CS115 – Computer Simulation—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.

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

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.

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 assure 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*.
>
> **Cube-based integration:** Divide each of the *d* dimensions into *K* segments, so that the hypercube is disected 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 roundoff 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*.

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 99.99% confidence? (In all cases, the cube-based method should guarantee the specified accuracy, 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), 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.

NOTE: your answer will be judged at least partly based on how far you can push *d*, and how accurate your answers are. 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

In this project you'll learn about numerical integration of ordinary differential equations. It helps if you like physics, because this project is pretty physics-heavy.

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 ½ Δt. 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 ½ mv^2 (use m=1), and potential for the ideal spring is ½ 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 100%.) Use timesteps at least as small as 0.01, or even smaller if you wish.

Project Aloha: Comparing the Slotted and Unslotted Aloha Protocols

This is an event-driver simulation, in case you're not totally sick of them. But it's not an easy one.

The Aloha protocol is one of the earliest packet transmission protocols for network communication. Here are the elements of the system:

- Assume all packets are the same length, say length 1. (could be KB, MB, whatever.)
- Assume that there are *n* hosts on the network, and everybody knows what *n* is.
- Assume there is only one medium of transmission—it could be one shared wire like the original Ethernet, or it could be Wifi.
- Since there's only one medium, packets can *collide* if two (or more) hosts try to transmit at the same time.
- In real life, packets collide because the bits travel at finite speed—either at the speed of light in the case of Wifi, or a bit slower on fiber or copper wires, but in any case if host A starts to transmit, host B may start at the same time or even a bit later because it hasn't *detected* A's transmission because A's transmission is still in flight towards B. In any case the transmissions are doomed to collide with no way to detect it.
- For simplicity, we are going to ignore propagation delay (ie., packets travel at infinite speed beween hosts), but to *mimic* the problems caused by propagation delay, every host will start transmitting whenever it feels like it even if another packet is in the midst of transmission.
- A host will detect if its packet was involved in a collision by detecting if the medium is *busy* both at the beginning, and at the end, of its packet transmission. (But it will still transmit even if the medium is busy at the beginning of its packet transmission—which approximately models collisions that occur in the real world while allowing our simulation to detect collisions by only checking twice for other packets—at the start and end of its own transmission—this is enough since every packet is of length 1 so if there is a collision we are guaranteed to detect it with only 2 checks for a busy medium.)
- Every packet involved in a collision is destroyed and must be retransmitted.
- Packets arrive globally in the system as a Poisson process at an average rate R, where R is in [0,1]. 1 represents a medium that is 100% busy all the time. (This would only be possible if there were no collisions.) Each packet is assigned to a transmitting host with equal probability. (Equivalently, each of the *n* hosts has packets arrive for transmission as a Poisson process with a rate *R/n*.)
- When a packet is involved in a collision, the host that was trying to transmit the packet retries the same packet again at a later time, after waiting a random amount of time that is exponentially distributed with mean n—ie., with *rate* 1/n. Basically, this means that if every host had a packet to transmit, then on average one host will be trying to retransmit every 1 unit of time (which, recall, is also the length of time to transmit one packet).

Part 1: What is the maximum global arrival rate of packets that this system can sustain? Demonstrate this rate using your simulation, by showing a plot of throughput as a function of R; if the system is below saturation, then the mean throughput should equal R; but if the system is overloaded, then the throughput will "max out" at some point and never get any higher, despite R increasing. Figure out R_{max} using your simulation. Is it independent of n? (Assume n>1.)

Part 2: Is there a better choice than 1/n for the retry rate? In other words, can we get a better R_{max} that what we found in Part 1, using a different retry rate? What's the best value of the retry rate?

Part 3: It turns out that you can do better if you use *Slotted Aloha*. This means that transmissions are only allowed to start at integer values of the time. You're not allowed to start transmitting at just any time; you need to wait until the time of the start of the next slot before you transmit. Collisions can still occur, but the restriction of the start-of-transmission times actually reduces the collisions and increases throughput. Re-do the same analysis in Parts 1 and 2, but for slotted Aloha. In other words, find R_{max} for slotted Aloha, and then find the optimal retry rate that maximizes R_{max} for Slotted Aloha.