through. The remaining two parameters, `dest_row` and `dest_col` specify the row and column of the destination cell in the grid. Your function should have the following signature and should return as output a 2D array, `distance`, of the same size as the `occupancy` array, where its values indicate how many steps each cell in the grid is from the destination cell.

```
function distance = grassfire (occupancy, dest_row, dest_col)
```

The grassfire algorithm is outlined below in the pseudocode:

- Create a `distance` array of the same size as the `occupancy` array;
- Initialize all of the entries in the `distance` array to infinity;
- Set the distance corresponding to the destination cell to zero;
- Loop
  - For each cell
    - Find the neighbor of this cell with the smallest distance value
    - Set the current distance value equal to this smallest distance + 1
    - (Remember that occupied cells in the grid will always have a distance of infinity – ie their distance values never get updated)
  - Break out of the loop when you go through a pass where none of the distance values change – remember that you can use the `break` function in MATLAB to exit a loop
- End loop

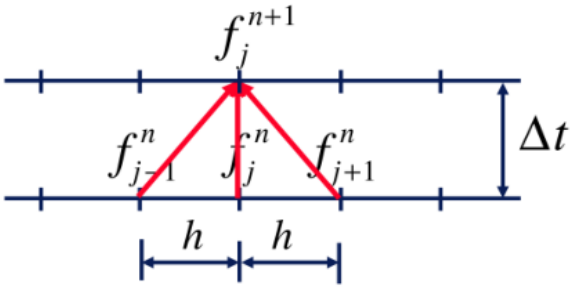
Hint: You may find it useful to initially create a distance array that has 2 more entries in either dimension than the occupancy array. This can be seen as morally equivalent to padding the occupancy grid with a border of occupied cells and will allow you to avoid some tedious checking for cells on the border of the occupancy array. Remember to remove your padding when you return the final distance result.

Problem 5 (20 pts) The Advection-Diffusion Equation

You need to write a program to solve the one-dimensional convection-diffusion equation. Let's first take a brief look at the convection-diffusion equation.

$$\frac{\partial f}{\partial t} + U\frac{\partial f}{\partial x} = D\frac{\partial^2 f}{\partial x^2}$$

Although this equation is much simpler than the full Navier Stokes equations, it has both an advection term and a diffusion term. The first term on the left is the time derivative, and the second term is the convection term. The right-hand term represents the diffusion term. You don't need to know what they mean, you just need to know how they evolved. The picture below will give you an idea of how it evolved.



$f_{j-1}^n, f_j^n, f_{j+1}^n$ represent three adjacent points in a one-dimensional region at time t. f_j^{n+1} represents the value of the point j in the one-dimensional region at time t + △t. As a result, the value of every point at level n+1 is given explicitly in terms of the values at the level n. The numerical solution of the one-dimensional convection-diffusion equation is as follows:

$$f_j^{n+1} = f_j^n - \frac{U\Delta t}{2h}\left(f_{j+1}^n - f_{j-1}^n\right) + \frac{D\Delta t}{h^2}\left(f_{j+1}^n - 2f_j^n + f_{j-1}^n\right)$$

In addition, you need to know the exact solution to it. For initial conditions of the form:

$$f(x, t=0) = \mathrm{A}\sin(kx)$$

It can be verified by direct substitution that the exact solution is given by:

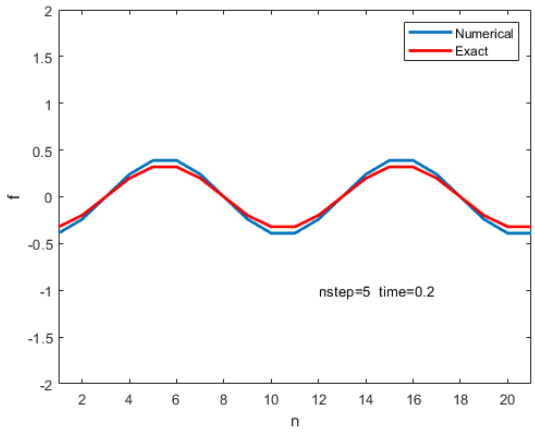
$$f(x,t) = e^{-Dk^2 t}\mathrm{A}\sin(k(x-Ut))$$

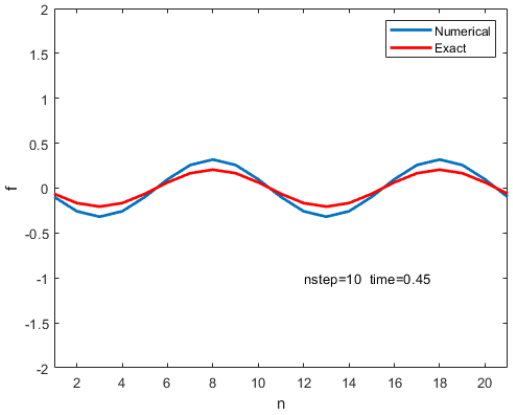
which is a decaying travelling wave.

The given parameters are as follows, A=0.5, k=2π, D=0.05, U=1, the length of the one-dimensional region is 2, and you have to divide n = 21 points. That means h = length/(n-1). And dt=0.05, you need to calculate the exact and numerical solutions for the first 10 steps. The first step is the initial step. You should generate 10 plots, and plots of step 5 and step 10 are shown below.

Periodic boundary conditions are used for boundary conditions. That means $f_1 = f_n$.

Pay attention to the evolution of boundary points!!!





Note that you should use `hold off` and `pause` to update each step instead of generating ten images separately.