Physics 512 Final Exam. Due by Monday December 16th

The target length is 3 hours. Please try to not go much over this amount of time. In particular, if you're in the middle of a problem and want to finish it, go ahead, but do not start a new one. You should also feel free to type up answers you have already finished outside of your allotted three hours (though you may also just send in photos of your handwritten work for *e.g.* the proof parts). You may generally use notes/google, but in the unlikely event you stumble across an explicit answer to one of the questions in a dark recess of the internet, you may not copy it. Anything you wish to submit, please put into a single tar/zipfile and email to me (jonathan.sievers@mcgill.ca) with "physics 512 exam" in the subject. Each problem will be worth 25 points.

Problem 1

In this problem, we will work out when it is faster to use Fourier-based techniques in $n-$body simulations instead of directly summing forces between each pair of particles.

A) First, we will work out how long it will take to calculate gravity by Fourier transforming a grid. If we have a 3-dimensional grid with m cells along each side, how many total cells do we have?

B) Roughly how many operations will we need to carry out to do the Fourier transform? I don't care about the coefficient, but I do care about the scaling with m. We will assume (usually justifiably so) that the work for the mesh is dominated by the FFTs. Let the run time equal a times this operation count.

C) If we instead calculate the forces between every pair of particles, how many operations do we need to do for n particles? Again, the scaling with n needs to be correct but don't sweat the coefficient. Let the run time equal b times the operation count (so, if the operation count scaled like n to the first power, our run time would be bn.)

D) Given the scalings from the previous questions, what is the critical value for m at which the grid-based and brute-force techniques take the same run time? In practice, though hard to predict, the coefficients a and b will be of similar magnitude, and easy to measure with quick timing runs. Note - if you get a transcendental equation, feel free to treat the logarithm of m as a constant. In practice, you can pick a starting value for the log, solve for m keeping that fixed, then repeat using your updated values of m in the log. This converges extremely quickly.

E) Interpret your answer to part D) and write down a rule of thumb for when a grid will be faster than brute-force in three dimensions. For instance, one possible answer would be "a grid will be faster when there are many particles in each 3-dimensional grid cell, and slower when the average number of particles per cell is much less than one." This may or may not be correct, but your answer should look something like that. There are many possible answers, but one useful one is to express the average number of particles per cell in terms of the total number of particles plus numerical constants.

Problem 2

We saw in class that the incompressible advection equation

$$\frac{\partial \rho}{\partial t} + u \cdot \nabla \rho = 0$$

became unconditionally unstable when we used a first-order derivative for t and a second-order derivative for ρ.

A) This scheme, the so-called forward time, centered space or FTCS, has

$$\rho(t+\delta t, x) = \rho(t,x) - u\delta t \frac{\rho(t, x+\delta x) - \rho(t, x-\delta x)}{2\delta x}$$

By plugging in complex exponentials, show that this scheme is indeed numerically unstable.

B) We also claimed (and saw in practice) that the *Lax* version of this was stable. The Lax solution uses

$$\frac{\rho(t, x+\delta x) + \rho(t, x-\delta x)}{2}$$

in place of $\rho(t,x)$ in the right-hand side. Derive the stability constraint for this Lax solution.

Problem 3

One way we saw in class to generate weights for arbitrary order numerical integration was to use Legendre polynomials and, after inverting a matrix, take the coefficients that go into the P_0 term. Another way is to fit a normal $(x, x^2, x^3 ...)$ polynomial to data points, then integrate to get the area under each term in the polynomial, and add together to get the integration weights. We will do this by working out the area under a polynomial defined by n points evenly spaced from -1 to 1. In this problem, we will derive the integration weights for arbitrary order using standard polynomial fits.

A) What is the integral of x^k from $x = -1$ to 1, as a function of k? We will refer to this area as a_k. If we have n points, k will go from zero to $n-1$. Explain why we will in general select n to be odd?

B) Show that the least-squares equations reduce to $m = A^{-1}d$ for the special case that A is square and invertible.

C) The area under the curve defined by n points can now be expressed as the sum over the m polynomial coefficients times the area under the k^{th} polynomial, *i.e.*

$$area = \sum a_k m_k = a^T m = a^T \left(A^{-1}d\right)$$

. This is not a convenient way of expressing the area, however. Instead, we can regroup the multiplication and do the A^{-1} and a_k multiplication first. This will give us a set of weights that will then be applied to the data, so our estimate of the area will be $\sum w_i d_i$ with i ranging from 0 to $n-1$. **Write code that does this, and prints out the weights as a function of n.** Since we are integrating from -1 to 1, if the data are all equal to 1, the area should be 2, and hence the sum of the weights as we have defined them so far should be 2. You can use this fact to debug your code (in case, say, you got a transpose wrong).

D) Normally, we define dx to be the spacing between points. In this case, the area under a flat segment of data should be $(n-1)dx$. Adjust (and reprint) your weights so that they are normalized as per usual. If all has gone well, you should get the usual Simpson's method weights (with the dx factored out) of $\frac{1}{3}[1\ 4\ 1]$, and the 5^{th} order weights are $\frac{1}{45}[14\ 64\ 24\ 64\ 14]$.

E) Use these weights to integrate e^x from -1 to 1. Use roughly 30 points for $n = 3, 5, 7, 9$ and print out your error vs. the analytic expectation. Be careful to make sure the actual number of points you use is consistent with the integration order you have chosen (*e.g.* for $n = 5$, you are allowed to use 5,9,14,19... points).

Problem 4

I have hidden an unknown Lorentzian in the file "lorentz_data.txt". Your job is to find it. The standard parameterization is

$$L = \frac{a}{1 + \left(\frac{x-x_0}{w}\right)^2}$$

where a is the amplitude, x_0 is the center, and w is the width. I will tell you the noise is white (Gaussian, with uniform amplitude, uncorrelated from sample to sample) and that the width is at least 10 samples and at most 2000. You can load the data with "y=np.loadtxt('lorentz_data.txt')".

A) Select a range of possible widths that cover the width range in question, and carry out matched filters for each of your widths. For each width, report the maximum amplitude and corresponding x_0.

B) Using your starting guess from part A), write *either* a Newton's method (or Levenberg-Marquardt if you have trouble converging, but you shouldn't need to for a suitably sensible starting guess) or MCMC. Your script should report the best fit parameters and estimates of their errors.