ENGN 2020:

Midterm #2

Brown University School of Engineering

Assigned: April 8, 2019, Due: April 12 (4pm), 2019

Instructions: You may not discuss this examination with any person or instructor, inside or outside this class. This includes all forms of communication, such as email, chat, etc.

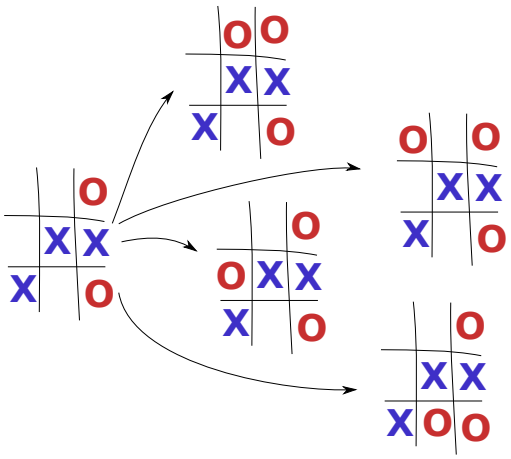
Turn in your assignment either as a single PDF (preferred) to Canvas or a single stapled document by 4pm on the due date. Your assignment should be clearly formatted, with all Figures and Tables logically numbered. Please turn in your well-documented code as an appendix to your solution, as appropriate, for consideration in partial credit. You will also be separately uploading your code for Problem #1, as noted in the problem statement.

This examination is worth 50 points in total.

Problem 1

[15 points]

Your assignment is to write an algorithm that uses the principles of graph theory to play against a human in tic-tac-toe. The rules of tic-tac-toe are simple: two players alternate putting their respective marks (X's and O's) on a 3×3 board; the first player to have three marks in a row (vertical, horizontal, or diagonal) wins. An example set of possible moves by O starting from a particular state is shown below.



Represent the current state of the game as a numpy array: 0's for empty spaces, 1's, for X's, and -1's for O's. The state of the left-most board above is:

```
array([[ 0,  0, -1],
       [ 0,  1,  1],
       [ 1,  0, -1]])
```

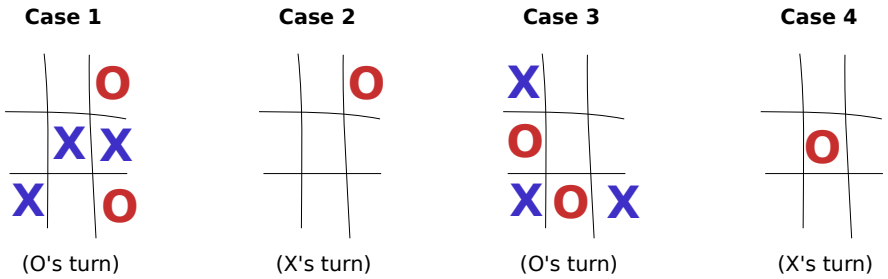
(a) First, create a `Vertex` class—similar to in your homeworks—that has an appropriate adjacency method (*e.g.*, `get_children`).

(b) Next, add a method to `Vertex` that tells you the current status of the game. (*E.g.*, `get_status`.) It should return one of three messages depending on the state of the game: `'X wins'`, `'O wins'`, `'tie'`, or `'in progress'`.

(c) Now, write a function that determines the computer's next move. It should take in a vertex (that is, an instance of the `Vertex` class) and return a new vertex that contains the state of the game after the computer's move. Call this function `computer_move`.

You should choose the next move based purely on the brute-force (frequentist) probability. That is, for each possible (next) move, find all possible outcomes of the game. Define the probability as the number of outcomes in which the computer wins divided by the total number of possible outcomes. Choose the move with the highest probability; if two moves have identical probability, choose the one with the lowest indices.

Use your function to report the next move for each case below. Also report statistics for your proposed move: the number of winning outcomes and number of total outcomes.



Part d. Now write a complete (but text-only) "game" that plays against you, a human. It should start with a blank board, randomly decide who goes first, and alternate moves between the human (input by the keyboard) and the computer (chosen by your algorithm).

Hint: You can take the human player's move (*e.g.*, '0,2') with the `input` command in python, and convert it to a numpy array like

```
move = input('Enter move: ')
move = np.array(move.split(','), dtype=int)
```

Turn in code. You must turn in a working piece of code, as a single text file called `problem1.py`. It should contain all of the following, (and we should be able to execute it without any non-standard libraries):

1. Your `Vertex` class from part a–b.
2. Your function `computer_move`, from part c.
3. Your complete working game from part d, in a function called `game`.

Problem 2

[20 points]

Here, you will look at how *linearizing* systems of ordinary differential equations can provide insights into their behavior.[1] In a particular chemical reactor that converts chemical A to chemical B, shown below as a "black box", the governing equations describing the exit conditions (C_A, T) can be derived to be:

$$\frac{dC_\mathrm{A}}{dt} = \frac{C_{\mathrm{A,inlet}} - C_\mathrm{A}}{\tau} - k\,C_\mathrm{A} \equiv f_1 \tag{1}$$

$$\frac{dT}{dt} = \frac{T_\mathrm{inlet} - T}{\tau} - \frac{H_\mathrm{rxn}}{\rho\,c_p} k\,C_\mathrm{A} \equiv f_2 \tag{2}$$

where $k = A\,e^{-E_\mathrm{A}/(RT)}$. C_A is the concentration of A (mol/L) and T (K) is the temperature at the reactor exit. (Ignore the concentration of B, as it can be found at any time just by monitoring how much A converted.) The parameters have the values:

Parameter	Units	Value	Description
A	s^{-1}	4.3×10^{18}	rate "pre-factor"
E_A	J/mol	125,500	activation energy
H_rxn	J/mol	-83,700	enthalpy of reaction (heat released)
$C_{\mathrm{A,inlet}}$	mol/L	2	concentration of A in the reactor feed
$\rho\,c_p$	J/L/K	4184	(density) $\times$ (heat capacity)
τ	s	60	residence time of reacting fluid in reactor
R	J/mol/K	8.314	ideal gas constant
T_inlet	K	300	temperature of the reactor feed

(a) At steady state ($t \to \infty$), the outlet conditions of the reactor will be constant. This means dC_A/dt and dT/dt are both zero. Use a non-linear algebraic solver with equations (1) and (2) to find the concentration (C_A) and temperature (T) of the outlet at steady state.

Hint: Multiple solutions are possible! You may want to factor out roots to ensure you have found all possible values.

(b) You should have found three unique solutions in the previous part. Assume that in operating this reactor, you have control over T_inlet. Perform a bifurcation analysis to determine the range of T_inlet where you will have multiple steady-state solutions; that is, find the values T^*_inlet where the number of steady-state solutions changes.

[1] Specifically, this problem will beat into your head how useful the Jacobian is.

(c) For this part, use $T_{\text{inlet}} = 300$ K, where three steady state solutions occur. Here, you will analyze the *stability* of each of the steady-state solutions.

As background on stability, consider performing the following experiment: the reactor is running at a steady state; that is, the outlet is at one of the concentration/temperature pairs you found in part a. Now *perturb* the system—for example, push the temperature just 1/10 of a degree higher—and observe what happens as the system evolves; that is, track $C_{\text{A}}(t)$ and $T(t)$. We can imagine two scenarios: (1) the temperature and concentration return to their previous values, or (2) the temperature and concentration drift off to some other steady state solution. If we observe the first behavior, we call the steady state *stable*; if we observe the second, we call it *unstable*. If you like, you can read more about stability in Beers, Chapter 4, beginning on page 169, but we'll briefly summarize here.

The Jacobian matrix, evaluated at a steady-state value, (*i.e.*, $\underline{\underline{\mathbf{J}}}\big|_{\underline{\mathbf{y}}_{\text{SS}}}$), contains the linearized version of the original equations about the steady state. Therefore, if we find the set of eigenvalues $\{\lambda\}$ of the Jacobian, we understand the system's behavior for small perturbations from the steady state (since the solution to the linearized form is $\sum_i \underline{\mathbf{v}}_i e^{\lambda_i t}$, where $\underline{\mathbf{v}}_i$ is the eigenvector associated with λ_i). This means we only need to look at $\{\lambda\}$, and can determine the stability of each steady state by:

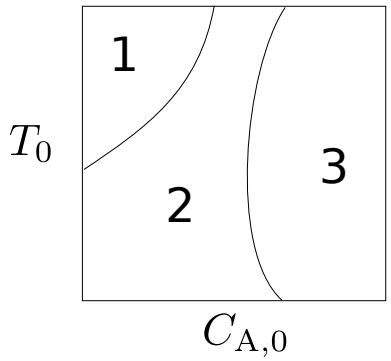
- All $\text{Re}(\lambda) < 0$: stable
- Any $\text{Re}(\lambda) > 0$: unstable
- Any $\text{Im}(\lambda) \neq 0$: oscillates

where $\text{Re}(\lambda)$ and $\text{Im}(\lambda)$ refer to the real and imaginary part of the eigenvalue, respectively. Use this approach to predict the stability of each steady state from part (a).

(d) Beers (Ch 4, page 180) provides a simple criterion to determine if your system of differential equations is *stiff*. Is this a stiff system? (Justify your answer.)

(e) Now, use a numerical integrator, such as `scipy.integrate.odeint` to make plots of C_A and T versus t. Do this for various initial conditions (that is, t=0 values of the exit conditions C_A and T); you should choose a large enough variety of initial conditions that we see solutions that approach or diverge from each of your steady state conditions above.

(f) Finally, create a "map" that could be used to determine from a given initial condition which steady state will be reached. *E.g.*, your map might look something like:



You should make your map by iterating over a grid of initial values of C_A and T; at each point on your grid you would perform an integration to see which steady state you approach.

Note/hint: Many portions of this problem are independent of the other portions; if you get stuck on a part, you do not necessarily need to skip the following parts!

Problem 3

[15 points]

Here, you will examine optimizing a noisy function. Consider the function that you used in a previous homework, but which now contains some noise ξ:

$$f(x_1, x_2) = 0.045x_1^4 - x_1^2 + 0.5x_1 + 0.065x_2^4 - x_2^2 + 0.5x_2 + 0.3x_1x_2 + \xi$$

where ξ is a random number drawn[2] from a normal distribution with mean 0 (zero) and standard deviation σ:

$$\xi \sim \mathcal{N}(0, \sigma^2)$$

Part a. Create contour plots of f at several values of σ. Make sure that you choose enough values of σ so that you can understand roughly where the noise starts to be significant, as well as when the noise washes out the underlying function.

Part b. Use a Nelder–Mead algorithm that you write, such as the one you wrote in Homework #4, to try to find the optimum of this "noisy" function. Start off from a simplex of [(0, 0), (-1, 0), (0, 1)], which should always converge to the global minimum when noise is absent.

Report on the effect of the noise on your algorithm's success. Specifically, for at least five different values of σ, report:

- the number of times out of 1000 runs that it converges to the *same* local minimum as the noise-free case, and
- the number of times out of 1000 runs that it converges to *any* of the true (noise-free) local minima.

Note that you will need to determine how you assess that you have found a local minimum in the presence of noise! (You should *not* use your pre-existing knowledge of the noise-free minima; your approach should be general to the case that you do not know the minima in advance.) *Concisely describe your procedure and criteria for determining this, along with 1–2 example plots.*

Part c. Repeat this with scipy's built-in Nelder–Mead algorithm and scipy's built-in BFGS algorithm, both available in `scipy.optimize.minimize`. Briefly explain any differences in performance from your home-made function.

[2] That is, each time the function is called a new value ξ is drawn from the same distribution.