Full Name: _____

Andrew ID: _____

15-462/662, Fall 2020

Final Exam

December 20, 2020

Instructions:

- Answers must be submitted by 8:30pm Eastern time on December 20, 2020.
- There is no time limit; you may work on the exam as much as you like up until the deadline.
- **You can skip one full question** (1, 2, 3, or 4) and still receive full credit—but you *must* mark this question on your answer template.
- This exam is **open book, open notes, open internet**, but you must work alone[1].
- Your answers **must** be filled out via the provided plain-text template, and submitted via Autolab. We will not accept any other form of submission (such scans of written exams, email, etc.).
- For answers involving code, your solution should have the same prototype as the function given in the prompt.
- Partial credit will be awarded, so please try to clearly explain what you are doing, especially if you are uncertain about the final answer. Comments in code are especially helpful.

Problem	Your Score	Possible Points
1		25
2		25
3		25
4		25
Total		100

[1] "Alone" means you cannot discuss the exam with your classmates, your friends, your dog, your cat, or any other creature containing deoxyribonucleic acid. Posting or asking questions online is strictly forbidden.

1 Warming up

(25 points)

Winter is here in Pittsburgh, the snow is falling, the wind is blowing…fortunately you have some cheery little 15-462 exam questions to warm you up!

a. (5 points) What is computer graphics? How is your understanding of computer graphics different from your perceptions at the beginning of the semester?

b. (5 points) In class we said that if you know the radiance along every line segment through free space, you know everything about light in a scene. But often radiance gives you much *more* information than you really need. What is the minimal amount of information needed in each of the following scenarios? For each answer you must give:

- the name of a radiometric quantity (e.g., "radiant energy," "radiant flux," etc.),
- what this radiometric quantity means (e.g., "radiant energy per unit time")
- *why* this quantity is appropriate for the given situation

(a) Suppose you want to "bake" the illumination of an object into a texture map. More specifically, you have a perfectly diffuse object, being lit by all sorts of interesting colorful lights, etc. Since evaluating the lighting is super expensive, you just want to precompute, for each texel [2] of a texture map on your model, a color value that can be used to (correctly) render the image in real time—assuming none of the lights or objects change/move. What quantity should be stored in each texel?

(b) Tis the season to be baking. Suppose you want to use Monte Carlo rendering to predict where and how much the surface of your cookies will get brown in the oven. What quantity do you need to measure?

(c) While you're busy baking, you should also take some basic food safety into account. In particular, salmonella bacteria (which often shows up in raw cookie dough) can be killed by bringing it to a temperature of about 167° F for about 10 minutes. What radiometric quantity should you measure, to ensure all those little critters are dead? (You may assume that your cookies are very thin—almost two-dimensional.)

[2] "Texel" is short for "texture pixel," i.e., just one square element of the texture image.

c. (5 points) A wise man once said that every problem in life has a solution. But this wise man was wrong. Which of the following optimization problems have solutions? Why or why not?

Problem A

$$\begin{array}{ll} \min_{x\in\mathbb{R}^n} & \sum_{i=1}^n x_i \\ \text{subject to} & x_i < 0, \quad i = 1, \ldots, n. \end{array}$$

Problem B

$$\min_{x,y\in\mathbb{R}} \quad \frac{1}{1+x^2} + \frac{1}{1+y^2}$$

Problem C

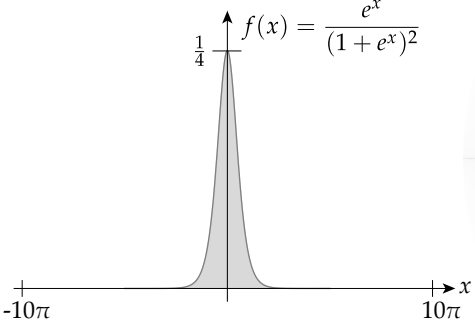
$$\begin{array}{ll} \min_{x,y\in\mathbb{R}} & (x-1)^2 + y^2 \\ \text{subject to} & (x-2)^2 + y^2 \leq 1. \end{array}$$

Problem D

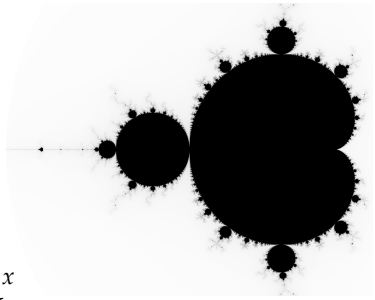
$$\begin{array}{ll} \min_{x,y\in\mathbb{R}} & x^2 + y^2 \\ \text{subject to} & \sin^2(x) < \frac{1}{2} - \cos^2(x). \end{array}$$

d. (5 points) Sometimes rejection is a good thing. For which of the three scenarios depicted below is rejection sampling a better strategy than the inversion method, and why? In particular, suppose we want to (A) pick samples on the interval $[-10\pi, 10\pi]$ proportional to the function $f(x)$, (B) sample points uniformly from the Mandelbrot set, and (C) sample points uniformly from a very small spherical triangle on the unit hemisphere. In each case you should describe your rejection sampling strategy.

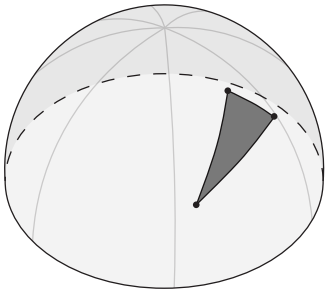
(A) 1D function



(B) Mandelbrot set



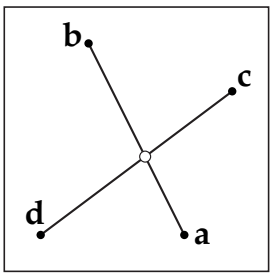
(C) spherical triangle

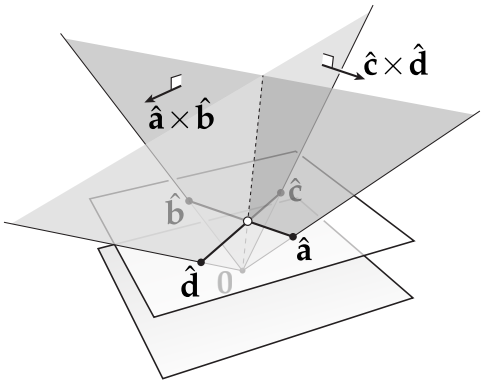


e. (5 points) Suppose you have a line L passing through two distinct points $\mathbf{a}, \mathbf{b} \in \mathbb{R}^2$, and another line M passing through two distinct points $\mathbf{c}, \mathbf{d} \in \mathbb{R}^2$. Let $\hat{\mathbf{x}} := (\mathbf{x}, 1) \in \mathbb{R}^3$ denote the homogeneous coordinates for any point $\mathbf{x} \in \mathbb{R}^2$ obtained by appending a coordinate "1". I claim that the intersection point $\mathbf{p}$ of L and M can be expressed as

$$\hat{\mathbf{p}} = (\hat{\mathbf{a}} \times \hat{\mathbf{b}}) \times (\hat{\mathbf{c}} \times \hat{\mathbf{d}}),$$

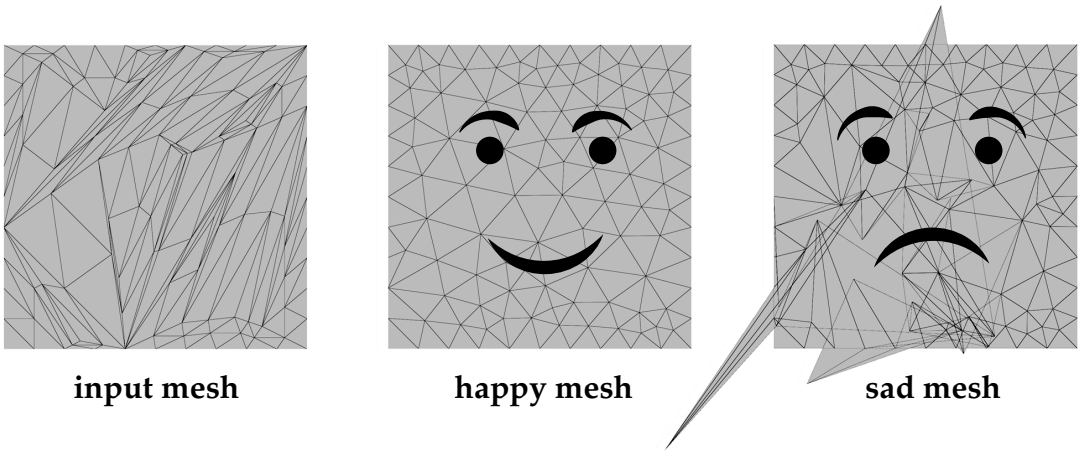
i.e., take the cross product of the two points defining each line, then take the cross product of those two vectors (and then divide by the third coordinate of $\hat{\mathbf{p}}$ to get $\mathbf{p}$). Explain why this formula makes sense, geometrically. What does it mean, geometrically, if $\hat{\mathbf{p}} = 0$?





2 Mesh Optimization

(25 points)



input mesh **happy mesh** **sad mesh**

Optimization shows up all over graphics—one useful thing we can do with optimization is improve the triangle quality of a mesh. There are a lot of fancy ways to do that; here we'll consider a simple, optimization-based strategy that works pretty well.

a. (4 points) The basic idea of our optimization scheme is to imagine that the mesh is a sheet that we want to pull tight—as we pull it tighter and tighter, all the "wrinkles" smooth out, yielding a nicer mesh. In particular, we want to minimize the energy Φ given by

$$\Phi := \tfrac{1}{2} \sum_{ij \in E} |\mathbf{x}_i - \mathbf{x}_j|^2.$$

Here, $\mathbf{x}_i \in \mathbb{R}^2$ is the location of vertex i, and the sum is taken over all edges ij in the mesh.

Question: What is the gradient of the energy Φ with respect to the location of vertex i? (Hint: you should not need fancy mathematical symbols to give an expression for this gradient!)

b. (5 points) Now that we have an expression for the gradient, we should be able to reduce the energy Φ, thereby improving the quality of the mesh. Write some simple [3] code to reduce the energy, using the gradient. You should assume a halfedge mesh data structure—feel free to write code that looks like Scotty3D, or just detailed pseudocode. (Note: it doesn't matter whether you got the answer to the previous part; we'll give you full credit as long as everything else is correct.)

[3] Seriously, hotshot—just write the *simplest* thing that does the job. Don't try to get fancy on me!

```
void decreaseEnergy( HalfedgeMesh& mesh )
{









}
```

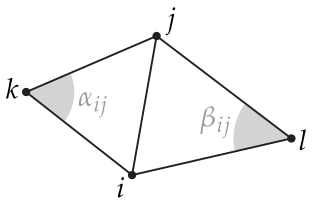
c. (4 points) There's something fundamentally broken about our setup so far: what's the smallest possible energy value for Φ, and what does our mesh look like if we've achieved this value? How can we fix this situation?

d. (4 points) Suppose we modify our code to fix the problem hinted at in the previous question. We then go to run our code... but instead of the **happy mesh** pictured above, we get the **sad mesh** instead. What might have gone wrong, and what's a *simple* way to fix it? [Hint: think about what happened with our pendulum...]

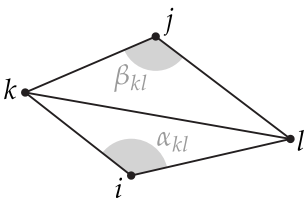
e. (4 points) Finally, optimizing the vertex positions will only get us so far: they can move around a bit, but they're still stuck with the same neighbors. Pictured below is the best we can do by *just* minimizing the energy Φ—not very happy! However, we can do a bit better by performing *Delaunay edge flips* as we optimize. In particular, an edge ij should be flipped if

$$\alpha_{ij} + \beta_{ij} > \pi,$$

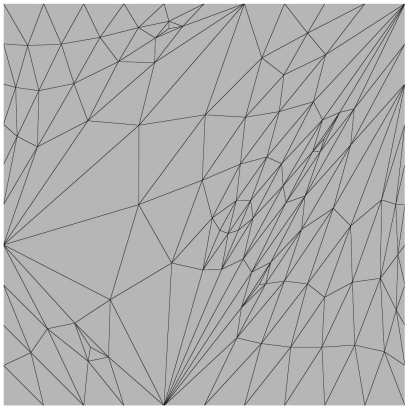
where α_{ij} and β_{ij} are the two angles opposite edge ij (see below). If we keep flipping (and minimizing the energy), we get the happy mesh pictured up top. Write some code that checks whether the given edge needs to be flipped. Boundary edges should not (and cannot!) be flipped.



$\alpha_{ij} + \beta_{ij} < \pi$



$\alpha_{kl} + \beta_{kl} > \pi$

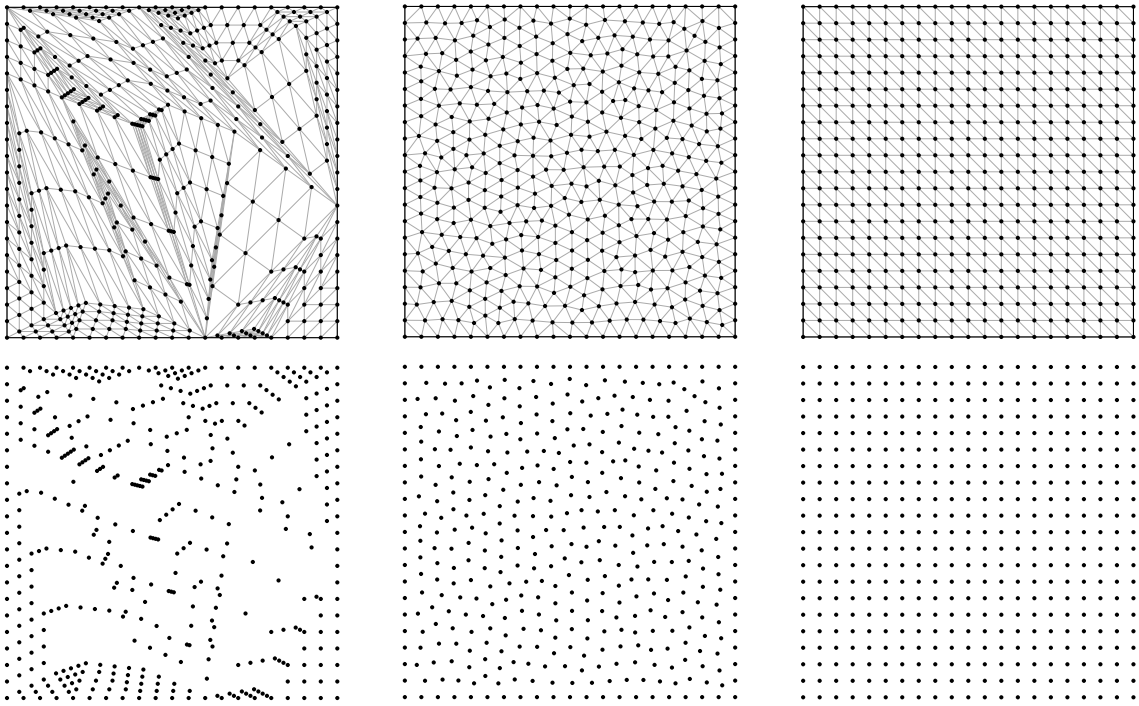


```
bool isDelaunay( Edge ij )
// returns false if this edge needs to be flipped; true otherwise
{




}
```

f. (4 points) Finally, you might be wondering *why* we want to optimize a mesh! One reason is that PDEs solved using the optimized mesh will generally be more accurate and better behaved than on the original mesh. But another, completely different reason to use mesh optimization is to get a nice distribution of sample points that can be used, *e.g.*, to integrate an image-based light source in a photorealistic renderer. For instance, suppose we need to compute an integral $\int_{[0,1]^2} \phi(x,y)\, dx\, dy$ of some function ϕ over the unit square $[0,1]^2$. We can take each of the meshes below, and throw away everything but the vertices to get a set of sample points $\mathbf{p}_i, \ldots, \mathbf{p}_n$. We can then estimate the integral as

$$\int_{[0,1]^2} \phi(x,y)\, dx\, dy = \frac{1}{n} \sum_{i=1}^{n} \phi(\mathbf{p}_i).$$

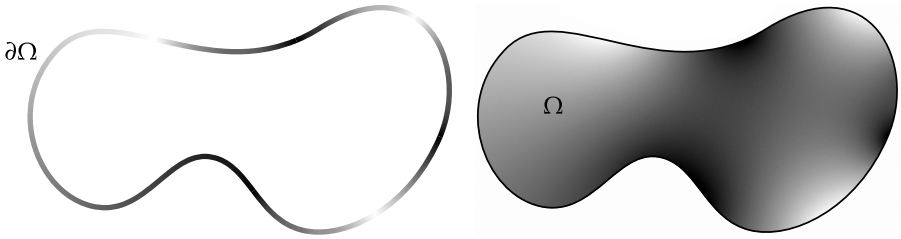
For each of the three meshes, say whether you think it provides a good sample pattern—*and why*—for this task. In each case, describe a function whose integral will *not* be captured well by the given sample pattern.



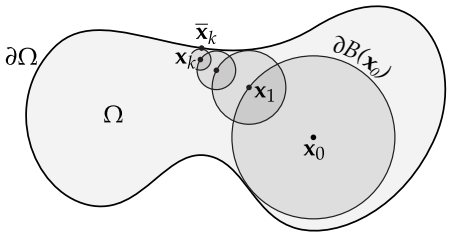
sample pattern A **sample pattern B** **sample pattern C**

3 Walking on Spheres

(25 points)



boundary values $g(\mathbf{x})$ solution $u(\mathbf{x})$



walk on spheres algorithm

In our lecture on PDEs we discussed how heat diffusion, waves, and all sorts of other physical phenomena can be simulated by chopping up space into a grid or mesh. But here's a totally different way to do it, that is much closer to the algorithms we used for solving the rendering equation.

In particular, suppose we want to solve the *Laplace equation* on some domain $\Omega \subset \mathbb{R}^2$:

$$\begin{aligned} \Delta u &= 0 \quad \text{on } \Omega \\ u &= g \quad \text{on } \partial\Omega, \end{aligned} \tag{1}$$

where $g : \partial\Omega \to \mathbb{R}$ is a function we want to interpolate over the domain boundary $\partial\Omega$. For instance, maybe g describes the height of the function u along the boundary, and we want to interpolate these height values smoothly over the interior of the domain.

The solution u to this problem is called a *harmonic function*, and has a very special *mean value property*: the value of $u(\mathbf{x})$ for any point $\mathbf{x} \in \Omega$ is equal to the average value over the boundary $\partial B(\mathbf{x})$ of a ball $B(\mathbf{x}) \subset \Omega$ centered around x and contained inside Ω. We can write the mean value property as an integral

$$u(\mathbf{x}) = \frac{1}{|\partial B(\mathbf{x})|} \int_{\partial B(\mathbf{x})} u(\mathbf{y}) \, d\mathbf{y}, \tag{2}$$

where $|\partial B(\mathbf{x})|$ denotes the area of the sphere.

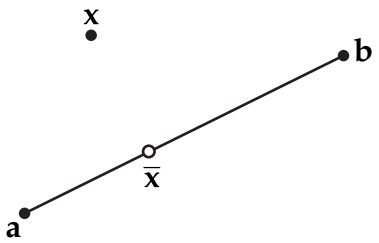
Notice that the value of u at $\mathbf{x}$ depends on the (unknown!) values of u at other points $\mathbf{y}$. So, just like the rendering equation, we will have to solve it *recursively*. In particular, to estimate the value of $u(\mathbf{x}_0)$, we will uniformly sample a single random point $\mathbf{x}_1 \in \partial B(\mathbf{x}_0)$ on the sphere around $\mathbf{x}_0$, then estimate the value of $u(\mathbf{x}_1)$ using the same procedure. Eventually, our point $\mathbf{x}_k$ will get close enough to the domain boundary $\partial\Omega$ that we can just stop and grab the (known) boundary value $g(\bar{x}_k)$ at the closest point $\bar{x}_k \in \partial\Omega$.

This *walk on spheres* algorithm is hence very much like the path tracing algorithm used to solve the rendering equation: we keep "bouncing around" (or in this case, "stepping around") until we hit a light—or in this case, until we can grab a boundary value. Your deep knowledge of rendering can hence be applied directly to solving PDEs—as we will explore below.

a. (3 points) As described above, we can use *any* ball $B(\mathbf{x})$ around $\mathbf{x}$ to estimate the value of $u(\mathbf{x})$. But to make the algorithm efficient, we should take the *biggest* ball, so that we converge to the boundary as quickly as possible.

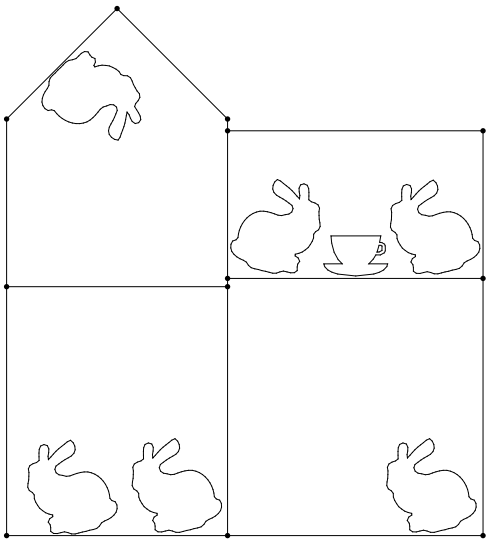
Question: Suppose the domain boundary $\partial\Omega$ is described by a list of 2D line segments. In words (NOT code), what's a high-level strategy for finding the boundary point closest to a given point $\mathbf{x} \in \mathbb{R}^2$? (This strategy does not need to be efficient; it just needs to produce the correct result.)

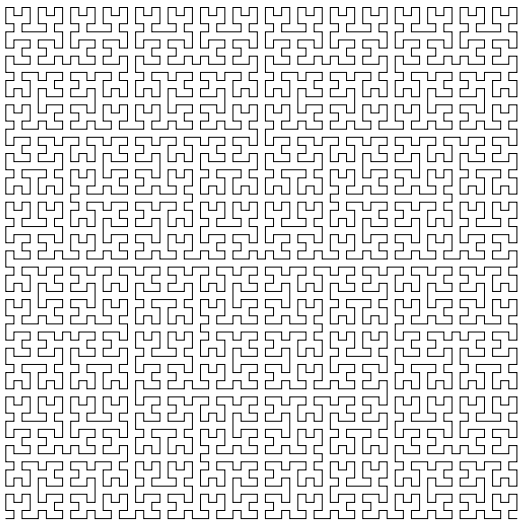
b. (4 points) Any strategy for computing the closest boundary point will need to compute, as a subroutine, the point $\bar{\mathbf{x}}$ closest to a given query point $\mathbf{x} \in \mathbb{R}^2$ along a segment with endpoints $\mathbf{a}, \mathbf{b} \in \mathbb{R}^2$. To evaluate the boundary value $g(\bar{\mathbf{x}})$ at this point, we will also need the barycentric coordinate $t \in [0, 1]$ describing the location of $\bar{\mathbf{x}}$ along the segment (assuming $t = 0$ for $\bar{\mathbf{x}} = a$). Implement a routine that computes these values.




```
void closestPointSegment( Vector2D a, Vector2D b, Vector2D x,
                          Vector2D& xbar, double& t ) {

}
```

c. (4 points) Suppose the boundary is made up of a large number of line segments. In particular, consider two examples: (A) a large dollhouse filled with a bunch of small, highly-tessellated bunnies and (B) a space filling *Hilbert curve*, as depicted above at left and right (respectively). What would be your strategy for reducing the time it takes to perform closest point computations in each case, and why?





d. (3 points) To estimate the solution to the Laplace equation (Equation 1), we have to approximate recursive integral from Equation 2. Consider a simpler, non-recursive integral

$$I := \frac{1}{|\partial B(\mathbf{x})|} \int_{\partial B(\mathbf{x})} \phi(\mathbf{y})\, d\mathbf{y},$$

where $\phi : \Omega \to \mathbb{R}$ is a fixed, known function. One idea for estimating this integral is to evaluate the approximation

$$\hat{I} := \phi(\mathbf{Y}).$$

where $\mathbf{Y}$ is a single random point drawn from a uniform distribution over the sphere $\partial B(x)$.

Question: Is $\hat{I}$ an unbiased estimator for I? Why or why not?

e. (5 points) Ok, let's put it all together. Write some code to estimate the solution to a Laplace equation at the point **x**. You can assume a routine has already been written for you to compute the closest point, with this prototype:

```
getClosestPoint(
   // INPUTS
   const vector<Segment>& boundary, // segments defining the boundary
   const Vector2& x, // query point
   // OUTPUTS
   Vector2& xbar, // closest point
   Segment& S,    // closest segment
    double& t     // barycentric coordinate for xbar along s
);
```

Each 2D line segment is represented by a simple struct that also includes the boundary values at the two segment endpoints; you should assume that these boundary values are linearly interpolated over the rest of the segment:

```
struct Segment
{
   Vector2 a, b; // endpoints
   double ga, gb; // boundary values at endpoints
};
```

Your routine should match the prototype below; the stopping tolerance `epsilon` determines how close you can get the boundary before you just go ahead and grab the boundary value from the closest boundary point, rather than estimating another integral. You can assume that you have a function `unitRand()` that will return a random number sampled uniformly from the unit interval $[0,1] \subset \mathbb{R}$.

```
double estimateLaplace( Vector2 x, vector<Segment>& boundary )
{
   const double epsilon = 0.0001; // stopping tolerance









}
```

f. (3 points) The routine you just implemented, `estimateLaplace`, computes an estimate that is unbiased—however, this estimator has very high variance, leading to a noisy solution. Write a simple wrapper around this routine that computes an estimate with lower variance.

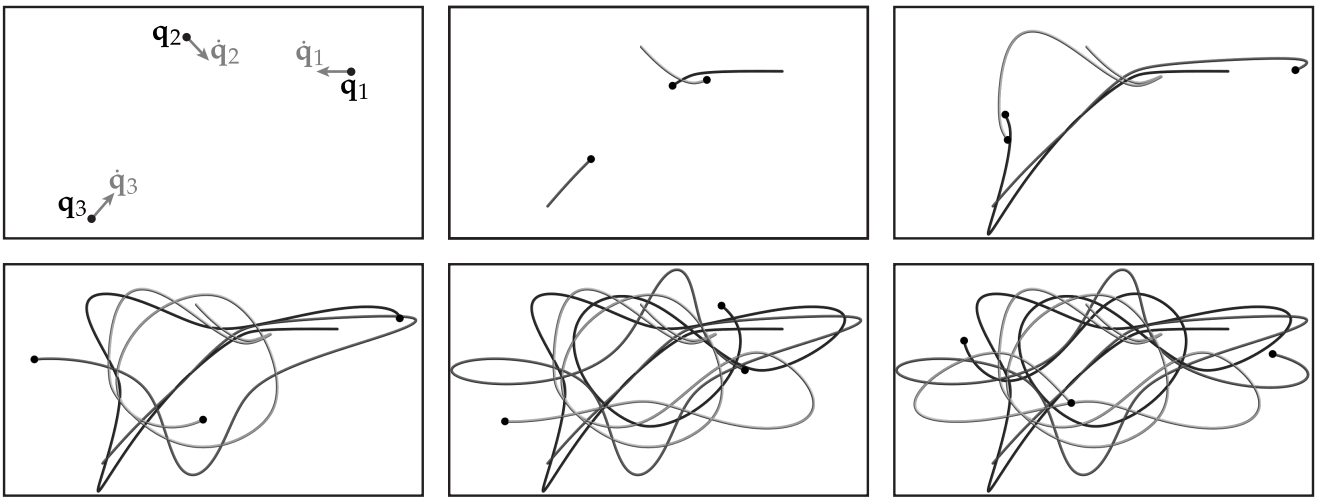
```
double estimateLaplaceLowVariance( Vector2 x, vector<Segment>& boundary )
{









}
```

g. (3 points) A far more standard way to solve a Laplace problem is to discretize the geometry using a triangle mesh, and seek values at each node that are equal to the average of their neighbors—not so different [4] from what you did on the mesh optimization question in this exam! What are some pros and cons of these two approaches, both in terms of computational cost as well as approximation quality? (There are many good answers here—think for instance about our many discussions about sampling and aliasing, or about explicit vs. implicit representations of geometry.)

[4] Except that a "real" Laplace solver will directly solve a system of equations using a linear solver, rather than iteratively adjusting values.

4 When Worlds Collide

(25 points)



As discussed in class, a lot of beautiful complexity can arise from very simple physical simulation. In fact, all the stars, galaxies, and living creatures in our universe essentially started with small particles drifting and spinning toward each other under the force of gravity. How can we simulate these cosmic creations?

A good starting point is to just write down the potential and kinetic energy of n interacting particles; from there we can use Lagrangian mechanics to figure out the equations of motion. Just like the double pendulum, this system will be *chaotic*: even for three particles, the trajectory is wildly unpredictable (as shown above). But we can still use numerical simulation to get a sense of what happens.

We will imagine that our particles are so small relative to the distance between them that we don't care about their particular shape; we just give them each a position $\mathbf{q}_i \in \mathbb{R}^2$ and velocity $\dot{\mathbf{q}}_i \in \mathbb{R}^2$. For simplicity, we'll assume that all particles have mass 1, and will also assume that the gravitational constant is equal to 1. The kinetic energy of our system is then just

$$K = \tfrac{1}{2} \sum_{i=1}^{n} |\dot{\mathbf{q}}_i|^2,$$

i.e., one half mass times velocity squared. The total potential energy is

$$U = -\sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{1}{|\mathbf{q}_i - \mathbf{q}_j|}.$$

Notice that we only sum over pairs $i \neq j$; intuitively, all distinct pairs of particles $i \neq j$ want to move toward each-other.

a. (4 points) Give an expression for $\frac{\partial}{\partial \mathbf{q}_i} U$, i.e., the gradient of the potential energy with respect to the position of particle i.

b. (4 points) Give the Euler-Lagrange equations for the n-body system, i.e., the equations of motion corresponding to the potential U and the kinetic energy K defined above.

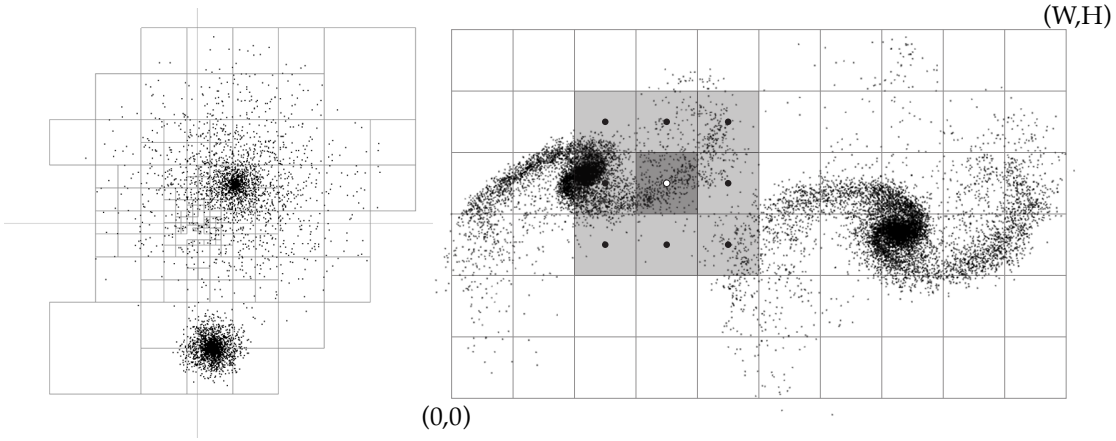
c. (4 points) Let's write a basic numerical integrator for the n-body system. To keep things simple, you should just use forward Euler, applied to a system of two first-order ODEs. You may assume you are given a function

```
vector<Vector2> computeForces( vector<Vector2> positions )
```

that correctly computes the forces $\partial U / \partial \mathbf{q}_i$ (one per particle), whether or not you correctly answered the previous questions.

```
void updateSystem( int n, // number of particles
                   vector<Vector2>& q, // positions
                   vector<Vector2>& p, // velocities
                   double t ) // timestep
// this method should take a single timestep of
// size t, updating the positions q and velocities p
// using the forward Euler method
{









}
```

d. (4 points) Suppose you run your simulation and instead of beautiful swirling motion you just get crazy motion (e.g., the particles immediately shoot off the screen). You check your code carefully, but everything looks perfect. However, you are using some pretty big initial velocities. What could be wrong, and what are two specific things that you could do to improve the behavior of your numerical integrator?



e. (4 points) Gravitational n-body simulations become *way* cooler when n is very large—like a million, or a billion! At this point, you can start to get very convincing simulations of phenomena like galaxies colliding. But the naïve solution of computing all pairwise interaction no longer works, since it takes $O(n^2)$ operations. To speed things up, we can use a strategy we've seen over and over again in graphics: accelerate computation with a spatial data structure. More specifically, we can "lump together" particles that are close to each other in space by (for instance) averaging their position and summing up their masses. Particles outside these groups can then compute their interaction with this "lumped" particle, rather than all the individual particles. Let's consider two specific strategies:

- The classic approach is the hierarchical *Barnes-Hut* algorithm, which uses a spatial tree (like an octree, BVH, etc.) to organize particles. Such a tree can be built in a top-down fashion, e.g., by splitting a node if it doesn't provide a good approximation. (The leftmost picture above shows a quadtree for an n-body simulation.)
- A simpler approach is to use a regular grid, just putting particles into nodes according to their coordinates. A rather crude approximation is for each particle to only consider the forces due to (i) particles in their own cell, and (ii) "lumped" particles in neighboring cells (as indicated in the picture above).

In which scenarios do you think each of these approaches will work well?

f. (5 points) It's the end of a long exam, during a long semester. So, we won't ask you to implement Barnes-Hut! But we will ask you to do a simpler force evaluation using a regular grid. The grid covers a rectangle of width `W` and height `H`, with lower-left corner at the origin (see above).

In particular, given a query point **q**, you should compute the force on a particle centered at **q**, due to all other particles in the system. But for particles outside the cell containing **q**, you should use the lumped approximation, rather than the individual particle centers. You can assume all particles have mass 1. The routine takes two arrays as input, namely:

- `centers` — For each cell `i`, `centers[i]` is a list of the location of all particles in that cell.
- `lumpedCenters` — The value stored in `lumpedCenters[i]` is the average (mean) location of all particles in cell `i`.

Both of these arrays are given in *row major, column minor* order, for a grid with `nx` columns and `ny` rows. Finally, you may assume that you are given a method

```
Vector2 pairwiseForce( Vector2 qi, Vector2 qj )
```

that correctly computes the force of q_j on q_i (regardless of what you answered for part(a)).

```
Vector2 approximateForce(
   Vector2 q, // location of force evaluation
   vector<vector<Vector2>> centers, //centers[i] lists particle centers in cell i
   vector<Vector2> lumpedCenters, //lumped center for cell i
   int nx, int ny, // number of columns and rows (respectively)
   double W, double H // width and height of region covered by the grid
   )
{









}
```