Scientific Computing, Courant Institute, Fall 2021
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2021/index.html

Third assignment

1. (This exercise gives an example of an unstable algorithm for a well conditioned problem. The algorithm is unstable because it relies on a subproblem that is ill conditioned.)

   The problem comes from computing probabilities related to a simple hopping process. A *hopping process* is a random process in which a particle "hops" between neighboring "sites" at random times. A simple one dimensional hopping process "lives" on sites $\{0, 1, \cdots .n-1\}$ (the integers between 0 and $n-1$, including 0 and $n-1$. The location at time t is $X(t)$, which is a random site. The value of $X(t)$ is one of the integers $0, 1, \cdots, n-1$. We say X *hops* at time t if the value changes at that time. As a mathematical function, $X(t)$ is "piecewise constant", with discontinuities at the time with it hops.

   Suppose $X(t) = k$ with $0 \leq k \leq n-1$. In a time interval from t to $t + dt$, if $k < n-1$, it hops up to $k+1$ with probability $r_u\,dt$. If $k > 0$, the particle hops down to $k-1$ with probability $r_d\,dt$. If $X(t) = n-1$, then it cannot hop up, and if $X(t) = 0$ then it cannot hop down. The *occupation probabilities* are $p_k(t) = \Pr(X(t) = k)$. There is a small probability of having a hop in a small interval of time, but if you neglected it then there would be no hops at all. The probability of more than one hop is even smaller and (take my word for it) may be neglected.

   These probabilities satisfy a system of differential equations derived as follows. We denote conditional probability using the symbol "|", so $\Pr(A \mid B)$ is the probability of A conditional on B. Conditional probability allows to express $p_k(t+dt)$ in terms of $p_k(t)$, $p_{k-1}(t)$, and $p_{k+1}(t)$. If $X(t+dt) = k$, then $X(t) = k$ (most likely), or $X(t = k-1)$ and there was a hop up, or $X(t) = k+1$ and there was a hop down. The derivation neglects the possibility of more than one hop in interval dt. Here is the calculation, which some explanations after:

$$
\begin{aligned}
\Pr(X(t+dt) = k) = {} & \Pr(X(t+dt) = k \mid X(t) = k-1) \cdot \Pr(X(t) = k-1) \\
& + \Pr(X(t+dt) = k \mid X(t) = k+1) \cdot \Pr(X(t) = k+1) \\
& + \Pr(X(t+dt) = k \mid X(t) = k) \qquad \cdot \Pr(X(t) = k) \\
p_k(t+dt) = {} & \Pr(\text{hop up}) \quad\; \cdot p_{k-1}(t) \\
& + \Pr(\text{hop down}) \cdot p_{k+1}(t) \\
& + \Pr(\text{no hop}) \quad\; \cdot p_k(t) \\
p_k(t+dt) = {} & r_u\,dt\,p_{k-1}(t) + r_d\,dt\,p_{k+1}(t) + (1 - r_u\,dt - r_d\,dt)\,p_k(t) \\
p_k(t+dt) = {} & p_k(t) + \left[r_u p_{k-1}(t) + r_d p_{k+1}(t) - (r_u + r_d) p_k(t)\right] dt \;.
\end{aligned}
$$

The basic rule of conditional probability (if you haven't taken a big probability course) is that if A is an "event" ($X(t+dt)=k$ in this case) and B, C, and D are distinct ways A can happen (B is $X(t)=k-1$, C is $X(t)=k$, etc.) then

$$\Pr(A) = \Pr(A \mid B)\cdot\Pr(B) + \Pr(A \mid C)\cdot\Pr(C) + \cdots\ .$$

The first equality in the derivation is this conditional probability formula. The probability of $X=k$ at $t{+}dt$ is the sum of the conditional probabilities multiplying the probabilities for the possible values of X at time t. The second equality says the same thing, using the above terminology and notation. The third inequality comes from substituting in the hopping probabilities. The probability of "no hop" is 1 minus the probability of a hop, which is $1 - r_u\,dt - r_d\,dt$. The notation is r_u for the rate to jump *up* and r_d for the rate to jump *down*. The code `MatrixExponential.py` uses $r_l = r_u + r_d$ for the "loss rate", which is the rate to jump out of site k. The corresponding probability to jump out of site k is $r_l\,dt$. The probability not to jump out is $1 - r_l\,dt$.

These formulas have to be modified if $k=0$ (no down hops) or $k=n-1$ (no up hops). The modified formulas are

$$\begin{aligned} p_0(t+dt) &= \quad p_0(t) \quad + [r_d\,p_1(t) - r_u\,p_0(t)]\,dt \\ p_{n-1}(t+dt) &= p_{n-1}(t) + [r_u\,p_{n-2}(t) - r_d\,p_{n-1}(t)]\,dt\ . \end{aligned}$$

These relations may re-arranged and expressed in traditional calculus notation as

$$\begin{aligned} \frac{d}{dt}p_0(t) &= \qquad\qquad - \quad p_0(t)\,r_u \quad + p_1(t)\,r_d \\ \frac{d}{dt}p_k(t) &= p_{k-1}(t)\,r_u r_u - p_k(t)(r_d + r_u) + p_{k+1}(t)\,r_d\ , \quad \text{for } 1 \le k \le n-2 \\ \frac{d}{dt}p_{n-1}(t) &= \ p_{n-2}(t)\,r_u \quad - r_d\,p_{n-1}(t) \end{aligned}$$

This system if differential equations is expressed in matrix/vector form, by tradition, using a row vector (not column vector) for the probabilities $p(t) = (p_0(1),\cdots,p_{n-1}(t))$. The matrix form is the differential equations is

$$\frac{d}{dt}(p_0(1),\cdots,p_{n-1}(t)) = (p_0(1),\cdots,p_{n-1}(t)) \begin{pmatrix} -r_u & r_d & 0 & \cdots & 0 \\ r_u & -(r_u+r_d) & r_d & & \vdots \\ 0 & r_u & \ddots & \ddots & \\ \vdots & & \ddots & & r_d \\ 0 & \cdots & & r_u & -r_d \end{pmatrix}$$

In matrix/vector form, this is

$$\frac{d}{dt}p(t) = p(t)\,L\ .$$

The matrix L is the *generator* of the random hopping process. You can see that it is *tri-diagonal*, with non-zero elements only on the "main diagonal" and the nearest "off diagonals".

(a) A *diagonal scaling* (more properly, diagonal *re*-scaling) is

$$\widetilde{L} = W^{-1}LW\ ,\quad W = \mathrm{diag}(1, w_2, \cdots, w_{n-1})\ .$$

Show that if W is non-singular, then the eigenvalues of L and $\widetilde{L}$ are the same. Find W so that $\widetilde{L}$ is symmetric. Conclude that the eigenvalues of L are real. Show that right eigenvectors of L are not left eigenvectors. *Hint for the last.* If $Lv = \lambda v$, then $WW^{-1}LWW^{-1}v = \lambda v$ so $\widetilde{L}\widetilde{v} = \lambda\widetilde{v}$, with suitable $\widetilde{v}$. [Not to hand in: any *sign symmetric* tridiagonal matrix (you supply the definition, allow for zeros on the off diagonal if you want) is similar to a symmetric tridiagonal matrix in this way. In differential equations, a *Sturm Liouville* operator (second order differential operator in one variable) is similar to a self-adjoint differential operator, using a diagonal "weighting function". Tri-diagonal matrices may be thought of as a discrete analogue of one-variable second order differential operators.]

(b) The code `MatrixExponential.py` implements three methods for solving the matrix differential equations $\frac{d}{dt}p = pL$ using the fundamental solution and matrix exponential. See `MatrixExponential.pdf` for more on this and a description of the methods. Experiment with the code on a variety of problems (change the dimension, the final time, how different the hopping rates are) to get a feel for which methods give accurate results for which problems. Look for problems that are not extreme that make the eigenvalue method look bad, and problems that make the matrix exponential method look bad. Note that you don't change the problem (or the solution algorithms) if you double the hopping rates and cut the final time in half.

(c) Modify the function `mee(L,t)` to return a tuple (Python term) consisting of the computed matrix exponential and the condition number of the eigenvector matrix R. Modify the function `meT(L,t,n)` to return its computed exponential and the largest norm $\left\|\frac{t^k}{k!}L^k\right\|$. Modify the output part of the main program (lines above 80) to add this information to the printout table. Comment on why/how well/not well this information explains the accuracy/inaccuracy of each method.

2. (This exercise explores linear least squares fitting in a setting where it can be ill conditioned. It takes you through the process of creating and

working with *fake data* to see how well the algorithm works when you know the answer.)

The problem (only slightly idealized from actual chemical estimation problems) involves concentrations $C_i(t)$ that decay exponentially in time because of some chemical reaction. There are m chemical species whose concentrations are decaying. Species i has decay rate r_i, which means that $\frac{d}{dt}C_i(t) = -r_i C_i(t)$. In theory, the total concentration at time t should be

$$f(t) = \sum_{i=1}^{m} C_i(t) \ .$$

The task is to estimate the initial concentrations of the species using only observations of the total, $f(t)$. and the fact that different species have different decay rates: $r_i \neq r_j$ if $i \neq j$. The initial concentrations are $A_i = C_i(0)$.

The approach will be linear least squares fitting. The quantities involved are

- Positive decay rates: $r_1, \cdots, r_m$, assumed known (for this exercise)
- Initial concentrations: $A_1, \cdots, A_m$, to be estimated from data
- Observation times: $0 < t_1 < t_2 < \cdots < t_n$
- Theoretical value at time t:

$$f(t) = \sum_{i=1}^{m} A_i e^{-r_i t} \ .$$

- Observed values F_j, $j = 1, \cdots, n$
- *Residual* (fitting error, statisticians' terminology) $\epsilon_j = F_j - f(t_j)$.
- Sum of squares of residuals

$$R^2 = \sum_{j=1}^{n} \epsilon_j^2 \ .$$

- m = the number of exponentials in the fitting function f
- n = the number of observations

Least squares fitting means finding the A_i to minimize R^2. Experiment with this in the following steps. Write up your procedures and results in a single paragraph or collection of paragraphs. Include printouts of the Python modules you wrote and some important results. You will receive more credit if you summarize your results in a few well formatted tables rather than printing out a lot of numbers that are hard to interpret. Interpreting results and understanding the main lessons of the exercise is an important part of the exercise.

(a) Write a module that contains a function or functions to create *fake data*. The function or functions should return numbers r_i, t_j, and F_j. To do that, they will need to generate fake numbers A_i. The fake observations F_j should be equal to $f(t_j) + \xi_j$, where ξ_j (fake observation errors) independent and are generated using a Gaussian random number generator with a specified standard deviation and mean zero. This code should be given arguments that describe the difficulty of the problem (more on this below). You will do experiments with easy problems to check that the code works and then hard problems to see what can and cannot be learned from observations.

(b) Write code to solve the linear least squares problem. Create a matrix, M and "right hand side" b, in terms of the data r_i, t_j, and F_j, so that the estimates $\widehat{A}_i$ are components of a vector x that solves

$$\min_x \|Mx - b\|_2 \ .$$

Do computational experiments to show that the estimates $\widehat{A}_i$ are close to the true values A_i for easy problems (small m, well separated r_i, lots of observations in a range that is not too small, not too large, and well spaced, small observation noise. Your fake data generator should be able to make data like this, with suitable input parameters. Use one of the solution methods from part (c).

(c) Experiment with three solution algorithms for the linear least squares problem.

   i. Form the normal equations using M^tM and solve using the Cholesky factorization.
   ii. Use the QR decomposition of M.
   iii. Use the SVD of M.

The SVD is so that you can print the condition number

$$\kappa = \sigma_{\max}/\sigma_{\min} \ .$$

All three methods should give (to within roundoff) the same estimates $\widehat{A}_i$ for easy problems. Check this.

(d) Experiment with harder problems. Part of this problem is to see what makes the problem hard. You can measure the difficulty using the condition number computed from the singular values. Try making the r_i closer together, increasing m, clumping the t_j. Comment on the agreement between the different solution methods from part (c) on hard problems. Can you tell which method does better or worse on hard problems?