Scientific Computing / Simulation: Open-ended Project list

The final project for this class is worth 40% of your grade. Below are your choices. You only need to do one of them. You can also visit my student recruitment page: http://www.ics.uci.edu/~wayne/research/students. Any project in the Bacterial Tracking, Graph Theory, or Spiral Galaxy projects are allowed, but consult with me first. Note that, since these projects are called "open-ended", and it's worth 40% of your grade, simply doing the bare minimum of what I suggest below is not guaranteed to get you 100%. Be creative. Be inventive. Be scientific. Be curious. Above all, be *interested*. Prove to me you've learned something, and that your answers make sense. Don't try to come back after-the-fact and complain that you didn't get 100% even though you did "everything that was specified". I can guarantee you that some of your classmates will turn in *fantastic* projects that go above-and-beyond. Such students will be appropriately rewarded by getting a better grade than those that hand in a bare minimum project. Note that a "good" project doesn't need to be ***long***. I'm not saying you need to be verbose to do well. Be concise, but complete.

Project MC: Monte Carlo *vs.* Deterministic Volume Integration

You may want to read Chapter 9 of the Bindel + Goodman textbook before attempting this question. Assume you want to estimate the "volume" of a *d*-dimensional (hyper-)sphere of radius *r*=1 centered on the origin. (Note that for d=1, 2, and 3, the common names for *d*-dimensional volume are length, area, and volume, respectively.) So for d=1, the answer is 2; for d=2, the answer is $\pi r^2=\pi$, and for d=3 the answer is $4 \pi r^3/3$, etc. (There are analytical formulae for higher *d* as well but we're going to pretend we don't know them. You are welcome to look them up if you want—see the Wikipedia page on the volume of the *n*-ball), and compare against them, but it's not required.) You can use *d*=1,2,3 as test cases to ensure your program is working.

Recall that in class we assume the existence of a Boolean function $\varphi(x,y)$ that, given a point (x,y) (or in our more general case, a vector in d-dimensional space), returns "true" if and only if (x,y) is inside the volume of interest. In our case, the volume of interest is the d-dimensional sphere, ***but you are not allowed to know that fact outside the function φ***. The goal is to write a generally applicable piece of software that does ***not*** know what φ is, and you are going to *test* your software using a problem with a known answer: the sphere.

We also need to discuss the general terms *precision* and *accuracy*. These two terms have a very different technical meaning. A measurement is *precise* if it is repeatable and has a narrow error bar. A measurement is *accurate* if the *expected* value of the measurement is close to the true value. It is possible to have one but not the other in both cases. For example, if you use a ruler to carefully and repeatedly measure the height of a textbook, then your measurement will probably have a precision of about a millimeter. However, if you *think* that the ruler is marked in inches but instead it's marked in centimeters (and you don't notice that fact), then your measurement will be precise but not accurate—you will report inches when you're actually reading centimeters. The number will be repeatable with small variance, but wrong: that is, it will be precise, but not accurate. On the other hand, if the ruler is flexible, stretchy, and made of the same material as a balloon, then your measurement will likely not be very precise: the numbers will be quite different each time you take the measurement. However, the expected value of your measurement, across many attempts, may be accurate, if the numbers marked on the rubber are correct and you know the units it uses.

If your function returns error bars, then the width of the error bars are a measure of your *precision*, but not your accuracy. You have no idea of the *accuracy* unless there exists an external, pre-determined, high accuracy value that you can compare against. In our case we want to test the *accuracy* of our software, so we're going to run it on a problem with a known answer. If the number you get (say, near the middle of your error bar) is actually close to the correct answer, then your answer is *accurate* regardless of its precision. Our goal, of course, is to have a measurement that is *both* precise and accurate, although it can be difficult to have both.

Your job is to do a detailed comparison of both the precision and accuracy of your software between two different methods for estimating volume as a function of *d*. In both cases your *aim* is to get the answer to be correct (ie., both precision and accuracy) to 4 digits. This means that your *precision* (ie., width of the error bars) should be 4 digits, and *also* the answer should be *accurate* to 4 digits. (If you have a bug in your program that means you are always off by some constant factor, you may have a precise but incorrect measurement. Alternately, if your answer changes a lot between calls to your program but the average is close to correct, you have accuracy but not precision. You want both.) Yet another way to state this is that your software-computed bounds are the precision; whether you actually get the *correct* answer is your accuracy. You should be able to write code that always gives a *precise* answer or fails ***gracefully*** if it cannot provide the requested precision. You will test *accuracy* by testing on the sphere, which has a known answer.

You are going to do a detailed comparison between two different methods for estimating the answer as a function of *d*. In both cases you want the answer to be correct to 4 digits with 99% confidence. The two methods are described below:

> **Monte Carlo integration**: Surround the hypersphere with a hypercube centered at the origin with sides of length 2; this is the smallest hypercube that can enclose the hypersphere. Clearly, the hypercube has volume 2^d. Select *N* points uniformly at random inside the hypercube, and count how many of them are within distance 1 of the origin (ie., inside the hypersphere). You are allowed to use a different *N* for each value of *d*, but for any given value of *d* you should use some constant value N_d. For each value of *d*, use multiple runs and confidence intervals to attempt 4 digits of accuracy with 99% confidence. It would be easiest for you if you figured out some way to automatically determine when your confidence interval satisfies the criterion, since you'll be doing this for many values of *d*. (If 4 digits, 99% confidence gets too CPU intensive, you can reduce precision to 3 digits, or confidence to 90%, but not both.)
>
> **Cube-based integration:** Divide each of the *d* dimensions into *K* segments, so that the hypercube is dissected into K^d "small" hypercubes. Clearly, each of the small hypercubes *h* has volume $(2/K)^d$. For each small hypercube *h*, you should be able to classify it into one of three categories: either it is wholly inside the hypersphere, wholly outside the hypersphere, or else the surface of the hypersphere passes through *h*. This should allow you to put strict upper and lower bounds on the volume of the hypersphere—you don't even need statistics or confidence intervals here, the bounds are *strict*. For each value of *d*, determine how large K must be (or equivalently, how small the small hypercubes *h* need to be) for you to *guarantee* that you have computed the volume of the hypersphere to 4 digits of precision. (Note that I'm suggesting that you simply write a program to dynamically determine *K* given *d*, I'm not asking for a closed-form solution.) Note that you are ***not*** allowed to use the fact that the volume is a hyper-sphere; you should treat the three-class classification question (inside, outside, intersects surface) as a black box where the only question you can ask at each point is "what class is this box?" However, your bounds must be strict (modulo round-off error, which you don't have to account for). The only way to do this is for a general volume is to sample *every* small hypercube *h*.

In both cases you want to provide error bounds (ie. precision) to 4 digits. Your bound will be *strict* in the case of cube-based (because the answer is deterministic and you know exactly what the upper and lower bounds are), and probabilistic (ie., based on standard deviation) in the case of Monte Carlo.

Part 1: push each of your methods as far as you can in terms of *d*. Which method is more efficient as *d* becomes large, given that we insist on 4 digits of precision in the answer? How far can you push *d* for each method? Plot figures showing run times and accuracies as a function of *d*. How do things change if we demand 8 digits of precision with 99.9% confidence? Do you think it's even *possible* to provide 8 digits of precision? Is the answer different for the two methods? Justify your answer either with an explicit attempt at it, or with an argument about how runtime scales with *d* and the requested precision (ie., size of error bars / stdDev). In all cases, the cube-based method should *guarantee* the specified accuracy (modulo round-off); the confidence only applies to the MC method.

Part 2: A completely different way to look at this is to choose a fixed, relatively large value of *N* for the number of Monte Carlo samples. Let's use 1 million ($N=10^6$), because it's going to be the same for every value of *d*. Then, for the deterministic (cube-based) method, pick *K* so that the number of small hypercubes *h* is also about *N*. (So, K rounded to the nearest integer to $N^{(1/d)}$.) Our goal here is to construct things so that both methods are about the same CPU cost. Then, show the *difference* between the two values as a function of *d*. For this part of the question, you can assume that the volume of the hypersphere in the deterministic case is the sum of those *h*'s that are inside the hypersphere plus half of those that are bisected by its boundary (is there a more clever way?). Based on Part 1, which do you think is giving the more accurate answer? (Feel free to compare to the exact values, which can be found on the Wikipedia page for the volume of an *n*-ball.)

Part 3: If I give you a fixed value of N (number of samples, say 10^6 or 10^9), what is the most accurate value (including minimizing the size of the uncertainty interval) that you can compute for π? In other words, which method, and what value (or values) of *d* should you use to compute π as accurately as possible if you have only a specified number of samples at your disposal, and you don't know N in advance? Think about how you could use variance reduction techniques to increase the accuracy of your answer. (In this part, and only this part, you *are* allowed to know you're working on sphere.)

NOTE: your answer will be judged at least partly based on how far you can push *d*, how accurate your answers are, and how fast your code runs. That is, you *will* be judged at least partly on efficiency. In particular, efficiency will count as 10% of the grade for this project, and students will be ranked based on how far they can push *d*, and those that push *d* farther will get more marks for efficiency. Feel free to compete with each other publicly on Piazza by bragging how far you've pushed *d* and in what amount of CPU time (anonymously if you wish). Same goes for Part 3: for example who can *consistently* get the best value of π if you're allowed no more than 1 million samples? Can you, for example, find a tighter *d*-dimensional bounding shape than a hypercube to surround the hypersphere? This would make for fewer "wasted" samples outside the hypersphere; your tighter shape would still need a closed-form computable volume. (For this part you *are* allowed to assume the volume is a hypersphere.)

Project "Spring": Numerical simulation of the Ideal Spring using Euler, LF, RK4

You may want to read Chapter 8 of the Bindel + Goodman textbook before attempting this question. In this project you'll learn about numerical integration of ordinary differential equations, and how to create a continuous approximation between grid-points. It helps if you like physics, because this project is pretty physics-heavy. Also, for all calculations, all floating point variables must use ***SINGLE*** precision, not double or long double. Single precision (the "float" data type in C or C++) requires 4 bytes of storage and provides about 7 digits (base 10) accuracy. Do *not* use double precision. We *want* to see the errors here and it's easier to see the errors if we use single-precision floating point.

Part 1: understanding *order* of integrators. In class I introduced a simple method of numerical integration, called *Euler's Method*. Given an ordinary differential equation $x'(t) = f(x)$, with the initial conditions $x(0)=x_0$, (where x_0 is a constant), we choose some small time-step Δt and iterate

$$x(t+\Delta t) = x(t) + \Delta t f(x(t)) \qquad (1)$$

or in code,

$$x_{i+1} = x_i + \Delta t * f(x_i) \qquad (1a)$$

To define the *order* of a numerical integration method, assume that we want to perform an integration across a relatively small (but not infinitesimal) duration *H.* If we divide the interval *H* into *n* segments, then the order of the

integration method is *k* if the total error of the numerical method across duration *H* compared to the exact solution across the same interval, scales as $1/n^k$. Put another way, if $\Delta t=H/n$, then a k^{th}-order numerical integration method has *global* error that scales as $(\Delta t)^k$ compared to the exact solution.

Euler is called *first-order* method because it turns out that, compared to the exact solution, as the timestep Δt gets smaller, the solution you get from equation (1) differs from the exact solution, *step-by-step* by about $(\Delta t)^2$, and if you sum up all these errors across the interval *H* then the total is $(\Delta t)^1$ —the power of Δt is the *order* of the method. (Remember that $\Delta t = H/$n, so the total error is the sum of *n* errors each of size $O(1/n^2)$.)

Now, if we're to do any physics, then we want to solve *F=ma*, but *a* is the acceleration, which is the *second* derivative of location. In other words, if *x(t)* is location as a function of time, then *F=mx''*, where *x''(t)* is the second derivative (with respect to time) of *x(t)*. Now, if we let *v(t)* be the velocity (in the *x* direction), then *v(t)=x'(t)*, and we can use Euler's method to solve the second-order differential equation using two first-order equations. (Footnote: don't confuse the order of the *numerical integration method* with the order of the *differential equation*. It's unfortunate that the same terms are used, but the order of the differential equation is simply the highest derivative, eg first, second, or third derivative, etc; the order of the numerical integration method is the exponent of Δt that measures the global error.) Anyway, to solve *F=ma*, we need to add an initial velocity $v(0)=v_0$ to our initial conditions, and then Euler's method becomes:

$$x_{i+1} = x_i + \Delta t * v_i,$$

$$v_{i+1} = v_i + \Delta t * a_i$$

where a_i is usually computed using *F=ma* or *a=F/m*, where the force *F* is computed using some physics. **NOTE:** if a_i is computed using x_i, then be sure to pre-compute a_i at x_i, rather than x_{i+1}. (In other words if you just code x=x+v*dt, then you've updated x before computing a, which is bad. Compute a first.)

Now, finally, we can talk about a second-order method, called *Leapfrog*. It's very simple, and only works for solving second-order differential equations. All we do is compute the new positions at timesteps *i* as above, but the velocities are computed at *half* timesteps. So given an initial position x_0 and initial velocity v_0, you solve *F=ma* to find the acceleration at time 0, and then perform a one-time half-step of *v* as follows:

$$V_{0.5} = v_0 + \tfrac{1}{2}\, \Delta t * a_0$$

Now, you effectively have position at time 0, and velocity at time $\frac{1}{2}\,\Delta t$. Now you just update the position and velocities as previously in full (not half) Δt incements, computing the acceleration at *new* position:

$$\text{Update } x_{i+1} = x_i + \Delta t * v_{0.5+i}$$

$$\text{Compute } a_{i+1} \text{ from } x_{i+1}$$

$$\text{Update } v_{i+1.5} = v_{i+0.5} + \Delta t * a_{i+1}$$

Conceptually, the position and velocities are "leapfrogging" each other, with each being computed halfway between the other, being careful to compute the velocities from the acceleration caused by the position half-way between the old and new velocities. It turns out that if you cut the timestep in half with leapfrog, then the local error goes down by a factor of 4—in other words, the Leapfrog integrator is a *second* order numerical integrator—much better than first-order, even though it looks exactly like the Euler method, and costs exactly the same amount of computation. The only difference between Euler and Leapfrog is the initial section that moves the velocities half a step ahead of the positions, and then you need to be careful to compute the accelerations in the right place.

Finally, there is a very popular 4^{th}-order integrator called the Runge-Kutta method. It's too long to explain here; look it up on Wikipedia, or else go find a canned RK4 integrator in whatever language you want to use. It's also used to integrate first-order differential equations, so you'll need to update the positions and velocities separately, just like with Euler.

YOUR TASK in Part 1: finally, we can define your task in Part 1. Your task is to *demonstrate* the orders of the above numerical integration methods. That is, for Euler, Leapfrog, and RK4, show that they are 1^{st}, 2^{nd}, and 4^{th} order, respectively, solving the Spring Equation, $F=-kx$. That is, the spring equation is $x''(t)=-k * x(t)$. Use $k=x_0=1$, and $v_0=0$, and integrate from 0 to $H=1$. Show that if $\Delta t=H/n$ where n is the number of steps taken to get from $t=0$ to $t=1$, then the total error goes as Δt^k, for k=1, 2, and 4, respectively. (Note that the spring equation has an exact solution; go look it up. Alternately, use the RK4 solution with a very small timestep to estimate the "exact" solution and compare all the others to that one.) Include plots of the errors as a function of timestep, on a log scale, to demonstrate the exponent k.

PART 2: Perform an integration of duration 1,000 (ie., t=0 to $t=10^3$). Try all three integrators and various timesteps. Plot all three of them as a function of time using the timestep that appears best for all three, regardless of CPU time taken. Which one provides the most accurate solution in the least amount of CPU time?

PART 3: At this point you should have convinced yourself that the RK4 integrator is the most accurate since its error goes down as $(\Delta t)^4$, which is pretty cool. Now we're going to show you that's not always the most important thing. Now, using all 3 integrators, try performing integrations of increasing durations 10^j, for j going from 1 to 9 (yes, that's an integration of duration 10^9)—although you may not need to go that far, as follows. You know that the total energy of the system is kinetic + potential, and that it should remain constant. Kinetic is always $\frac{1}{2} mv^2$ (use m=1), and potential for the ideal spring is $\frac{1}{2} kx^2$. Since our simulations have numerical error, they will actually violate the conservation of energy, and the amount by which we violate it is a measure of error even in the absence of knowing the exact solution. Plot the total energy of the system as a function of time for all three integrators for various timesteps. Which one provides the best long-term conservation of energy? Why is this surprising? (You can stop once at least two of the integrators violate conservation of energy by at least 50%.) Use any timesteps you need to make the error large enough to violate energy conservation at the above level, but also use smaller timesteps for comparison.

PART 4: Add continuous interpolation of the position to either the Leapfrog or RK4 method (your choice). Use the 3^{rd} order piecewise interpolation method shown in class, where both the points and their derivative (ie., both position and velocity) are continuous everywhere. Then, plot your resulting curve using two different integration timesteps: one small enough so that the curve is continuous with accuracy close to a pixel; and then a much larger integration timestep but with the curve plotted using the interpolant values rather than gridpoint values. Make the second one have time steps big enough that the two curves show differences visible to the naked eye on your plot. (That is, you want to intentionally use an integration timestep in the second case that the departure of the interpolated curve is visible to the naked eye, compared to the curve drawn using smaller integration timestep.)

Project Distributions: Given some empirical data, enumerate the best-fitting theoretical distribution(s) that underlie that data.

You are given a text file consisting of numbers, one per line. Consider it a vector D of scalar data points. You are to write a program to read those numbers and determine what underlying distribution X best fits the hypothesis that those numbers came from distribution X. You will need to compute the best choice of parameters P for model X (using MLE is

fine) assuming data *D*. For each distribution model X you attempt to fit, print the p-value that the data D came from model X using parameters P. Your output should be sorted best-to-worst, by p-value (larger is better).

Your program should be called "distfit", it *must* run on the openlab Linux machines. (I don't care what language you use—if necessary, include a Makefile to create your executable, but it must work on openlab.) The command-line usage of it must be as follows:

./distfit [datafile]

As with the Unix standard for command-line descriptions, anything in square brackets is optional; if no data file is given, your program should read from the *standard input*; that is, it can receive data via a pipe. (**This feature is useful but optional, since some of you are not Unix literate**).

Some "training" data is provided in the file http://www.ics.uci.edu/~wayne/courses/cs115/distfit-data.zip. You are free to discuss these data on Piazza and your fits to them, but your code may also be tested on data that you have not yet seen.

Your code must support at least the following theoretical distributions:

- The Uniform distribution *u[a,b]* with arbitrary lower and upper bounds *a* and *b*, respectively.
- The Exponential distribution with mean rate *r*. (Careful—rate *r* is the *inverse* of mean-time-between-arrivals.)
- The Normal distribution with mean μ and standard deviation σ.

Your output must consist of the following: for each distribution you attempt to fit, output the MLE estimate(s) of its parameter(s), and the fit values of all statistical tests you perform (K-S, Chi-square, etc), and the *p*-value resulting from each test. **Sort the output distributions best-to-worst**. That is, you are not to tell the user only the best-fit distribution, but tell the user the fit from *all* your implemented distributions, along with your estimate of the *p*-value, with the outputs sorted by *p*-value. If you perform more than one test (K-S, Chi-Square, etc.,), output two different lists, one sorted by each of the tests. (Note that the tests can give slightly different results. This is normal, but there should not be *major* discrepancies; for example if one statistical test gives a p-value of 0.5, and another gives a p-value 10^{-10}, there's probably something wrong.)

In your write-up, provide the output of your code on all the given data sets at the URL above, along with an English explanation of what your program concludes about each.

If you have generated some test cases of your own (this is highly recommended!), include the input data files in your submission, along with your program's conclusions about them in your write-up.

Wherever the *t*-distribution would be most appropriate, you are allowed to assume the Standard Normal distribution instead.

Addendum

Below is a response to a student's question on Piazza. The student asked what the "null" hypothesis should be, and if we are computing *likelihoods*, or *probabilities*. Both are very good questions.

Answer: We are computing the *probability* that the *data* (call it D, just a vector of values) could have come from the proposed *model*. The world has plenty of models to choose from, but in this assignment you're allowed to limit yourself to a small number of them: uniform, exponential, Normal.
For each possible model X, you use the *Maximum Likelihood Estimator* (MLE) procedure to choose the *most likely* set of model parameters, say P, under the assumption that model X generated the given data. ***That is the null hypothesis:*** that the data D came from model X under parameter setting P. Let's call that hypothesis X(P,D). (Note that MLE doesn't tell

you the actual likelihood, it only provides the parameters that are *most likely*--but that's OK, we don't care about the likelihood.)

Finally, once you have the most likely parameters for model X that could have generated data D, you can now compute the actual *probability* that this data came from X(P). You do that by computing the probability that D would be observed, had it actually been drawn from X(P). Every model distribution has an algorithmic way of computing such a probability, so you need to program that algorithm into your code.

Once you've done the above for all models X, you'll get a *probability* that the data D came from model X, for each X. Those are p-values—actual probabilities. The smaller the p-value, the ***less*** likely it is that the data came from model X. The highest p-value is the most likely model. (Note of course that all those p-values won't sum to 1; but for any given model X, the probabilities across all possible D vectors *does* sum to 1.)

Note: in the "real" world, if the above process tells you that even your most likely model has a very low p-value (eg. I'd be concerned if the best model gave a p-value of 1e-3 or less), then it would be time to consider adding more models to your code. Especially if the p-value is ridiculously low, like 1e-10, then it's almost certain that the data came from some underlying model you've not considered.

Project "Shadowing"

Read the following paper: Hayes, Wayne. "Computer simulations, exact trajectories, and the gravitational N-body problem." *American journal of physics* 72, no. 9 (2004): 1251-1257. Then answer all the questions in the "Exercises" section at the end. (Note you need to be on campus in order to download the paper from their website, because it's behind a paywall.)

Other possibilities: talk to me if you want to try one of the things listed below:

- Integrate a PDE, which requires solving a linearized system between timesteps.
- Implement an interval arithmetic package for the 4 basic ops plus sin, cos, and exp().