Mathematical Programming MATH20014: Group projects

March 12, 2020

Outline of the group projects

The group projects for MATH20014 will be organised as follows. From the set of potential projects detailed below, each students will choose a favourite project and a backup project, according to their interest. The projects are all of a similar level of difficulty, but focus on different areas of mathematics, including mechanics, statistical mechanics, statistics, cryptography, number theory and knot theory.

Each group will then consist of four students who showed interest in the same project, and who (as much as is feasible) also share the same tutorial group.

Please select your first and second choice of group project through the Blackboard test.

Format of the group projects

We expect the group project to take the form of a jupyter notebook, similar to the lectures and tutorials.

Your jupyter notebook will contains the different pieces of code, and the same jupyter notebook will also need to contain explanations, formulas, graphs, context and references for your project.

As you have seen in the lectures and tutorials, the notebooks understand latex, so typesetting equations is straightforward. On the other hand, it is difficult to include external images into them (as opposed to plots generated from your code, which are fine). Note that html image links break as soon as you move from your own machine, even when they shouldn't.

Therefore, in addition to the core jupyter notebook, we encourage you to create and submit other forms of content, including (but not limited to)

- Python code files (.py) with, for example, modules that you have created, or scripts that carry out the time-consuming calculations required by some projects.
- High-resolution images and movies obtained as output from your program
- Additional notes with graphs and images, as well as explanations or discussions, as a compiled latex file (i.e. a pdf), or even as a Word file.

Assessment criteria

The projects here contain a core part (marked **[core]** below) which you need to satisfactorily complete to obtain a passing mark. The other parts are pick-and-choose, and open ended. Grading criteria:

- Correctness of the code, and its ability to perform the computations required in the **core** parts of the project.
- Quality of your code, including structure, number and usefulness of comments, and use of functions, arrays, libraries and other elements introduced in the course.

- Scope of your project, i.e. how far the topic has been developed beyond the **core** parts.
- Quality of the surrounding explanations and analysis, and quality of the typesetting and English language.

There will also be a peer assessment of your group where you rate the commitment and performance or your group members. This will take the same form as for the group projects in Mathematical Investigations in year 1, i.e. a blackboard from. **The projects and peer assessments are due on Monday, May 11 (i.e. the first day of revision week).**

How to collaborate on a computing project

The first step is to make sure to have a shared drive that is accessible to all members of your team, for example by creating a shared folder on Dropbox, or on Onedrive. Then create your jupyter notebook for the project, and edit it. A key issue is **version control**, i.e. that there is only one current version of the project (and not, say, four different branches that four different people are working on). At the same time, you will want to keep the previous versions of the notebook, so that potentially important pieces are not overwritten or lost and you can always undo changes that were a mistake.

The simplest (though not the most efficient) form of version control is to save a copy of the jupyter notebook with the current date and your name each time you make an edit, and to always, always start from the most recently modified file.

If you are keen to explore more formal options for working on a coding project as a team, please consider using the **git software**, and the github online code sharing platform (https://github.com/).

1 The two-dimensional Ising model

In solid state physics, to understand how magnets works, a very simple model has been developed. We represent the individual magnetization (i.e. spin) of each molecule by a one dimensional vector that can either point up or down, that is a variable $\sigma_i = \pm 1$. We want to understand a ferromagnet where neighbouring spins want to align with each other. A simple way to do this is to assign an energy E_i to each spin, a with smaller energy for aligned spins then for non-aligned ones.. For the whole magnet, we define

$$E = \sum_i E_i = -J/2 \sum_i \sum_{j \text{ nei } i} \sigma_i \sigma_j \tag{1}$$

Here the fist sum runs over spins on a two dimensional lattice, and the second sum runs over all lattice neighbours j of spin i. As you can see, when spins are aligned, $\sigma_i \sigma_j = 1$, and the energy of a pair of spins is one unit of interaction energy J lower than when they are not aligned, and $\sigma_i \sigma_j = -1$.

To simulate the effect of **thermal fluctuations** of the individual spins on the magnet, we use what is called a **Monte Carlo** (MC) algorithm. In a MC algorithm, we choose one spin, flip it (i.e. $\sigma_i \to -\sigma_i$), and then accept or reject this move depending on how likely it is to happen at that given temperature. The probability of this flip is:

$$\begin{array}{ll} 1 & \text{if} \quad \Delta E < 0 \\ \exp(-\Delta E/kT) & \text{if} \quad \Delta E > 0 \end{array} \tag{2}$$

In the long run, after many flips, this algorithm converges to a Boltzmann distribution $P(E_i) \sim e^{-E_i/kT}$ for a system in thermal equilibrium. These concepts will be discussed in detail in Statistical Mechanics class in 4th year. This particular, simplest, Monte Carlo algorithm is called the **Metropolis algorithm**, after Nicholas Metropolis (Metropolis et al., 1953).

Your job is to simulate the Ising model on a two-dimensional square grid with periodic boundary conditions for $J = 1$ via the Metropolis algorithm, for values of the temperature between $kT = 0$ and $kT = 4$.

The main quantity you will want to focus on is the **mean magnetization** m, that is the average orientation of spins $m = |\sum_i \sigma_i / N_{\text{spins}}|$. As a function of temperature, the mean magnetization will go from a positive value to approximately 0 as a function of temperature, with a transition temperature of $kT_c = \frac{2J}{\log(1+\sqrt{2})} \approx 2.27J$. Detailed instructions:

- **[core]** Create a $N \times N$ square grid/array of spins σ, and initialize your system randomly with ± 1 values; a good starting value would be $N = 10$. Calculate the initial energy of the system, E_{ini} (choose $J = 1$). Create a plot of your system that shows the different spin values as colors.

- **[core]** Now compute the Monte Carlos algorithm: At each step, randomly choose one of your spins $\sigma[k, l]$; that means it has a uniform random index in both directions. Calculate the initial energy of the system, E_{ini}. Then tentatively flip the spin, and calculate the new energy E_{fin}, to obtain the energy difference ΔE. If you are smart, you can compute ΔE using just the values $\sigma[k, l], \sigma[k \pm 1, l]$ and $\sigma[k, l \pm 1]$. Please employ **periodic boundary conditions**, as explained in the supplementary information on the last page of these notes. If $\Delta E < 0$, save your flipped spin value to your array. If $\Delta E > 0$, only save your flipped spin to the array with probability $\exp(-\Delta E/kT)$, otherwise keep the original unflipped state. That means choose a random number r between 0 and 1, and keep the flipped state if $r < \exp(-\Delta E/kT)$.

- **[core]** In any case, whether you flipped your spin or not, keep a record of E and the mean magnetization m for this step. You have now simulated a thermal fluctuation event.

Repeat the last four points for a large number of steps, $T_{tot} > 100N^2$, representing a long time of fluctuations. Create a plot of the mean magnetisation m as a function of Monte Carlo step (note that this does not quite correspond to time), with appropriate labels and title.

- Repeat all of the above for a range of values of the temperature kT, preferably by putting your code above into a function. Save the traces for E and for m in a separate csv file for each run with an appropriate name. Now compute your $m(kT)$ phase transition graph, from averaging the m traces for the portion of your run that has equilibrated, i.e. the part where E does not systematically reduce anymore. Make it pretty; this is your main result! Error bars are appreciated, too.
- Create a movie of the spin dynamics in your system.
- Consider an **antiferromagnet** where spins want to anti-align with each other, corresponding to $J < 0$. Which types of states do you observe (a system plot here is necessary), and what values of m do you find?

2 Planetary motion

We want to investigate how planets and comets move around the sun. There are several possible scenarios for this exercise; pick one. Everything starts with the Newtonian equations of motion for the positions of each object i:

$$m_i \ddot{\mathbf{r}}_i = G \sum_j m_i m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3}, \tag{3}$$

where the m_j are the masses of the other objects, $\mathbf{r}_j$ are their positions and $G = 6.6734810^{-11} m^3/kgs^2$ is the universal graviational constant.

Gravitation is a conservative force; that means that the total energy of the system, $E = T+V$ does not vary in time:

$$E = \frac{1}{2} \sum_i m_i |\dot{\mathbf{r}}_i|^2 - G \sum_i \sum_j \frac{m_i m_j}{|\mathbf{r}_j - \mathbf{r}_i|} \tag{4}$$

It is theoretically difficult to find a numerical integration scheme that is consistent with energy conservation, or what is called Hamiltonian dynamics in classical mechanics. If we find it, such an algorithm is called a *symplectic integrator*. The standard algorithms that we have discussed, like the Euler method or the Runge-Kutta algorithm, are not symplectic integrators. Instead, we are going to use the symplectic Euler algorithm, also known as the Newton-Stormer-Verlet algorithm, which superficially resembles Euler. Suppose that our equations of motion can be written as standard Newton equations

$$\dot{\mathbf{r}} = \mathbf{v} \tag{5}$$

$$\dot{\mathbf{v}} = \mathbf{F}/m \tag{6}$$

Then in the Verlet algorithm, we first update the velocity, and then with the *updated velocity*, we update the positions:

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \frac{\mathbf{F}_k}{m} dt \tag{7}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k + \mathbf{v}_{k+1} dt \tag{8}$$

Simulate the motion of at least one planet with the Verlet algorithm.

- **[core]** While the equations above are written in three dimensions, conservation of angular momentum makes the orbits of planets planar. All in all, for a single planet, you will be left with only two degrees of freedom, i.e. two coupled second order ODEs, corresponding to four coupled first-order ODEs. First simulate only the motion of an earth-like planet around the sun. The mass of the sun is $1.989\ 10^{30} kg$, and the radius of the earth is on average 149.59787 million kilometers (i.e. $R = 149.59787\ 10^9 m$). Assume that the sun is fixed and that only the earth moves (a pretty good approximation in practice).

- **[core]** Verify that you find a closed orbit for a planet with the earth's initial position $\mathbf{r}_0 = (R, 0)$ and tangential initial velocity $v0 = (0, 29.8 km/s)$. Check that the total energy (kinetic plus potential) remains constant. Analytically derive that this value of v_0 is the only one compatible with a (near) circular orbit. You will find the Kepler problem homework from Mechanics 2 very useful here.

- **[core]** Now vary the total energy of the orbit by changing the value, but not the direction, of v_0, between 0 and a maximum value to be determined below. Plot the now elliptical and hyperbolic trajectories, and the positions of the aphelion/perihelion. For the elliptical trajectories, calculate the orbital eccentricity (defined as the ratio of the minor to the

major axis). Construct a criterion to distinguish elliptical and hyperbolic trajectories (only one ever comes back to the start). Determine the escape speed of an earth-like planet from the sun's gravitational well. Determine the corresponding energy, and compare it to the theoretical result, which you can derive from $T + V|_{\text{initial}} = T + V|_{r=\infty}$

- For the elliptical trajectories, test Kepler's second law, which states that "A line joining a planet and the Sun sweeps out equal areas during equal intervals of time". In other words, if A is the area enclosed between the sun at the centre and the planetary position at (r, θ) in polar coordinates, the rate of change of area $\frac{dA}{dt} = \frac{r^2}{2}\frac{d\theta}{dt}$ is constant.

- Investigate the **three-body problem** where three objects interact with each other gravitationally. For example, consider three sun-like stars with different but not too dissimilar masses. The three-body problem does not have an analytical solution and frequently leads to chaotic trajectories.

- Simulate the solar system including multiple or all planets (and yes, planets interact with each other). This is harder than it looks, since the periods of the outer planets are much longer than the inner planets. We therefore obtain *numerically stiff* differential equations, where we need a small time step because of the inner planets, but also a long simulation time for the outer planets. Several compromises are possible, e.g. only simulating inner or outer planets, and not simulating long enough to find closed trajectories for outer planets.

3 The Vicsek model

How do flocks of birds or swarms of fishes work together to move so coherently in large groups? Do they have a leading fish or bird that directs the motion? Research indicates that this is not the case, instead collective motion is a spontaneous phenomenon that emerges from very local interactions of individuals with each other.

The first, and simplest, model of flocking in animal groups was proposed by Tamas Vicsek et al. (PRL 1995). Each bird is represented by a point particle that moves at constant velocity v_0 along its polar direction $\hat{n}$. At each discrete time step, each bird checks its neighbours within a radius R around its position, and then reorients itself along their mean direction. Since birds are imperfect, there will be noise added to that new direction. In a two-dimensional plane, the equations of motion for bird i at step $k+1$ are:

$$\vec{r}_i(k+1) = \vec{r}_i(k) + \hat{n}_i v_0 \tag{9}$$

$$\theta_i(k+1) = \text{angle}\left[\sum_{j=1}^{z_i} \hat{n}_j\right] + \eta_i \tag{10}$$

Here the polar direction is $\hat{n}_i = (\cos\theta_i, \sin\theta_i)$. The sum in the second equation is over all z_i neighbours within radius R, and we take the angle of the final vector. To this, we then add a normally distributed noise η_i, with mean 0 and variance σ, i.e. at each step, η_i is chosen from a $N(0,\sigma)$ normal distribution. Note that this is not a set of ODEs, but a model with a unit time step.

As a function of σ, the Vicsek model undergoes a phase transition from an aligned, moving flocking state at low σ to a randomly moving state at large values of noise σ. Simulate the Vicsek model for $R = 1$, $v_0 = 0.5$ and a number $N = 200$ individual birds for at least 1000 steps. At every compute the Vicsek order parameter

$$n = \frac{1}{N}\left|\sum_{i=1}^{N} \hat{n}_i\right|. \tag{11}$$

which is a measure of the alignment between birds and has values between 0 and 1. Detailed instructions:

- **[core]** Create your birds with uniformly random positions and uniformly random initial orientations in a $L \times L$ container, with suggested $L = 15$, or in general roughly $L = \sqrt{N}$. You will crucially need periodic boundary conditions for the positions: When you birds cross the system boundary to the right, they should appear on the left again, going in the same direction. The same applies for birds that disappear at the top, they need to reappear at the bottom. Please see the supplementary information at the end of this document.

- **[core]** For equation 10, you will need to determine the distances of *all* the other birds and then choose the ones closer than R. Again, you will need periodic boundary conditions here as birds will also interact with other birds "over the edge". Equation 10 is tricky to implement numerically! A clean way of doing so is to do the two-dimensional vector sum of the $\hat{n}_j$, normalize the result, and then take the angle of it (you won't need to divide by the number of neighbours z_i anymore, either).

- **[core]** Run your system for a long time, at least $T = 10000$ steps, and compute and record the Vicsek order parameter at each step. Plot the Vicsek order parameter as a function of time. Also, make a couple of representative plots. The matplolib 'quiver' plot creates a field of arrows, and is well suited for this task.

- Make one or several movies of your flocking simulation at different σ.
- Compute the $n(\sigma)$ phase transition plot: For each σ, run one simulation; your code so far should go into a function for this. Save the output n traces in a csv file for each run of your system. Finally, compute the average n as a function of σ. For this, remove the values of n for early times before the system reaches steady state from the average. Create a plot with appropriate error bars and labels.
- As an extension, add a bird of prey to your system, i.e. a particle that changes its orientation of motion towards the position of birds within a radius R_{pred}. Conversely, the other birds near enough the bird of prey will gain an extra term in equation 10 that turns them away from the position of the predator.

4 Random walks

When very small objects, such as dust particles or individual molecules, move through space, they do not do so in a straight trajectory. Instead, they will constantly bump into other molecules that jostle them around. From a (short) distance, it appears as though each particle performs a *random walk* where it randomly goes switches direction every so often. Your task is to simulate one or multiple types of random walks. From an initial position $\mathbf{r}_0 = (0, 0)$ in (x, y) coordinates, the local dynamics is:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + (\cos\theta, \sin\theta)\Delta, \tag{12}$$

where θ is a random angle uniformly distributed between 0 and 2π, i.e. a completely random direction. Choose $\Delta = 1$ as spatial unit.

- **[core]** Create a code that simulates an individual random walk of T steps (suggested $10^3 < T < 10^6$), and store all of the intermediate points in a numpy array. You will want to allocate your array. Plot the trajectory of the random walk with matplotlib, and generate a couple of clear and pretty pictures.

- **[core]** Since the trajectory is random, to obtain a clear idea what is happening on average, we need to perform a statistical average. To do so, repeat the simulation a number of $N > 10$ times. Modify your code so that your initial code is a function (or class) that is called by a loop. Make sure to use a new random seed each time. Save each run in a separate csv file with an appropriate name.

- **[core]** Compute the **mean square displacement** (MSD) of your trajectory. This means the following: Formally, the MSD is the statistical average of the square distance the particle moves in a time δt, i.e.

  $$\mathrm{MSD}(\delta t) = \langle |\mathbf{r}(t + \delta t) - \mathbf{r}(t)|^2 \rangle_{t,\mathrm{ensemble}}. \tag{13}$$

  Here the ensemble average means that the MSD is the mean over different runs of your system (you have that data saved already). The t-average means that you will need to average over all t that exist such that both $0 \leq t < T$ and $0 \leq t + \delta t < T$. In detail, this means:

  $$\langle |\mathbf{r}(t + \delta t) - \mathbf{r}(t)|^2 \rangle_t = \frac{1}{N_t} \sum_k |\mathbf{r}(k + \delta t) - \mathbf{r}(k)|^2, \tag{14}$$

  where N_t is the number of k's such that this condition is fulfilled. You can have your code compute N_t directly as you do your sum. It is possible to use numpy slicing together with np.mean(...) to do the sum in a single step; this is also faster than a loop.

- **[core]** Plot your final MSD as a function of δt. Include errorbars $\sigma = \mathrm{std(MSD)}/\sqrt{N}$, where std(MSD) is the standard deviation among the different runs and N is the number of runs. This is the formula for the error of the mean for statistically independent, normally distributed time series. You should find that your curve resembles

  $$\mathrm{MSD}(\delta t) = D\delta t, \tag{15}$$

  i.e. it is *linear* as a function of time in spite of being the *square* distance the particle goes. That means that the particle undergoes *diffusive* motion where it advanced only proportional to the square root of time. Compute your diffusion constant D from the curve, and check that $D = 2\frac{\Delta}{dt}$, where $dt = 1$ is the time step you have used in your simulation.

- In a **persistent random walk**, the particle does not completely change direction at every time step. Simulate a persistent random walk using the equations

$$\mathbf{r}_{i+1} = \mathbf{r}_i + (\cos\theta_i, \sin\theta_i)\Delta \tag{16}$$

$$\theta_{i+1} = \theta_i + D_r\eta_i, \tag{17}$$

  where η_i is chosen from a standard normal distribution $N(0,1)$. Compute the MSD of this walk, and show that the curve scales as $MSD \sim t^2$ for $t < 1/D_r$, and $MSD \sim t$ for $t > 1/D_r$, and that the diffusion coefficient is given by $D = 1/(2D_r)$. You will want to choose D_r
  1 for a noticeable effect.

- Suppose that you, a random particle staggering out of a bar next to a sheer cliff face at 1am on a Sunday morning, are attempting to make it back to your flat but you are not sucessful. Starting from the bar located at $(0,0)$ you instead hit the cliff, a north-south line located at $x = L$, and you very unfortunately fall down. Calculate your mean life expectancy $\tau(L)$ before you to meet your end, as a function of the distance L of the beach. Reasonable starting values would be $dt = 1s$ as before, $\Delta = 1m$, and $L = 5m$. This type of time to reach a certain distance or threshold is called a **first passage time**.

5 Cryptography

The purpose of this project is to analyse various approaches to encryption and decryption and to understand the security implications involved. The **core** part of the project consists of sections 1-5 below. In each section you are expected to discuss the relevant material and methods and to write and test algorithms in python that implement these methods. (For background material you may find [1, 2, 3] and section QA 76.9 in Queens Library useful.)

1. **(core)** Discuss how to encrypt and decrypt messages using the *Caesar* cipher and the *Vigenère* cipher and implement algorithms to encrypt and decrypt messages using both these methods. Also work out how you can systematically crack these ciphers. Implement and test functions to do this.

2. **(core)** A common use of the *RSA* protocol is to digitally sign a message. Suppose that (p, q) and (N, e) are the private and public keys computed by Alice as seen in lectures (so p, q are 512-bit primes etc). Then Alice *hashes* her message into a nonsense sequence of bits (*digested* into bytes) and converts this into an integer h. Alice computes signature $s = h^f$ modulo N where f is the multiplicative inverse of e modulo $\phi(N)$—the *totient* of N—and sends s to Bob with the message. On receiving the message Bob also hashes the message (using the same hash function) and converts the outcome to an integer h'. Bob also computes $s^e \equiv h^{fe} \equiv h$ modulo N. If Bob finds that h and h' are equal he knows that Alice sent the message. I.e. the message has been authenticated.

   Discuss the RSA signature protocol. Implement the protocol and test your code by implementing one or more examples.

3. **(core)** Apply the RSA related functions defined in lectures to encrypt and decrypt a message consisting of a long string—for example hundreds or thousands—of characters. Do this by slicing the original message in to strings of an appropriate length, encrypt each string, and enumerate these in a list to be transmitted. (Use 512-bit primes p and q for the encryption.) Then decrypt the transmitted list. Implement and test, using a message of at least 2000 characters, functions to do this. You might want to define your own functions for encoding and decoding strings of characters as integers. Discuss the security implications of this way of proceeding and investigate other possible ways of dealing with longer messages.

4. **(core)** For the *RSA* protocol we used 512-bit (i.e. 154 digits in decimal) primes p and q. The security of the protocol relies on the fact that that it is VERY HARD to recover p and q—i.e. to factorise N—from $N = p \cdot q$ if you only know N. To test that this is indeed the case try out the `decompose` function from lectures. To do this generate primes p, q and input $N = p \cdot q$ to the `decompose` function. Starting for example with $l = 30$ bit primes (e.g. the integer 1 billion is 30-bit) write an algorithm that shows the average computation time on input $N = p \cdot q$ for k-bit primes p, q for $k = l, l+1, l+2, \ldots$ Conjecture at what bit length the use of `decompose` becomes unfeasible.

5. **(core)** One way of factoring large integers is via the *Pollard rho* method. For this method note firstly that, given composite integer N, the smallest prime factor p of N satisfies $p \leq \sqrt{n}$. Then if we take a random (or in practice a *quasi-random*) sequence of integers $m_1, m_2, \ldots, m_k$ with integer $k \simeq 10 \cdot n^{\frac{1}{4}}$ $(\geq 10 \cdot \sqrt{p})$ we find, with high probability, that the m_i are pairwise distinct modulo N. Also—and this is the vital part—we are almost certain to find indices $1 \leq i < j \leq k$ such that $m_i \equiv m_j$ modulo p. This means that $1 < \gcd(m_i - m_j, N) < N$ and so we have found a non trivial factor of N, namely $\gcd(m_i - m_j, N)$. Now, by carefully choosing the way in which we compute a pseudo

random sequence $m_1, m_2, m_3, \ldots$ we are able to apply this method to factorise N in time $O\left(n^{\frac{1}{4}}\right)$.

Investigate and fully describe the Pollard rho method. Write an algorithm/function that implements this method. Test your algorithm on k-bit integers for $k = l, l+1, l+2, \ldots$ with l chosen appropriately and compare the results to the use of `decompose` above. Conjecture at what bit length the use of your algorithm becomes unfeasible.

Further Directions. The following are suggestions of further material that you may include in the project. (a) In section 1 consider a further basic cipher, for example the *Affine* cipher. (b) Describe and implement other methods for factoring large integers. (c) Give a detailed overview of the time complexity and security aspects of the methods considered in sections 1-5. (d) Investigate the *Diffie-Hellman* protocol in the manner outlined in sections 6-7 below.

6. If Alice and Bob are able to securely encrypt messages using a shared key then they can safely transmit these messages provided that they know the key and NOBODY ELSE does. The *Diffie-Hellman* protocol allows Alice and Bob to share an encryption key in this way. Namely Alice or Bob publishes a large prime p—here we can let p be a 256-bit prime—and a *primitive root* g modulo p (meaning that $g^{p-1} \equiv 1 \pmod{p}$ but $g^t \not\equiv 1 \pmod{p}$ for all $0 < t < p-1$). Alice and Bob choose large (again 256-bit is appropriate here) integers a and b. Alice computes $A = g^a \pmod{p}$ and Bob computes $B = g^b \pmod{p}$. Then Alice sends A to Bob and Bob sends B to Alice. Now Alice computes $B^a = g^{ba}$ whereas Bob computes $A^b = g^{ab} = b^{ba}$. Hence they are now both in possession of key $K = B^a = A^b$.

   Discuss and implement functions to carry out this protocol. Implement an example of this protocol being used in conjunction with the Vigenère cipher. (I.e. Alice and Bob share a secret key for the Vigenère cipher using Diffie-Hellman, and then one party encrypts a message and the other decrypts it.) Adapt your example to work with three parties. *Hint:* the secret key will be of the form g^{abc}.

7. The security of the *Diffie-Hellman* protocol depends on the fact that, given p, g and $A = g^a \pmod{p}$, it is not feasible to find a—otherwise, if we intercept B, we could compute the key $K = B^a$. But finding a is precisely the *discrete logarithm problem* and the point here is that, for large a and p it is VERY HARD to find a. To get an insight into this problem proceed in a similar manner to your investigation of the RSA protocol. Namely, you should begin by implementing and testing a *brute force* algorithm that, on input p, g and $M \equiv g^a \pmod{p}$ outputs a, using *small* prime p and integer a. (E.g. use as base g in $\{2, 3, \ldots, 99\}$ and start testing with 30-bit p and a.) Conjecture at what bit length your brute force method becomes unfeasible. Then go on to discuss and implement an algorithm for solving this—i.e. the discrete logarithm problem—in a more time efficient way. For example you might implement the *Baby step/Giant step* algorithm. Test your algorithm on k-bit integers for $k = l, l+1, l+2, \ldots$ with l chosen appropriately and compare these results to the brute force method above.

References

[1] J.A. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996. (Also available at http://cacr.uwaterloo.ca/hac .)

[2] A.L. Sweigart. *Cracking Codes with Python.* No Starch Press, 2018.

[3] M.H. Weissman. Python for number theory. http://illustratedtheoryofnumbers.com/prog.html.

6 Aliquot sequences

For a positive integer n the sum of proper divisors function is

$$s(n) = \sum_{\{d|n, 0<d<n\}} d.$$

It gives the sum of all positive divisors of n, excluding n. Interest in this function goes back to the Pythagoreans (6th century B.C.E.). Aliquot sequences are the seqeunces formed by repeatedly applying this function.

Definition. *For each $n \in \mathbb{N}$ the sequence*

$$A_n = \{n, s(n), s^2(n), s^3(n), \ldots\}$$

generated by applying the function s repeatedly is the aliquot sequence starting at n. If $s^j(n) = 0$ for some j, the sequence terminates after j.

There are some interesting seqeunces. For example, $s(220) = 284$ and $s(284) = 220$. This means $\{220, 284, 220, 284, 220, \ldots\}$ is an infinitely looping aliquot sequence. These are called 'amicable' numbers. Other examples arise from 'perfect' numbers such as 6 which has the property that $s(6) = 6$. There are also sequences which terminate at 0, such as $\{7, 1, 0\}$. The same should happen starting at any prime.

Surprisingly little is known about aliquot sequences. It is easy to see that there are three possible types of aliquot sequences:

1. Those which terminate at zero.
2. Those which enter a loop.
3. Those which continue infinitely but do not contain repeats.

It is not currently known whether there are any of type 3 but equally it is possible that most aliquot sequences are of this type. The goal of this project is explore these sequences computationally.

One of the difficulties is that calculating $s(n)$ becomes computationally difficult once n is large because it involves factoring n into primes. Your project should consist of two main parts:

- Write code to compute some aliquot sequences.
- Use this code to explore questions about these sequences.

The efficiency of your code will determine how effectively you can investigate the sequences.
To do:

1. (**core**) Write a function to calculate $s(n)$. (Note: In week 8 we'll see a method that exploits the fact that the sum of **all** divisors is a multiplicative function.)
2. (**core**) For a given n, compute the aliquot sequence starting at n (up to a sensible point).

   Practical suggestion: Write your code so that it computes at most the first n terms of the sequence. Also, write it so that for some i, your code stops computing new terms once $s^j(n) > i$. At first, you can use lower values but you should aim to get code that runs in a reasonable time for $n = 30$ and $i = 10^9$.
3. (**core**) Find a way to detect loops. (There's a little bit to this – look at the sequence starting at 562 to see why.)

4. (**core**) For each $k < 20000$ try to classify it according to the end state of the aliquot sequence starting at k. It should either terminate at zero, enter a loop or be unknown (you might want to distinguish between the cases where the calculation was cut short because you reached term n and those where the sequence exceeded i).

5. (**core**) If your code takes too long to do (4) for this range for k, revisit steps (1) and (2).

Here are some questions about aliquot sequences to explore:

1. (**core**) How long can the loop be if an aliquot sequence ends in a loop?

2. (**core**) If a sequence terminates at zero, how long can it be (relative to the starting n)?

3. Find a nice way of plotting to visualise aliquot sequences which enter loops.

4. Related to perfect numbers are 'abundant' numbers with $s(n) > n$ and 'deficient' numbers, where $s(n) < n$. A looping aliquot sequence should contain some of each. Compare the number of each up to a fixed value n.

5. For a fixed n, how large can the preimage $s^{-1}(\{n\})$ be? This should tell you something about when aliquot sequences starting at different values merge together.

6. Can you make your code faster by precomputing factorisations for small integers (e.g. for all integers up to 10^6)? Investigate using the sieve of Eratosthenes to do this efficiently.

7 Continued fractions, 2-bridge links and d-invariant

Continued fractions

Let $\frac{p}{q}$ be a rational number. A continued fraction for $\frac{p}{q}$ is a sequence $[a_1, a_2, \dots a_n]$ where each $a_i \in \mathbb{Z}$ so that

$$\frac{p}{q} = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \frac{1}{a_n}}}}.$$

You might also see continued fractions for any real number. Rational numbers have finite continued fraction expansions. Continued fractions can be calculated recursively using the quotients you get in Euclid's algorithm. They are not unique, for example

$$[1,3,2] = 1 + \cfrac{1}{3 + \frac{1}{2}} = \frac{9}{7} = 2 + \cfrac{1}{-2 + \cfrac{1}{2 + \frac{1}{-3}}} = [2,-2,2,-3].$$

To do:

1. (**core**) Write at least two functions which calculate a continued fraction for a given $\frac{p}{q}$ (to generate different expansions).

2. (**core**) Write a function that recalculates the rational number from a list $[a_1, \dots a_n]$.

2-bridge links

A *knot or link* is a embedded loop (for a knot) or several loops (for a link) in 3-dimensional space. Two knots are equivalent if one can be deformed into the other without passing through itself. It is usually best to visualise them using knot diagrams. The diagrams in Figure 1 are called 2-bridge because the pictures each have two local maxima.

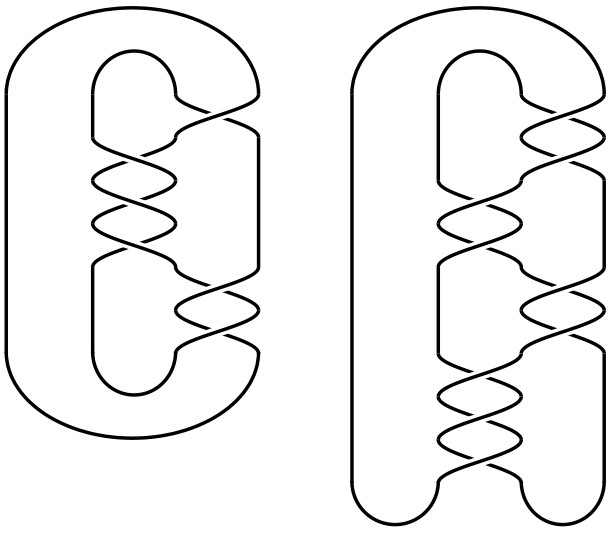

Figure 1: The 2-bridge diagrams corresponding to $[1,3,2]$ and $[2,-2,2,-3]$. They are different pictures of the same knot.

For every rational number $\frac{p}{q}$ there is an associated 2-bridge link $S(p,q)$. Here's the algorithm for drawing it: Start with four parallel strands. Then take a continued fraction expansion for $\frac{p}{q}$. For each a_i add twists between two strands with a_i crossings – use the rightmost two strands if i is odd and the middle two if i is even. The crossings should be right-handed (i.e. right strand over left as you go down) if i is odd and a_i is positive and should change orientation if you switch either of these. Then connect all these twist regions together and connect the top and bottom of the diagram. The pattern for this depends on if you had an odd or even number of twist regions (but the key is that it shouldn't immendiately trivialise the last one). (See Figure 1, [1, Figure 3.9 on p59] or [2, Section 1]. The sign choice varies depending on where you look but the important thing is to pick one and stick with it.)

To do:

1. (**core**) Write code that prints a 2-bridge link diagram for a given $\frac{p}{q}$. It is fine for the diagram to be quite ugly, e.g. built from pieces similar to

```
if crossing_type == 1:
    print("| |  \ / ")
    print("| |   /  ")
    print("| |  / \ ").
```

2. (**core**) Use this function help you check these facts about 2-bridge links for three or four cases:
   (a) $S(p,q)$ is a knot if p is odd and has 2 components if p is even;
   (b) the link $S(p,q)$ does not depend on the choice of continued fraction, even though the diagram does;
   (c) if $n \in \mathbb{Z}$ then $S(p,q) = S(p, q+np)$ (i.e. we only need $q \mod p$);
   (d) changing the sign of all the crossings in $S(p,q)$ gives you $S(p,-q) = S(p,p-q)$.
3. Investigate the relation between the links $S(p,q)$ and $S(p,q')$ when $qq' \equiv 1 \mod p$.

d-invariant

The d invariant of $S(p,q)$ is a collection of p rational numbers[1]. These can be denoted by $d(S(p,q),i)$ and when $p>0$ they are defined recursively by the rules[3, Proposition 4.8][2]:

- For any q and i, $d(S(1,q),i) = 0$;
- If $a\%p$ denotes the integer between 0 and p which is congruent to $a \mod p$ then $d(S(p,q),i) = d(p, q\%p), i\%p)$;
- When $p > q > 0$ and $0 \leq i < p+q$,

$$d(S(p,q),i) = \frac{(2i+1-p-q)^2 - pq}{4pq} - d(S(q,p),i).$$

Since these are rational, you'll need to need ti be careful about rounding errors. For example

$$d(S(9,7)) = [0, \frac{-2}{9}, \frac{4}{9}, 0, \frac{4}{9}, \frac{-2}{9}, 0, \frac{-8}{9}, \frac{-8}{9}].$$

To do:

[1] Technically there is a missing step. First we should use the link to determine a 3-dimensional manifold $L(p,q)$ and then d is an invariant of that manifold.

[2] The orientation convention in that paper is different, hence the different sign.

1. (**core**) Write a function that calulates $d(S(p,q),i)$.

2. (**core**) Write a function that computes the list of d invariants for $S(p,q)$.

3. For examples where you know $S(p,q_1) = S(p,q_2)$ determine how this is reflected in the d-invariant.

   Examine these two questions about the d invariant:

4. (**core**) Does it completely distinguish between 2-bridge links. In other words, is it true that whenever $S(p,q_1)$ and $S(p,q_2)$ are inequivalent links they have differing d invariants? Check this for $p < 1000$.

5. In the case where $p = m^2$ for an odd number m, it is interesting to look at which knots $S(m^2,q)$ have the property that m of the d-invariants are zero. (This is related whether the knot is what is called a slice knot – this is something you can detect from a diagram but it's hard to computationally.) For $m < 105$ determine which of the knots $S(m^2,q)$ have this property and compare the numbers $\frac{m^2}{q}$ to the set $\mathcal{R}$ in [2, Definition 1.1].

References

[1] P. Cromwell, *Knots and Links*, Cambridge University Press (2004)

[2] P. Lisca, *Lens spaces, rational balls and the ribbon conjecture*, Geom. Topol. **11** (2007) 429–472. (Also at https://arxiv.org/abs/math/0701610.)

[3] P. Oszváth and Z. Szabó, *Absolutely graded Floer homologies and intersection forms for four-manifolds with boundary*, Adv. Math. **173** (2003), 179–261. (Also at http://arxiv.org/abs/math/0110170v2.)

8 Bus routes

A bus route in a certain non-UK city has 14 stops, labelled $A0, \ldots, A13$. The distances between neighbouring stops can be found in the attached csv files. Please note that the distance between the same two neighbouring stops may depend on the direction of travel for the bus. A survey counting the number of passengers getting on and off the bus at each stop during each hour can also be found in the attached csv files. A portion of the data looks like the following table:

station	-	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
distance	-	-	1.6	0.5	1	0.73	2.04	1.26	2.29	1	1.2	0.4	1	1.03	0.53
6:00-7:00	on	1990	376	333	256	589	594	315	622	510	176	308	307	68	0
6:00-7:00	off	0	99	105	164	239	588	542	800	407	208	300	288	921	615

Each bus used on this route has a capacity of 100 passengers. But the bus company can supply pushers for any stops, which pushes passengers into the bus when the number of passenger on a bus exceeds 100. These pushers increase the capacity of each bus to 150, but it is undesirable to have more than 120 passenger per bus even with pushers. You should assume that using pushers will lengthen the amount of time the bus waits at each stop. At non-peak hours, the bus company would like the bus to be at least 50% full (i.e. with at least 50 passengers). It is also undesirable to make any passenger wait for more than 10 mintues for a bus to arrive. During peak hours, the waiting time should be less than 5 minutes. You can make any reasonable assumptions on the other parameters, e.g. how long it takes each passenger to get on and off, when these passengers actually arrive at the bus stop, and whether these passenger numbers are taken to be actual or merely a single observation of a random quantity.

You should perform the simulation of bus schedules based on the following steps. The first 2 steps form the core of the project, which you should do to achieve a passing mark.

1. **[Core]** Based on the supplied data, devise a model for the arrival of passengers (e.g. a deterministic arrival process or a Poisson arrival process with non-constant rate of arrival). You may also need to devise a model for how passengers get off the bus. You can employ a relatively crude method to estimate the parameters of your model based on the supplied data. You do not need to employ an optimisation techniques to estimate parameters in your model.
2. **[Core]** Design a bus schedule by hand and simulate the running of this schedule with the model you have devised in the last step. Compute the median and a few important quantiles of the waiting time of passengers. Compute the percentage of passengers that experience being pushed into the bus, and the percentage that experience being a sardine (i.e. in a bus with more than 120 passengers). You can suggest other quantities of interest to compute.
3. Do a simple but informative visualisation of the running of this schedule.
4. Devise a computational method to estimate the parameters in your model. You can use a least squares fit of your model to the supplied passenger data and use optimisation functions in `scipy`.
5. Devise an algorithm to look for an optimal bus schedule. You should impose reasonable constraints on the space of strategies you search in, e.g. the total cost per day (in terms of the total number of bus journeys throughout the day, number of pushers employed) is constrained to be less than a certain budget.
6. Discuss how well your schedule (the one you designed by hand or found by an algorithm) performs when you perturb the underlying model, e.g. what if the number of passengers increases by 20% during a certain hour.

Supplementary information: A note on periodic boundary conditions

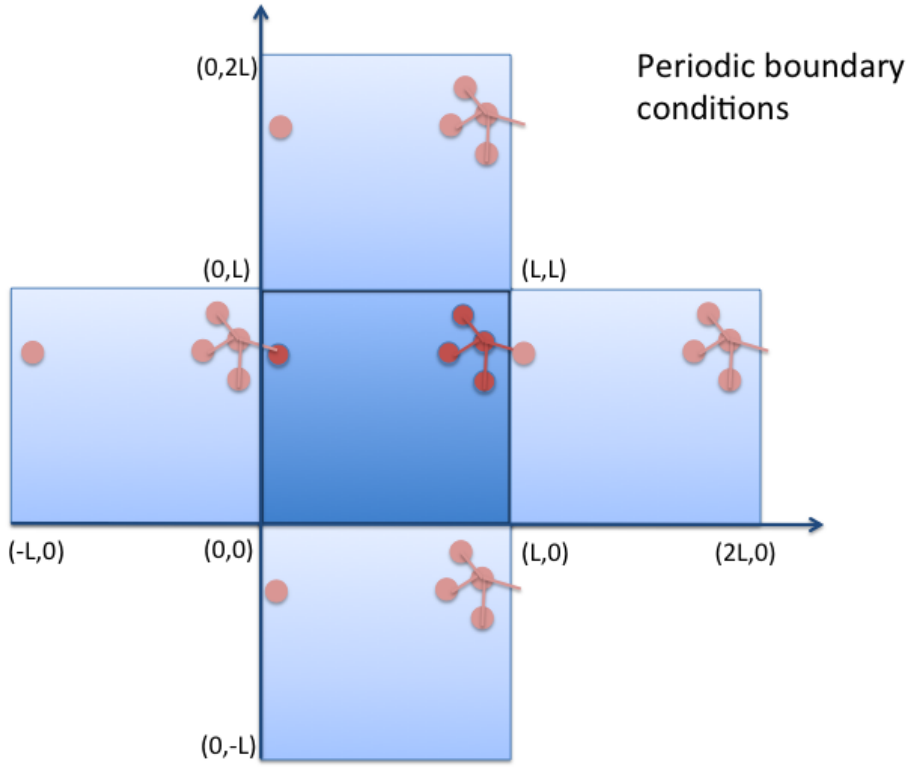

Figure 2: Illustration of periodic boundary conditions. The central square between 0 and L is copied in all directions: moving over the boundary is equivalent to re-entering the square from the other side. The cluster of particles interacts with its nearest neighbours, defined modulo the periodicity.