Last updated: 10/18/20

Math 260, Fall 2020: Final project

Project roadmap

The project will consist of writing a substantial code to solve a problem inspired by a real-world application, and discussing the results. You **may work in pairs** for this project, and are encouraged to do so. Your work will be maintained on a github repository.

The project is organized into a series of 'milestones'. At each step you are expected to have certain work done. Note that the content requirements for milestones can be flexible, but you should adhere to the deadlines. All deadlines at 11;59 PM.

- **(Tues.) Oct. 27**: Topic/partner selected (required: discuss with me)
- **(Wed.) Nov. 4**: Plan, scope, first steps (Milestone 1)
- **(Wed.) Nov. 11**: Working code (Milestone 2)
- **(Wed.) Nov. 18**: Project 'almost-final' draft due (Milestone 3)
- **(Tues.) Nov. 24**: Project due at **6:00 pm**. No later work accepted.

1) **Exploration, plan:** Three related items here: plan, initial research, and a first step.
   i) A plan identifying the subject of your project - the problem and algorithm of interest, and, broadly, what you intend to do. The details may change later, but the plan should include concrete first steps.
   ii) Explore your topic. Find relevant resources, and get a sense of the topic so that you know what concepts to learn, what to study in detail and so on.
   iii) Similar to (i), identify a clear first step for coding - for instance, writing a simple test case, implementing foundational pieces (e.g. a tree class for binary trees).

2) **Working code, scope:** Two tasks here: complete the first step, define the plan.
   i) Decide (almost for sure) on the scope of your project. What problems will you try to solve, what aspects of the code will you address? What will be left out?
   ii) Have a working version of the algorithm that solves a relevant test case. This code will be expanded upon as you add features, twists to the problem etc.
   Your code should have 'debugging/testing code' to verify correctness

3) **Almost-final draft:** A (nearly) content-complete version of your project. All coding should be complete and (mostly) debugged, along with drafted discussion of your topic and results. Plan only for revisions, not adding anything new.

4) **Final version:** The final version of your project, committed to your github repository.
   The contents of the repository committed by the deadline will be graded.

Content guide and requirements

Your project has three components: exploration, production and discussion. Only the latter two are graded, but the first provides an essential foundation.

A. Exploration/Study (not graded): Background, foundational work.

- **Context:** Learn about the underlying mathematics of your topic and the algorithms involved. What problems are being solved (and why)? How do the math and problem constraints inform the algorithms? What are the computational challenges?
- **Implementation:** Understand the workings of the key algorithms and how they are translated into code. What concerns must be addressed for implementation (efficiency, design, etc.)? Learn any necessary coding tools, such as extensions of tools from class, programming tricks or data structures.

B. The product (graded, $\approx 85\%$): The core of the project - code and computation.

- The code base will be submitted in the form of a **github repository**. You are also required to use the repository to update your project as it evolves.
- Understand the relevant algorithms, implement and test them. The code should be functional and (briefly) documented. Provide test cases that demonstrate functionality.
- Apply your code to a computational problem as identified in the exploration phase. You may have some freedom to play around in this space - and be willing to adjust your initial questions to pursue what is interesting or feasible.
- Write 'good python code'. Make your code clear, organized, and implement the algorithms in the 'right way' (or choose simplicity instead, if you intend to use your time to focus on other aspects).

C. The discussion (graded $\approx 15\%$): Say something about your work - roughly one page.

- Write your document in LaTeX. (I'll post templates that include formatting python code and math -just the bare minimum is needed here; learning TeX a low priority).
- **Description:** Give an informal discussion of your code - how it works, how you organized the code, and and the main computational challenges (what is notable). This is not formal documentation - more of an overview of the code and its purpose. You can include a bit of the mathematical context, but this is not a focus.
- **Discussion:** Discuss the problem of interest and your results - interesting features, what matters about the results, and so on. For some topics, you may be discussing computational aspects (e.g. comparing efficiency, does the algorithm always work, behavior of errors...) and in others you may be discussing the application (modeling context; does it answer the question posed by the problem, etc.)

1 Project topics

The project topics are, in some cases, broad, so you have an opportunity to select a sub-topic of interest. If you are interested in an extension of a **topic from class**, that can probably be done (ask me)!

It's possible to branch out and explore an adjacent topic or something else, but do not assume that a new suggestion will be accepted (it's certainly possible, but does need to be feasible).

Topic A: Fractals and root-finding (miscellaneous):

- **background:** You are likely familiar with the famous Mandelbrot set - a **fractal** with incredibly complex structure. These fractals are interesting to compute, as they can be 'zoomed in' infinitely to obtain finer and finer structure.

  While the dimension of a rectangle and a line in the plane (2 and 1) are obvious, its a remarkable fact that fractals have dimensions somewhere in between. This is a quantity that you can compute numerically with the right algorithm.
- **your work:** Write code to create some fractals - for instance, the fractals related to Newton's method (and learn Newton's method along the way, although this is incidental). Play around with this type of fractal to get interesting shapes, and visualize the fractal with pyplot. Then - the computational focus - calculate the 'dimension' of the fractal using boxes, and see how far you can get with it given the time constraints and computer power you have.
- **notes:** You'll need complex numbers and an interest in using the plotting tools of python.

Topic B: Swarms and flocking (solving ODEs)

- **background:** Systems of particles that interact can be explored by solving an initial value problem for a system of ODEs. One fascinating example can be found in nature - groups of birds mysteriously organize into flocks ('swarming'). We can model this with a system of ODEs, defining an interaction between birds, and see when they will form a flock and when it will break apart.
- **your work:** You will need to implement a suitable integration method (RK4 works). You will implement a solver that can simulate the system and then explore its behavior, producing some plots (or other measures to distinguish features). Time permitting, you may try to make your code more efficient to add more and more particles, but you will not have time to optimize fully.
- **notes:** There is some tricky setup required to get the ODE system to work. This is an initial value problem where the hard part is writing the ODE function. The focus here is on *using* an ODE solver, not implementing it, although it's worth implementing a solver (like RK4) along the way.

Topic C: Signal processing (Fourier applications)

- **background:** Further exploration of the Fourier transform introduced in class. The idea is to find some signals to process and a goal, and then write code to accomplish that goal. Examples might include, for instance, music (synthesizing notes), removing noise or cleaning up signals or digital filters.

  Another Fourier transform application. In this case, you'll get into processing signals with the discrete Fourier transform in more depth. For instance - with it, you can create a synthesizer (that plays notes with a desired sound), do other processing on a real signal. instance, removing noise...)
- **your work:** You'll be using the numpy implementation of the fast fourier transform - the work will be in designing and implementing the appropriate processing code. Note that you will also need to gain a deeper understanding of the subtleties.
- **notes:** We cover the discrete Fourier transform during the first half of Week 5.

Topic D: Solving PDEs numerically [many options here]:

- **background:** We briefly introduce PDEs in the last few weeks. Solving them numerically involves reducing to ODE initial value problems and/or linear algebra problems (LU factorization is useful!). There are innumerable reasons to care about PDEs that can motivate a topic, from modeling traffic jams, chemical reactions, fluid flow and much more. This is obviously a deep subject, so you'll be exploring a specific problem and method (see: the heat equation, Laplace's equation, advection-diffusion)
- **your work:** Pick a topic, an associated PDE and write code to solve it numerically. This will mostly involve linear algebra (LU factorization or iteration like what we did with the power method) or just iteration, but there are more subtleties.
- **notes:** Essentially no ODE theory is required, but you will need to be comfortable with finite differences (e.g. the 'meshes' of points for estimating integrals/derivatives), and a bit of linear algebra.

Topic E: Image compression (Fourier applications)

- **background:** We'll cover the Fourier transform during the start of Week 5. A variant is the Discrete Cosine transform, which you can use to compress images. The idea is to cut off high frequency data (what does this mean for an image? You'd have to explore this) from the transformed image. The FFT can also be used, but doesn't work as well!
- **your work:** Implement the algorithm, making use of the FFT/DCT functions in numpy (there are a number of steps required to go with it). Vary the parameters a bit or try using the FFT instead of the DCT and see the differences - which works best?
- **notes:** You will likely get to simplify a bit; you want an algorithm that represents the method and illustrates the key features. The work involves the '2d' Fourier transform - the implementation is short given the 1d version, but it's subtle and requires managing a lot of indices.

Topic F: Arbitrage and Bellman-Ford (graphs)

- **background:** Suppose you have a table of exchange rates for currency. Can you trade one for another in succession to make a profit? This process is called **arbitrage**, and has obvious applications in the stock market. You can find such opportunities by creating a weighted graph of these exchanges; then the problem can be reduced to finding a certain optimal path or cycle, which is done with the **Bellman-Ford** algorithm.
- **your work:** You will build up some code to represent the graph in the appropriate form, then implement the Bellman-Ford algorithm. Time permitting, you can apply this to some real data (e.g. current exchange rates), although no set of real data provides a positive example [all arbitrage opportunities in real markets are snapped up immediately!].
- **notes:** The algorithm takes a bit of work; it's more substantial than the recursive depth-first search example. I should also note that you must be careful to not search too closely for examples, since there is alot of python code floating around. Part of the challenge will be trying to write the code (representing the graph, the algorithm) in a nice way.

Topic G: Simulated annealing (optimization)

- **background:** A probabilistic algorithm for minimizing an energy'' $E(\mathbf{x})$ over some space - e.g. finding the minimum of a function, the optimal path in a graph, and much more. The idea is that by stepping 'downhill' only, you may get stuck in a local minimum. If you randomly allow small steps in the wrong direction, the algorithm can escape and keep looking for the true minimum.
- **your work:** Understand the basics of simulated annealing, construct an example problem and write code to get it to work. Then, play around with the various features to see what is important, what makes it more efficient, challenges etc. There's a number of features that need to be adjusted - getting it right is part math, part intuition, part guesswork.
- **notes:** This topic is somewhere between random walks and gradient descent. Only a bit of probability background is required. There may not be a good 'standard' resource for learning the basics, so you may have to cobble it together (with guidance).

Topic H: k-d trees or quadtrees (computational geometry)

- **background:** Suppose I have a set of n points in the plane. Asking what point in the set is closest to x takes $O(n)$ operations when checking directly. However, we can construct a tree structure that better stores the positions of the n points to reduce this to $O(\log n)$. Two structures are 'k-d trees' (in any dimension), which area special type of binary tree, and 'quadtrees' (in 2d), which are trees with four children that recursively partition the plane into four quadrants. Quadtrees are used in computer graphics, for instance, to efficiently store image data for use with algorithms that need to quickly locate points.
- **your work:** Choose either a k-d tree or quad tree and implement it. Write the code and make it reasonably efficient, and the operations you can do on the tree.
- **notes:** This topic is all about recursion and trees (think mergesort, depth-first search etc.), and requires a good amount of figuring out algorithm details (like we did for insertion/deletion of nodes into trees).