Spark 10 Project

Choose one of the following 10 options for your project, or you may come up with your own project, but you must email the instructor your project idea for approval no later than May 1, 2025. These projects span a wide range of difficulty (and are roughly in order of least difficult to most difficult). The more difficult projects will be graded more leniently than the less difficult projects. Note that the first two of the projects do not require programming.

You may work alone (10% extra credit) or in a group of 2-3 other students in the Spark 10 class. If you work in a group, one member of the group should submit the project and the group members should upload the names of the other members of the group.

The final due date for the project is Thursday May 15 at midnight (no extensions are possible past that date). You will get 10% extra credit if you submit the project by Friday May 9 at midnight.

Project Option 1: One sided Turing Test on Modern AI programs.

The Turing Test was proposed by computer pioneer Alan Turing as a way to test if a computer was actually intelligent. It involved a person talking via teletype to a person and to a computer and then trying to determine which was which. In this project, use one of the online AI program either one from the list below or another of your choice, to have a "conversation" of **at least** 25 exchanges (each involving a line from you and a line from the AI) and then write a short (~250 word) description of your conversation, noting which exchanges the AI seemed most human-like and which the AI seemed least human-like. (Note that some require you to create a free account.)

New AI systems:

ChatGPT: https://chat.openai.com/

Gemini: https://gemini.google.com/app

Claude AI: https://claude.ai/

Perplexity: https://www.perplexity.ai/

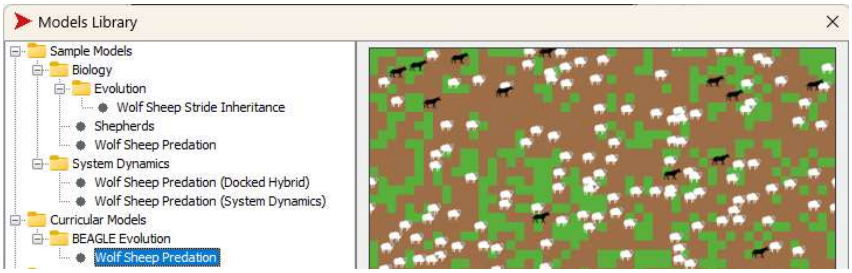
Old AI systems:

Kuki: https://www.kuki.ai/

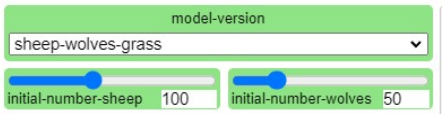
Eliza: http://psych.fullerton.edu/mbirnbaum/psych101/eliza.htm

Project Option 2: Wolf-Sheep-Grass 3-Population Predator-Prey Model in NetLogo

This project involves using the NetLogo model described at (https://ccl.northwestern.edu/netlogo/models/WolfSheepPredation) to study the dynamics of a 3-population predator-prey model including wolves, sheep, and grass. You can run this model in your browser or download the NetLogo app to run on your computer. Be sure to select the version of this model in the "Curricular Models" section of the NetLogo library:



When starting the model, first select the "sheep-wolves-grass" model version.



Then run the model several times, adjusting the starting conditions and model parameters shown here:

Find starting conditions that lead to the following 4 outcomes. For each, include a screenshot of the model that shows the parameters and the graph of the populations of sheep, wolves, and grass. If you cannot find parameters that give one of these specific outcomes, submit your best results. Outcomes to find:

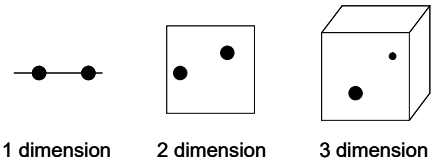
1) Sheep population goes extinct first
2) Wolf population goes extinct first
3) Roughly periodic cycle of peaks and valleys in the wolf and sheep populations lasting at least until time 1000.
4) Nearly "steady state" where the sheep and wolf populations are roughly constant for a time interval of at least 200 time steps.

Project Option 3: More than 2 Random Walkers in the Fog.

In Class 7 we wrote a program to model 2 random walkers moving on a ring and the final slide for the class outlined that would be necessary to extend the model to multiple random walkers. Starting with **`fog.py`** code provided on the Project page, implement this multiple-walker program, including an outer loop of 2-9 walkers.

Project Option 4: Hypercube Line Picking using NumPy:

A long-standing mathematical question is the *Hypercube Line Picking* problem, which asks what is the average distance between two points randomly chosen in a hypercube with *n* dimensions and unit sides (i.e. each side has length=1). For example, consider the hypercubes with 1, 2, and 3 dimensions. The problem involves picking two points in the space, defined by 1, 2, or 3-coordinates for the 1-, 2-, and 3-dimensional cases, (and *n* coordinates for the *n*-dimensional case).



$$\frac{1}{3}\sqrt{n} \le Dist(n) \le \sqrt{\frac{1}{6}n}\sqrt{\frac{1}{3}\left[1+2\sqrt{\left(1-\frac{3}{5n}\right)}\right]}$$

Low n High n

The limiting average interpoint distances for the low-*n* and high-*n* limits are given by the equations shown at left (equations from http://mathworld.wolfram.com/HypercubeLinePicking.html)

For this project you will write a Monte Carlo Python script (named **`hypercube.py`**) to calculate the average distance between random pairs of points in 1- to 30-dimensions and make a plot of this average distance vs the number of dimensions. The **`np.random.random()`** function will generate evenly distributed random numbers between 0 and 1**.**
Hint 1: Have an outer loop over the number of dimensions and an inner loop over number of trials (use at least 1000 trials). Hint 2: For each trial store the distance between the points in an Numpy ndarray that grows by 1 for every new trial, and then calculate the average value in the array using the **`numpy.mean()`** command and store that result in another array that grows by one with each new dimension added. After your script is complete this second vector will contain 30 values (the first for the 1-D case, the second for 2-D, etc. up to 30-D), which you can plot using mathplotlib.

Project Option 5: Programs from Project Euler Challenges

Project Euler is a website (www.projecteuler.net) that contains about 930 "programming challenges" each of which produces a single numerical output that you input to the site to get credit for solving each challenge. Most programs require less than 25 lines of programming and many less than 10. Solutions have been posted in nearly 100 programming languages (Python & C/C++ most common). For this project, sign up for a free Project Euler account and solve **four** of the problems using Python, not including problem 1 that we solved in class.

Project Option 6: Largest of N normally distributed numbers using NumPy

The failure of biological and physical systems often depends not on the average behavior of its components, but on the behavior of one or a few "outliers". For example, a tumor can begin when one cell out of the trillions in the body is transformed into a cancerous cell, and a physical structure can fail when a single critical connector or support fails, even if all the other components are fine.

You can create a simple model of how the risk of a system failure grows with the number of components by generating a set of *N* normally distributed random numbers (that is, a set of *N* numbers that would form a bell curve if made into a histogram), and then look at the value of the largest number in that set of *N* values. If you repeat this calculation many times, you will get an estimate of how far the largest single individual value deviates from the average of a set of *N* values, which is an approximate estimate of how the likelihood of a system failure grows with the number of critical components. If the set includes a lot of numbers, the largest value will be much larger than the average value of the set. For example, the following ten normally distributed numbers have a mean of 0.025 (and a standard deviation of 0.969), but the largest value in the set is 1.783.

(-0.125, -0.247, -0.049, -1.468, 1.416, 0.389, 1.783, -0.124, -0.652, -0.675)

In this project you will write a Python program (which should be named **`maxnorm.py`**) that will investigate how the maximum value in a set of normally distributed random numbers (with mean=0.0 and sd=1.0) changes with the number of values in that set. Specifically, your program will generate sets of *N* normally distributed random numbers and pick the largest from that set. (The Numpy function **`numpy.random.normal(mu, sd, N)`** will produce an array of *N* normally distributed values with mean **`mu`**, standard deviation **`sd`**.) Do this for a range of *N*'s (from 1 to 200) and you will do this 5000 times for each value of *N* and calculate the average largest value for each *N*. Here is the pseudocode for this project.

```
Loop over values of N (1 to 200):
   Initialize sum variable to zero for calculating average maximum value
   Loop over trial (0 to ntrials-1):
         Calculate set of N normally distributed random numbers (NumPy)
         Select largest value (NumPy)
         Add largest value to sum variable
   Divide sum by number of trials to get average maximum value
   Store the average maximum value in an array
Plot the average maximum values vs N using matplotlib
```

Project Option 7: Expanded point class

Expand the point class **`point.py`** posted on the CatCourses Project page to add several new functions and capabilities listed below:

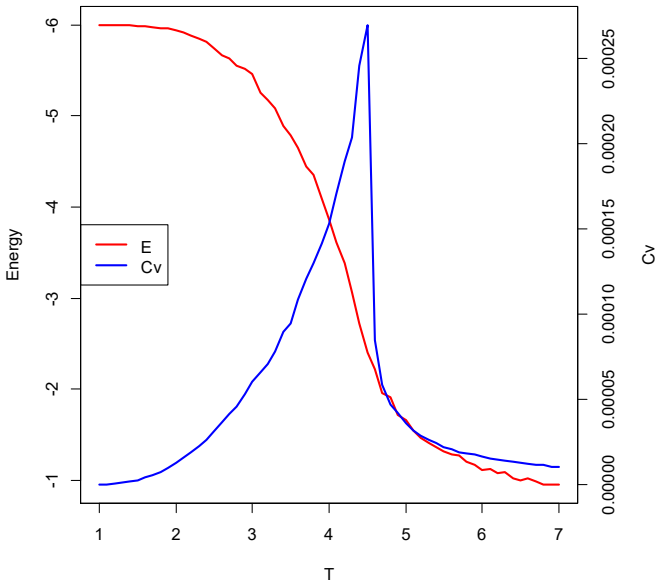
a. **`test(p1,p2)`**: Test that two points have the same dimensionality. If they don't, use the **`sys.exit()`** function to quit with an appropriate message and add this to the functions operating on 2 points.
b. **`cross(p1,p2):`** Return the vector cross product between the vectors defined from the origin to the point
c. **`distance(p1,p2):`** Return the distance between the two points.
d. **`sum(p1,p2):`** Return the vector sum of the two points.
e. **`sub(p1,p2):`** Return the vector resulting from subtracting **`p2`** from **`p1`**

Project Option 8: 3D Ising Model

The Ising model has a different critical temperature in different numbers of dimensions. For this project, you will modify a 2D Ising model program to run in three dimensions (the original 2D version is provided at the CatCourses Project page). In three dimensions the energy of each spin will depend on its six nearest neighbors—the original 4 neighbors, left, right, up, down, as well as front and back. The energy equation will be the same, +1 for each neighbor with an opposite spin and -1 for each neighbor with the same spin, which will give you a minimum energy of -6 per spin. Once completed, you can compare your results at different temperatures to the following graph of average energy and heat capacity vs. temperature for a 20x20x20 3D Ising model.

Hint 1: this will require using the 3D list structure mentioned on Day 2 when we talked about data structures in the workshop.

Hint 2: Be sure to change the energy equation in both the calculation of the total energy and the ΔE.



Project Option 9: Analysis of Hailstone Sequence

In class we went over a short program to calculate the *hailstone* sequence of numbers that follows the following rules for some number n:

- If n is even, divide it by 2
- If n is odd, multiply it by 3 and add 1
- If n equals one, stop

For example, here's the hailstone sequence for the number 6:

6→3→10→5→16→8→4→2→1 [Stop]

The so-called Collatz Conjecture is that a hailstone sequence starting from any *n* will eventually stop (and this has been tested out to *n* greater than 5×10^{18}). For more info about hailstone sequences, see http://en.wikipedia.org/wiki/Collatz_conjecture

Another interesting property of hailstone sequences is how many values are in a sequence from the starting value to 1. The sequence starting at 6 shown above contains 9 values. The sequence starting with n=179 has 871 values. These lengths (also known as "stopping times") form an interesting pattern when you graph them versus the initial n values. There are a few other interesting numbers about each sequence, including the maximum value reached during the sequence (e.g. 16 in the sequence starting with 6) and the frequency at which different integers appear in different sequences.

For this project write a Python program to compute the values in the hailstone sequences starting from each of the integers from 1-1000 (or higher) and produce separate plots of the following values:

a. Stopping time versus the starting value.
b. Maximum value reached during the sequence versus the starting value.
c. A histogram of the number of times each integer value appears in the whole set of hailstone sequences you calculated (i.e. for all starting values 1-1000)

Project Option 10: Monte Carlo Analysis of "One-handed Solitaire"

Stan Ulam, the inventor of Monte Carlo simulation, claims that the idea came to him while he was recuperating from an illness in the 1940s and playing solitaire with a deck of cards. He tried to figure out the probability of winning purely mathematically, but the vast number of possible shuffles of a card deck ($52! \approx 10^{68}$) made the analysis too difficult. Instead, he imagined using one of the early electronic computers (to which he had access as part of the Manhattan Project) to rapidly play the solitaire game multiple times from different random shuffles and then estimate the probability of winning from the observed win/loss record. In this project, you will write a Monte Carlo simulation of a simple solitaire card game. The rules of this game are simple (see play examples below):

a. Deal each card successively
b. If the suit matches the N-3 card, remove middle 2 cards
c. If the value matches the N-3 card, remove all 4 cards
d. Goal: End up with no cards

First use this program to evaluate a regular 4-suit, 13-face, 52-card deck, **but also use the program to determine the win probability for decks with different number of suits or faces** (but note that you can only win this solitaire game if you start with an even number of cards in your deck).