Portfolio of Computer Science Projects

Chad Bloxham
M.S. Computer Science Applicant

1 Overview

The following sections give brief descriptions of the most notable programs I have written while an undergraduate at UCLA, either by assignment or of my own accord. For a more detailed report of a particular program and/or the text of the actual code, email cbloxham@ucla.edu.

2 MatLab Programs

2.1 Path-finding with the A* Algorithm

Initializes a grid of nodes as either closed (cannot be used in a path solution) or non-closed (can be considered when solving for a path from the start node to the finish node). Proceeds by entering a while loop that plots the nodes and path from the start node to the current node. The loop continues by iterating through the neighbors of the current node and updating their f values. This estimates the "cost" of choosing a particular node as the next one in a path. A node's f value is determined by its distance from the finish node as well as the total distance travelled to get to the node. Then the next current node is found by determining which open node (walkable and previously explored) has the smallest f value. If there are no open nodes, the program terminates because the barrier nodes have made a solution impossible. Alternately, the loop terminates if the current node is the finish node, meaning the path solution has been found.

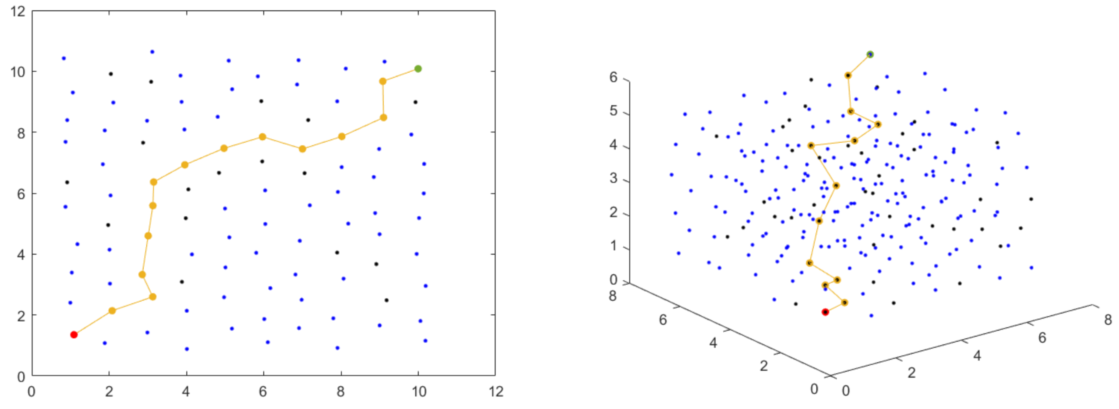


Figure 1: Path solution found on an obstructed (a) 2D grid and (b) 3D grid. The start node is denoted by a red coloring and the finish node is denoted by a green coloring.

We can perform path-finding on an image file by iterating through each pixel's assočiatted red, green, and blue value and placing a node at that pixel if the values lie within a user-specified range. For our image of the UCLA campus map, we choose rgb values such that the grey sidewalks and roads would be chosen as nodes. For efficiency we only check the rgb value of every fifth pixel in each row and column of the image. Also, to ensure that there are not an excessive number of nodes we impose a minimum node separation.

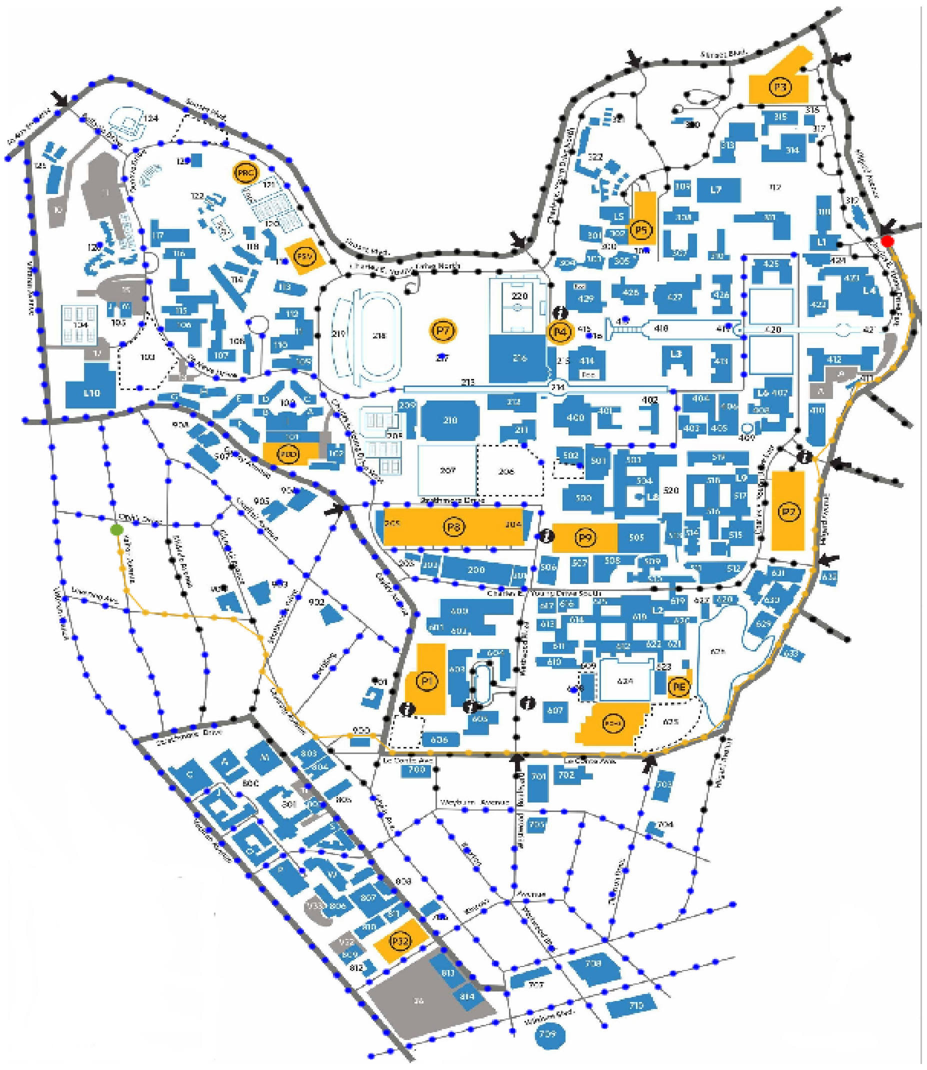


Figure 2: Path solution found from the intersection of Hilgard and Charles E. Young (red node) to the intersection of Ophir and Kelton (green node).

2.2 Fluid Dynamics in a 2D Lid Driven Cavity

The lid-driven cavity is a popular problem in computational fluid dynamics in which a viscous incompressible fluid is initially at rest in a non-slip cavity except for an initial horizontal velocity along the maximum y-position (the lid). The purpose of this program is to use a relaxation method to solve differential equations for the stream function and vorticity of the fluid. In their non-discretized form, these equations are

$$\nabla^2\psi = -\omega \qquad \nabla^2\omega = \frac{1}{\nu}\left(\frac{\partial\psi}{\partial y}\frac{\partial\omega}{\partial x} - \frac{\partial\psi}{\partial x}\frac{\partial\omega}{\partial y}\right) \tag{1}$$

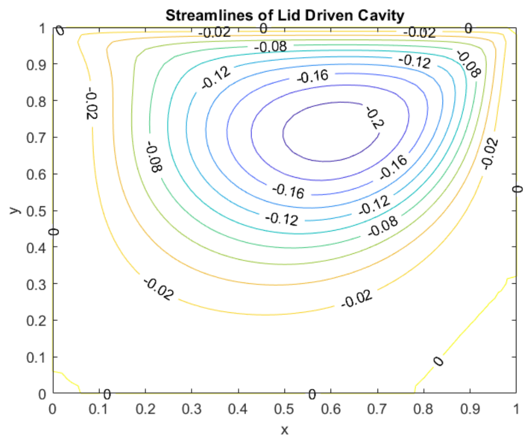
The vorticity ω is the curl of the velocity field, while the stream function is related to the velocities by:

$$u = \frac{\partial\psi}{\partial y} \qquad v = -\frac{\partial\psi}{\partial x} \tag{2}$$

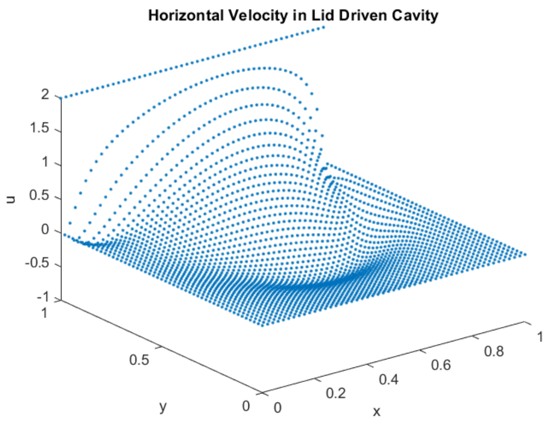
The program begins by prompting the user to input values for the constants of the system (number of nodes, Reynold's number, lid velocity). Next we initialize matrices for the stream function, vorticity, and the velocities. For each of these we assume that the initial value at every point is zero with the exception of the nonzero lid velocity in the last row of the horizontal velocity matrix. We then enter our main for loop in which the ψ and ω values at each node are updated using a relaxation method:

$$\psi\left(i,j\right)^{(k+1)} = \psi\left(i,j\right)^{(k)} + r\left(f - \psi\left(i,j\right)^{(k)}\right) \tag{3}$$

where r is a "relaxation factor" we set to 1 and f is a "residual," which we find by discretizing the differential equations and solving for $\psi(i,j)$. The while loop ends by plotting our results so we can observe the dynamics of the system. The program terminates once the main loop has reached the user-specified final number of iteration



(a)



(b)

Figure 3: (a) Contour lines of the stream function in a lid driven cavity. (b) Plot of the horizontal velocity. For both figures we used a 51X51 grid, a lid velocity of 2, a Reynold's number of 100, and 400 iterations.

2.3 Euler-Bernoulli Beam Bending

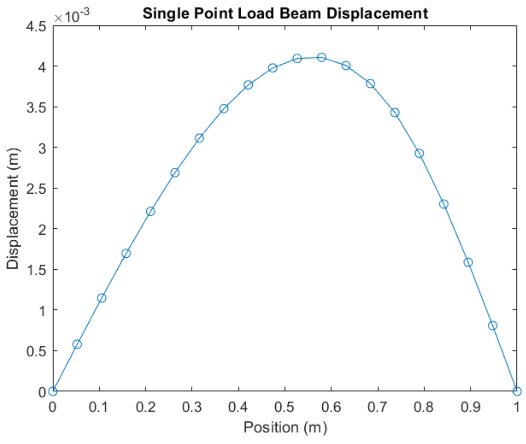
Solves for the displacement of a fixed beam subjected to a single point load. Uses the second order central difference method to discretize the following differential equation:

$$EI\frac{d^2y}{dx^2} = M\left(x\right) \tag{4}$$

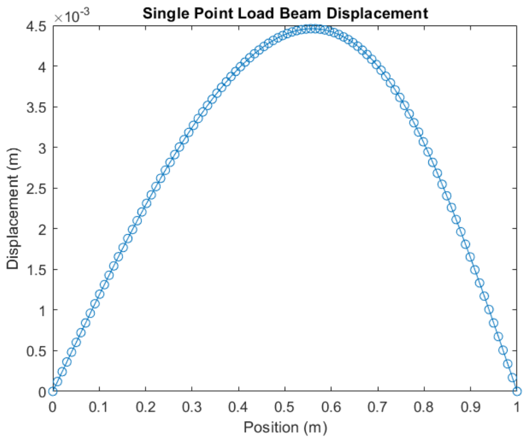
where E (elasticity) and I (cross section moment of inertia) are constants of the beam and M (the bending moment) is a function of node position. After discretizing using the second order central differencing method and rearranging we have

$$y_{x_{i-1}} - 2y_{x_i} + y_{x_{i+1}} = \frac{h^2}{EI}M\left(x_i\right) \tag{5}$$

This can be written as a matrix-vector product of the form $Ay = b$. Using MatLab's "dot operator" we solve for the displacement of each node using $y = A.b$ and plot the results against the x-position of each node.



(a)

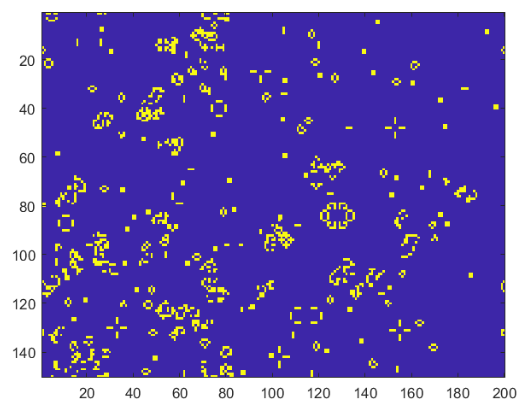


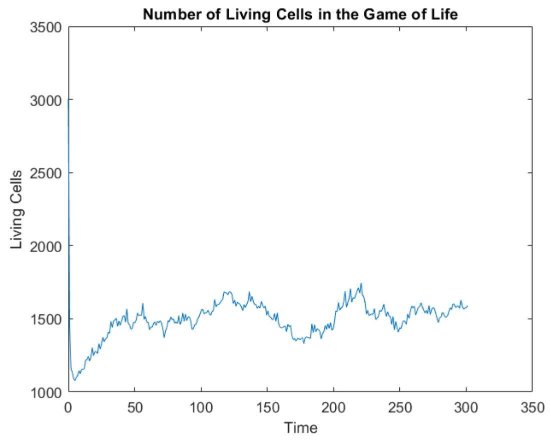
(b)

Figure 4: Displacement of the beam when discretized with (a) 20 nodes and (b) 100 nodes.

2.4 John Conway's "Game of Life"

Initializes a matrix of zeros (dead cells) and ones (alive cells) and allows the colony to "evolve" based on a set of survival rules. In each "generation" (iteration of the main for loop), we iterate through each element of the matrix and determine how many of its eight surrounding neighbors are living. Next we enter a series of if-elseif statements that use the number of living neighbors and the current state of the cell to determine its state in the next generation. Once we have performed these operations on every element in the matrix, we take the total sum of the matrix and store it so that we can plot the population dynamics over time.


(a)



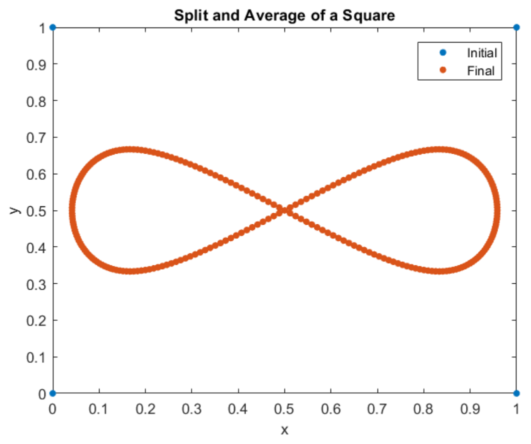
(b)

Figure 5: (a) Cell colony after 300 generations. Initialized with 10% alive (yellow) cells. (b) The living cell population over successive generations.

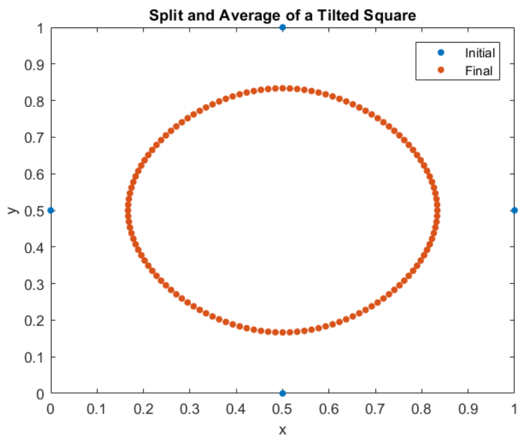
2.5 Split and Average Algorithm

Takes an initial set of coordinates and creates a new shape using a repeated "splitting and averaging" routine. The splitting routine creates a new array in which the average of neighboring elements is placed between them. The averaging routine takes the resulting array and creates a new one in which each element is a weighted average of itself with its neighbors according to a normalized weighing vector w:

$$xa_k = w_1 xs_{k-1} + w_2 xs_k + w_3 xs_{k+1} \tag{6}$$



(a)



(b)

Figure 6: The final split and averaged coordinates for initial coordinates corresponding to (a) a unit square cornered at the origin (b) a tilted square with corners touching the x and y axes.

where xa is the averaged array, xs is the split array, and k is an arbitrary index. The splitting and averaging routine ceases when the difference between xa and xs becomes less than a user specified limit.

3 Python Programs

3.1 Monte Carlo Simulation of a Spin Lattice

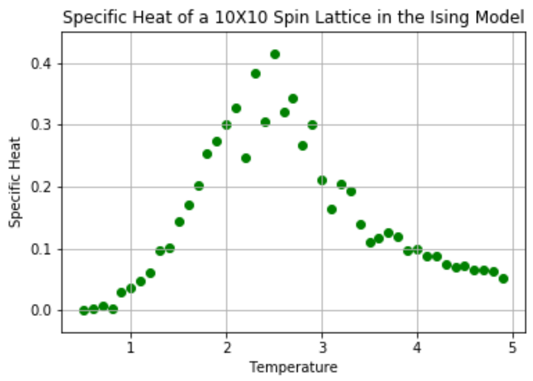
Initializes a matrix with a random distribution of 1 and -1 to represent a spin lattice whose normalized energy is given by

$$E_{\text{tot}} = -\sum_{i=1}^{N}\sum_{j=1}^{4} s_i s_j \tag{7}$$

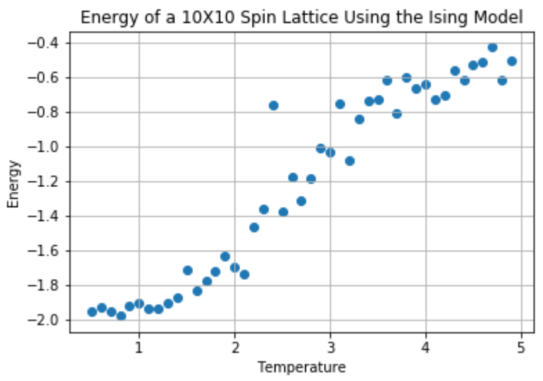
where i iterates over all spins in the lattice and j iterates over the nearest neighbors of s_i. We proceed by creating a Markov Chain via the following protocol: pick a random spin site s_i and calculate the change in energy as a result of flipping that spin:

$$\Delta E = 2s_i\left(s_{\text{above}} + s_{\text{below}} + s_{\text{left}} + s_{\text{right}}\right) \tag{8}$$

If $\Delta E \leq 0$ then the next state in your Markov Chain is that with s_i flipped. Otherwise, flip s_i with probability $\exp\left(-\Delta E/T\right)$. If the flip is accepted, add ΔE to the previous energy to obtain the new energy. If the flip is not accepted, set the new energy to the previous one. Once the number of energy values we have stored is sufficiently large, we can take the average. Because the probability of accepting a spin flip is temperature dependent, we can repeat this entire process for different T values and plot the temperature dependence.



(a)



(b)

Figure 7: Temperature dependence of (a) the average specific heat and (b) the average energy for a 2D spin lattice.

3.2 Solving the Quantum Harmonic Oscillator

Solves for the eigenstates and eigen-energies of the quantum harmonic oscillator given by the Hamiltonian matrix

$$\hat{H}_0 = \hat{a}_+\hat{a}_- + \frac{1}{2}I \tag{9}$$

where $\hat{a}_+$ and $\hat{a}_-$ are the raising and lowering matrices, respectively. When operated on by a particular eigenstate of the quantum harmonic oscillator, the resulting vector is proportional to the next excited eigenstate (in the case of $\hat{a}_+$) or the next eigenstate lower in energy (in the case of $\hat{a}_-$). We also have the property $\hat{a}_- |0\rangle = 0$ (there is no state lower than the ground state). The elements of the raising and lowering operators are defined by

$$\hat{a}_{+mn} = \sqrt{n+1}\delta_{m,n+1} \qquad \hat{a}_{-mn} = \sqrt{n}\delta_{m,n-1} \tag{10}$$

The anharmonic oscillator is given by the Hamiltonian matrix

$$\hat{H}_\lambda = \hat{H}_0 + \lambda\hat{x}^4 \tag{11}$$

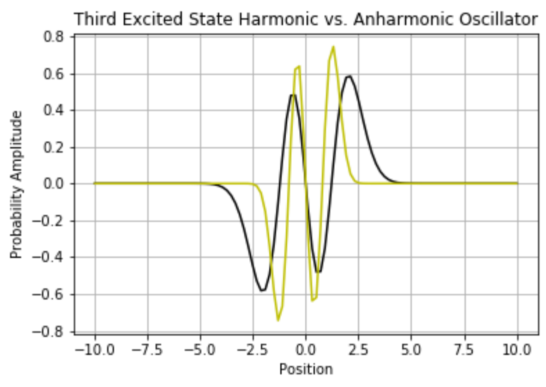
with $0 \leq \lambda \leq 1$ being the level of perturbation from the quantum harmonic oscillator. We can define position matrix $\hat{x}$ using the lowering and raising operators:

$$\hat{x} = \frac{1}{\sqrt{2}}\left(\hat{a}_+ + \hat{a}_-\right) \tag{12}$$

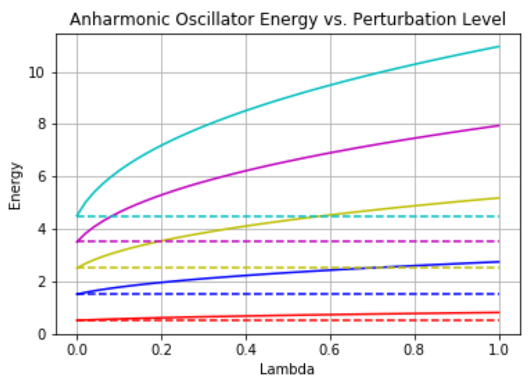
Therefore we can construct the Hamiltonian for the anharmonic oscillator by constructing matrices for the raising and lowering operators. We do this by defining functions that take as input the desired dimension of the matrices and construct $\hat{a}_+$ or $\hat{a}_-$ via Eqn. 10. Once we have the anharmonic Hamiltonian, we use linalg from the numpy library to solve for its eigen-energies and eigenstate vectors. From the eigenstate vectors we can construct (and then plot) the eigenfunctions using

$$\psi_n(x) = \left(2^n n!\sqrt{\pi}\right)^{-1/2} e^{-x^2/2} h_n(x) \tag{13}$$

where h_n is the n-th Hermite polynomial (here we use "hermval" from Numpy).



(a)



(b)

Figure 8: (a) black = harmonic, yellow=anharmonic. (b) Energy levels vs. perturbation.

3.3 Cosmic Ray Correlation with Sun Spots

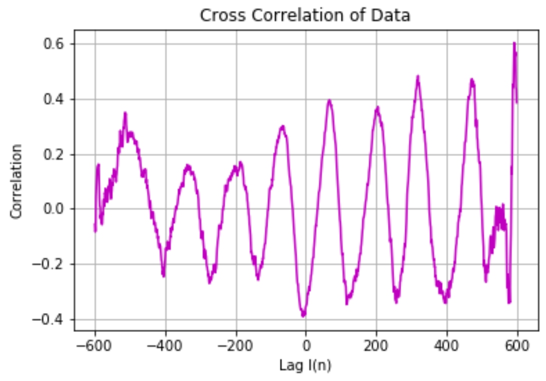
Sunspots are dark areas of intense magnetic field activity on the surface of the sun. During time periods with increased solar activity the sunspots act to strengthen Earth's planetary magnetic field. This in turn makes it more difficult for cosmic rays (a type of radiation originating from outside the solar system) to reach Earth and be detected. Our program begins by importing data taken over the past 50 years of both the average monthly number of sunspots and the average monthly count rate of cosmic ray detections. We can describe the nature of this relationship between the data sets by using them as the input arrays X and Y in the correlation function:

$$C_{X,Y}\left(l\left(n\right)\right) = \frac{1}{N - \left|l\left(n\right)\right|} F^{-1}\left(F\left(\tilde{X}\right) \cdot F\left(\tilde{Y}\right)^{*}\right) \tag{14}$$

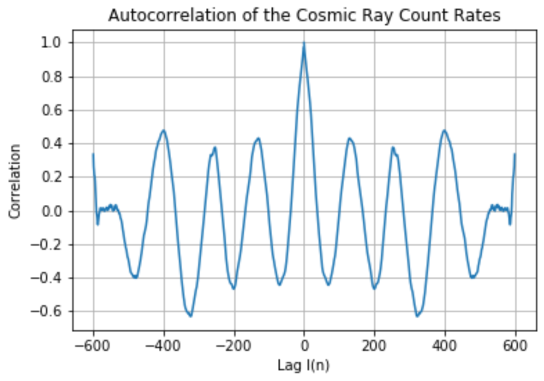
where F and F^{-1} are the Fourier and inverse Fourier transform, respectively, and l(n) is a "lag" function defined as

$$l\left(n\right) = \begin{cases} n & 0 \leq n \leq N \\ n - 2N & N < n < 2N \end{cases} \tag{15}$$

$\tilde{X}$ and $\tilde{Y}$ are X and Y after they have been "normed and padded". The norming routine is done via $\tilde{X}_n = \left(X_n - \bar{X}\right)/\sigma_X$ where $\bar{X}$ is the average of X and σ_X is its standard deviation. This normalization is convenient because it bounds the correlation function between values of -1 and 1. The padding routine simply doubles the size from N to 2N by appending N zeros. After importing the data our program defines a function to norm and pad the data as well as functions for the lag and correlation. Our function for the correlation uses numpy's fast Fourier Transform algorithm. We can evaluate our function using a range of lag values to plot the relationship between the data. As expected, the two data sets are fairly anticorrelated. We can use a particular data set as both X and Y (autocorrelation) in order to estimate its frequency and the uniformity of such oscillation.



(a)



(b)

Figure 9: (a) Relationship between the two data sets. (b) Determining the cosmic ray count frequency

4 C++ Programs

4.1 Lin-Kernighan Algorithm for TSP

My implementation of the Lin-Kernighan Algorithm, which creates a random distribution of points (cities) and attempts to find the optimized path which visits all points and finishes at the same point it started at. The program begins by randomly generating the points, assigning each point an ID, and calculating and storing the distance between each of them. Next we create an intial tour by generating a randomized list of the ID points such that none are repeated and the first and last ID are the same. We then iterate through the edges (lines connecting points) and perform a routine in which we remove the edge in question and create a new one. If the new edge has a shorter distance than the one we removed, we construct a new path by reversing the order of all points visited after the new edge. This results in the same endpoint that completed the removed edge.

For example, if the edge that we remove connected points a and b i.e. $E_{\text{rem}} = a \to b$, and our new edge is $E_{\text{add}} = a \to c$, then we will have produced the path

$$P_{\text{new}} = a \to c \to ... \to b \tag{16}$$

as opposed to the original path

$$P_{\text{original}} = a \to b \to ... \to c \tag{17}$$

and our new improved tour would be

$$T_{\text{new}} = s \to ... \to P_{\text{new}} \to ... \to s \tag{18}$$

where s is our start and finish point. The program ends when we have iterated through all edges and have not found an improved tour. I am currently working on making the program more efficient so that I can use more points, however as of now it is only able to find the optimized path for a small number points in a reasonable amount of time. Luckily these solutions can be easily verified with pen and paper so I know that the program is correct.

4.2 Betting Game

Plays a card game in which the user draws a card and bets the dealer whether a second drawn card will be higher or lower. Game terminates when the player's balance is negative (he has lost) or five times his starting amount (he has won). We create two classes. The first is entitled "Player" and it stores a string corresponding to the name of the player and their balance. We write functions to access the name and balance, as well as a function that can change the value of the balance. We also create a class called "Card" which stores random integers for the rank (between 1 and 13) and suit (between 1 and 4) of a card as well as corresponding strings. We write functions to access each of these.

The main function of the program begins by seeding the random number generator by time and prompting the user for a player name and starting balance. Once these have been inputted we initialize a “Player” data type. Next we enter a while loop that will terminate if the player’s balance ever exceeds five times the inputted starting amount or if their balance becomes negative. The first operation in the loop is to prompt the user to input a bet amount. Once it is inputted, we ensure it is not a negative number and that the player has enough money to make the bet. Next we create a “Card” variable for the first card, display its rank and suit in the console, and ask the user if the second card will be higher or lower. Once we have the response (“h” for higher or “l” for lower), we enter a series of if-else if statements that compare the rank (and if needed, the suit) of the two cards to determine if the the player’s balance will be increased or decreased by the bet amount. This concludes the while loop. Once we exit the loop, the program terminates by displaying a message which either congratulates the player or tells them they lost.

4.3 Dice Game

Program in which weighted dice are rolled until a desired sum is achieved. This is performed in a function that takes as input the number of times the user would like to play, the number of dice which will be rolled, the desired sum which will end a turn, and the probability of landing on each side of a die (six sides). The program begins by seeding the random number generator by time, declaring variables for all the function parameters, and prompting the user to input values for them.

The function contains an outer for loop to iterate through the number of turns. Within this is a while loop that performs the “rolling and summing” operations, which terminate when the sum of a particular roll equals the end sum. To account for the fact that we could be rolling multiple dice, we enter yet another for loop. In each iteration a “roll” occurs in which a random integer between 1 and 100 is generated and a side of the dice is chosen via a set of if-else if statements that use the probabilities inputted into the function. The side that is chosen is added to the sum from the previous die. Once all dice have been thrown we increment a variable called“rolls”. If the sum from all the dice is equal to the desired sum then the while loop ends and the next turn (set of rolls) begins. Once the desired sum has been achieved on every turn, the function returns the total number of rolls. The program terminates by displaying to the console the average number of rolls per turn.

5 Potential Future Projects

5.1 Self-Driving Car Simulation

A MatLab program which incorporates path-finding on an image of a map (see Sec. 2.1) with routines that determine the actions of a "car" as it travels along its desired route. For example, we can program a car to stop at a red light by giving nodes which occur at intersections "stoplight states". If a state is red (false) in the direction that the car wishes to travel, it must stay stationary unti the state changes to green (true). We can also put multiple cars on a given route and program their behavior to avoid collisions while maintaining efficient travel. Perhaps we will perform the simulation on a street map of midtown Manhattan in order to lend it some real-world practicality.

5.2 Personal RSA Algorithm

A C++ program in which the user inputs two large prime numbers and a message they would like to have encrypted. The program then uses the prime numbers to generate a public key that encrypts the message as well as a private key to decrypt it. The program will terminate by displaying to the console the encrypted message as well as the decrypted message to ensure that it matches with the original, verifying that the program is correct.