Homework 5: Sampling and Geometry

Introduction to Computer Graphics and Imaging (Summer 2012), Stanford University
Due Monday, August 6, 11:59pm

> You'll notice that this problem set is a few more pages than the usual CS 148 homework; part of this problem set is a reading assignment. The problems are intended to make sure you understand both what we've covered in class and what the narration below discusses: You are responsible for both on the final exam!

In the past few weeks, we have covered a diverse set of topics relevant to computer graphics, from sampling and Fourier theory to computer aided geometric design. In this homework, we will be exploring a number of these topics to help you get better acquainted with the issues we have discussed in class.

We'll start by discussing some of the details of sampling theory laid out in class to help determine the resolution you need to display different continuous images. Recall that we stated the following "theorem" in class:

Theorem(-ish). Any *function can be written as a combination of simple wiggles.*

Of course, the word "any" here is very suspect, but as computer scientists we'll ignore the intricacies of Lebesgue integration and Schwartz distributions in favor of an entirely un-rigorous approach.[1]

Before we define the Fourier transform, we should state a famous mathematical relationship that you likely have seen before:

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{1}$$

This equation often is called "Euler's Formula." Let's fill in some details:

- $e = 2.718\ldots$ is the natural logarithm base.
- $i = \sqrt{-1}$ is the imaginary unit of the complex plane $\mathbb{C}$.
- $\theta \in \mathbb{R}$ is any real number.

If you never have seen (1), you should make the following straightforward observation: *This equation makes no sense*. And, you're not wrong! In particular, raising any number to an imaginary exponent i isn't a reasonable operation in the usual sense of exponentiation as "repeated multiplication."

[1] Please come to Justin's office hours or post on Piazza if you would like work out the details of making this theory more rigorous. Some fascinating and deep mathematics are hiding in the straightforward-looking derivations we cover in CS 148!

In fact, in many ways we can think of Euler's formula as a *notational convenience*. We can prove that the right hand side of (1) behaves identically to exponentiation in almost every possible way. This observation allows us to compactify derivations of many facts we know from high school trigonometry. For instance, consider the following derivation:

$$\begin{aligned}\cos(\theta+\phi)+i\sin(\theta+\phi) &= e^{i(\theta+\phi)} \text{ by (1)}\\ &= e^{i\theta}e^{i\phi} \text{ by properties of exponentiation}\\ &= (\cos\theta+i\sin\theta)(\cos\phi+i\sin\phi) \text{ by (1)}\\ &= (\cos\theta\cos\phi-\sin\theta\sin\phi)+i(\sin\theta\cos\phi+\cos\theta\sin\phi) \text{ since } i^2=-1\end{aligned}$$

Make sure you understand each step here before moving on!

But, what did we just show? If you take the real and imaginary components of the left and right hand sides of the derivation above separately, we find:

$$\begin{aligned}\cos(\theta+\phi) &= \cos\theta\cos\phi-\sin\theta\sin\phi\\ \sin(\theta+\phi) &= \sin\theta\cos\phi+\cos\theta\sin\phi\end{aligned}$$

You probably worked much harder to prove these statements in high school trigonometry.

Anyway, our goal today is to understand the Fourier transform of a function. We now have all the tools we need to do so.

Definition (Fourier Transform)**.** *The Fourier transform of a function $f:\mathbb{R}\to\mathbb{R}$ is the function $\hat{f}(\xi)$ given by*

$$\hat{f}(\xi)=\int_{-\infty}^{\infty}f(x)e^{-2\pi ix\xi}\,dx \tag{2}$$

The variable ξ here represents a frequency. So, $\hat{f}(\xi)$ represents the "component" of f that wiggles at frequency ξ.

Problem 1 (10 points)**.** *Recall that in Lecture 11 of CS 148 we stated that the Fourier transform does the "simplest possible thing:" multiplying a function against a wiggle at a fixed frequency and integrating. Use Euler's formula* (1) *to justify why the Fourier transform* (2) *does exactly this. Why does $\hat{f}(\xi)$ need real and imaginary components?*

As tempting as it is for your mathematician instructor, we will not derive most of the (beautiful) relationships between properties of a function and properties of its Fourier transform. There is, however, one relationship that is fundamental to the sampling theory we discussed in lecture.

Problem 2 (15 points)**.** *Take any real number $a>0$, and define $h(x)=f(ax)$. Derive the following relationship:*

$$\hat{h}(\xi)=\frac{1}{a}\hat{f}\left(\frac{\xi}{a}\right) \tag{3}$$

Explain how this relationship substantiates the claim made in class that increasing the rate at which you sample a (band-limited) function makes it less likely that you will see artifacts.[2]

[2]For students with a strong math background, note that in CS 148 you need not be concerned with whether your integrals converge or are well-defined: feel free to apply and reasonable-looking operation without regard for such analytical details!

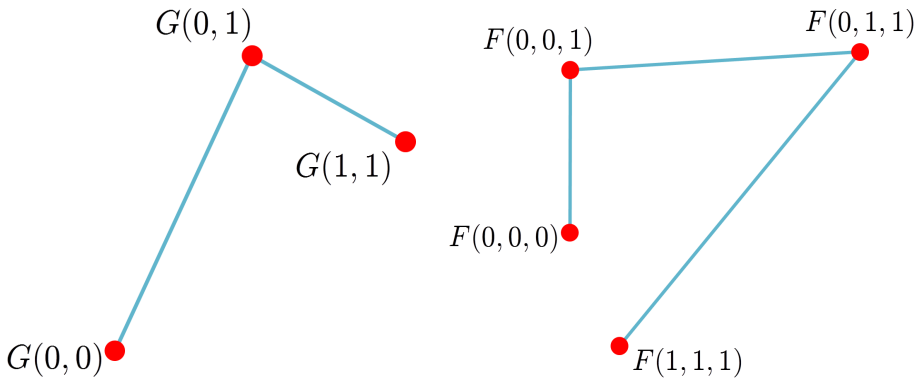

Figure 1: For problem 3(a).

Hint: *To derive* (3)*, substitute h into the definition of the Fourier transform and make the substitution $y = ax$.*

There are myriad other facts we could derive from the Fourier transform $\hat{f}$ that are relevant to sampling theory as it applies to computer graphics, but we'll abandon our discussion at this point in favor of moving to the topic of geometric modeling.

Problem 3 (10 points)**.** *(a) The control points of a quadratic Bézier curve $g(t)$ and a cubic Bézier curve $f(t)$ are shown in Figure 1. Construct the points $g(3/2)$ and $f(1/3)$. You'll need to turn in a copy of Figure 1 with your constructions drawn on top; you may wish to enlarge these structures and/or print them on separate pages for more space. (b) Find the (quadratic and cubic, resp.) blossoms of the functions $g(t) = t^2 - 2t + 1$ and $h(t) = (t-2)^3$.*

One of the key ideas of computer-aided geometric design (CAGD) as it pertains to curves is that of finding a *basis* for polynomial functions of a given degree. As we discussed in lecture, the idea of a basis here is slightly more abstract than what you might have seen in Math 51, so it's worth putting some thought into our construction.

We consider a basis of the set of polynomials of degree n to be a set of $n+1$ polynomial functions $p_1(t), \ldots, p_{n+1}(t)$ such that *any* degree-n polynomial can be written as a linear combination of these p_i's. For example, an obvious basis for the degree-three polynomials would be $p_1(t) = 1, p_2(t) = t, p_3(t) = t^2, p_4(t) = t^3$. Why? Because the general form of a degree-three polynomial is $at^3 + bt^2 + ct + d$, or equivalently $ap_4(t) + bp_3(t) + cp_2(t) + dp_1(t)$.

Certain polynomial bases are convenient for certain problems, so a strategy in CAGD is to work in a basis that simplifies our math as much as possible. We explored this strategy in class when we derived the Hermite basis, which simplified the problem of finding a curve with given begin and end points/tangents. In our final written problem, we will derive an alternative basis for cubic polynomials simplifying math for Bézier curves.

Problem 4 (15 points)**.** *Suppose we label the control points $\vec{a}$, $\vec{b}$, $\vec{c}$, and $\vec{d}$ of a cubic curve in Figure 2 and use the structure drawn to find the point $f(t)$ (assume the control points specify the curve between $f(0)$ and $f(1)$). Label the constructed points as linear combinations of the control points; we provide you with one label to get you started. Use your label for $f(t)$ to define the Bernstein basis for cubic polynomials, that is, a basis that makes it easy to write down the cubic curve associated with a set of four control points.*

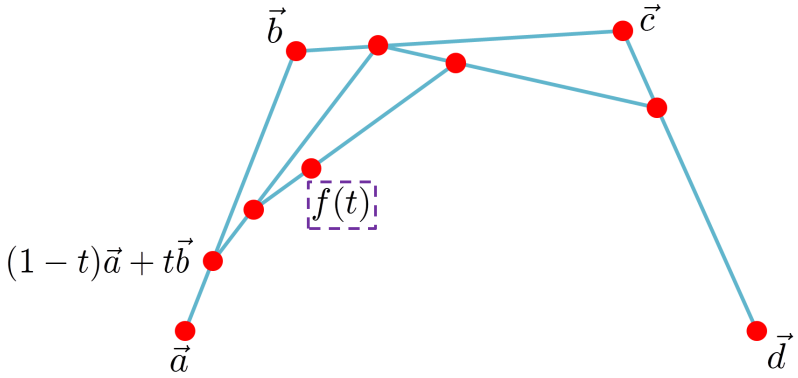

Figure 2: For problem 4.

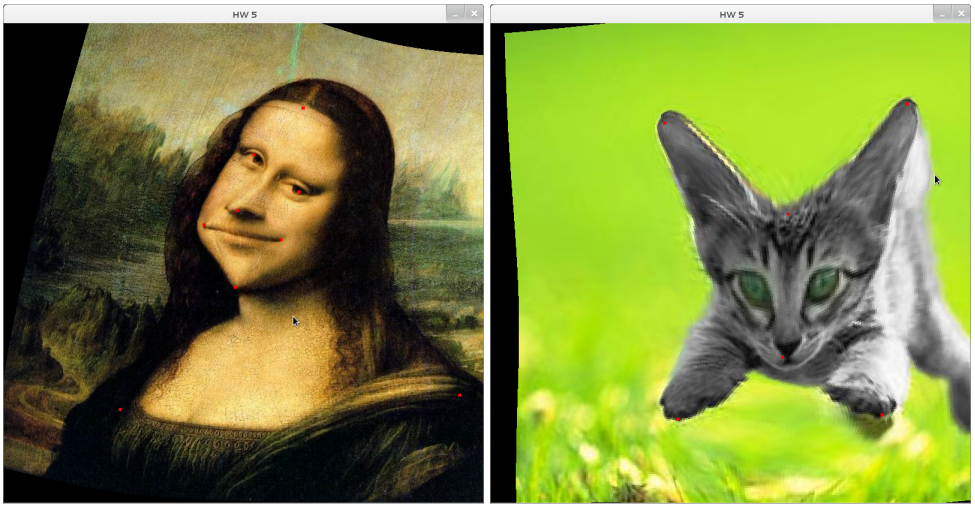

Figure 3: Examples of MLS output.

For the coding part of this assignment, you will implement the *Moving Least Squares* (MLS) image deformation algorithm introduced in SIGGRAPH 2006.[3] Examples of the output of MLS are shown in Figure 3. The purpose of this algorithm is fairly straightforward: The user marks a number of "handles" on an image and then starts moving them around. As the handles are moved, the image deforms smoothly while satisfying the constraints that the handles are mapped to their targets. MLS can be used to achieve lots of interesting image effects with relatively simple computations.

To motivate MLS, we first consider a slightly easier problem. Let's say we mark a number of points $\vec{p}_1, \ldots, \vec{p}_n \in \mathbb{R}^2$ on the plane, and we specify a list of targets $\vec{q}_1, \ldots, \vec{q}_n \in \mathbb{R}^2$. We wish to find an *affine* transformation of the plane given by $\vec{p} \mapsto M\vec{p} + \vec{t}$ for some $M \in \mathbb{R}^{2\times 2}, \vec{t} \in \mathbb{R}^2$ such that $M\vec{p}_i + \vec{t} \approx \vec{q}_i$ for each $i \in \{1, \ldots, n\}$. When is $n > 2$, the $\approx$ is key: it might be the case that *no* affine transformation can map all the $\vec{p}_i$'s to their corresponding $\vec{q}_i$'s exactly!

Thus, we can only hope to approximate such a relationship. To this end, we'll introduce one

[3] You're encouraged to read the original paper for a more detailed discussion. Note that the paper uses row rather than column vectors, so their math is a bit different.

more set of inputs: a set of "weight" values $w_1, \ldots, w_n > 0$. A large weight w_i says we really care that $\vec{p}_i$ maps to $\vec{q}_i$, while a small $w_i \ll 1$ means we don't really care if M and $\vec{t}$ don't do a good job mapping $\vec{p}_i$ to $\vec{q}_i$.

Note that if $M\vec{p}_i + \vec{t} \approx \vec{q}_i$ then $M\vec{p}_i + \vec{t} - \vec{q}_i \approx \vec{0}$. In other words, we desire that $\|M\vec{p}_i + \vec{t} - \vec{q}_i\|^2 \approx 0$. So, let's define an energy:

$$E(M, \vec{t}) = \sum_{i=1}^{n} w_i \|M\vec{p}_i + \vec{t} - \vec{q}_i\|^2 \tag{4}$$

We can make a number of important observations about E:

- We always have $E \geq 0$.
- If there exist $\vec{t}$ and M such that $\vec{q}_i = M\vec{p}_i + \vec{t}$ exactly for *all* $i \in \{1, \ldots, n\}$, then E will be minimized with value 0 at this point.
- When such a zero minimum is not possible, the minimum $E > 0$ occurs when M and $\vec{t}$ give the best affine map approximating $\vec{p}_i \mapsto \vec{q}_i$ in a "least squares" sense. The weights w_i prioritize with $(\vec{p}_i, \vec{q}_i)$ pairs matter the most.

Thus, if the user inputs lots of pairs $(\vec{p}_i, \vec{q}_i)$, we can frame an optimization problem to find M and $\vec{t}$ as follows:

$$\min_{M \in \mathbb{R}^{2\times 2}, \vec{t} \in \mathbb{R}^2} E(M, \vec{t}) \tag{5}$$

You probably spent much of calculus class solving problems exactly of this nature! Simply take derivatives of E with respect to the elements of M and $\vec{t}$, set them to zero, and solve.

We'll resist the urge to force you to derive these formulas yourself, although you're encouraged to check them on scratch paper; our discussion above plus a few tricks from Math 51 are all you need. Instead, we'll jump right to the solution of our minimization problem (5). To clean up our notation a bit, let's define some helper variables:

$$\vec{p}^* = \frac{\sum_{i=1}^{n} w_i \vec{p}_i}{\sum_{i=1}^{n} w_i} \tag{6}$$

$$\vec{q}^* = \frac{\sum_{i=1}^{n} w_i \vec{q}_i}{\sum_{i=1}^{n} w_i} \tag{7}$$

It's easy to differentiate (5) with respect to the elements of $\vec{t}$ (at least try this part at home!), yielding the following relationship when we set this derivative to zero:

$$\vec{t}_{opt} = \vec{q}^* - M_{opt}\vec{p}^* \tag{8}$$

That is, we can find the optimal translation $\vec{t}_{opt}$ easily enough once we know the optimal matrix M_{opt}.

Finding M_{opt} takes a bit more bookkeeping. We'll need two more sets of helper variables:

$$\hat{p}_i = \vec{p}_i - \vec{p}^* \tag{9}$$

$$\hat{q}_i = \vec{q}_i - \vec{q}^* \tag{10}$$

We can reduce finding M_{opt} to a least-squares problem, so when the smoke clears we'll get the following "normal equation" solution:

$$M_{opt} = \left(\sum_{j=1}^{n} w_j \hat{q}_j \hat{p}_j^\top \right) \left(\sum_{i=1}^{n} w_i \hat{p}_i \hat{p}_i^\top \right)^{-1} \tag{11}$$

This finishes our story. From the inputs $(\vec{p}_1, \vec{q}_1, w_1), \ldots, (\vec{p}_n, \vec{q}_n, w_n)$ we can compute $\vec{p}^*$ and $\vec{q}^*$. We can use these to compute $\hat{p}_i$ and $\hat{q}_i$ for all i. Then, we use (11) to compute M_{opt}. Finally, we find $\vec{t}_{opt}$ using (8). The formulas might look complicated, but they're quite simple to evaluate; the 2×2 inverse in (11) can be found in closed-form with the usual formula (built into Eigen using the `.inverse()` method of `Matrix2d`).

We may wish to put additional constraints on M_{opt}. It's easy to see that regardless of our constraints on M_{opt}, the formula (8) for $\vec{t}_{opt}$ stays the same. This aside, if we want only to consider *similarity* matrices M that contain no shear–that is, they only can rotate and scale–then we constrain $M^\top M = \lambda^2 I$ for some $\lambda \in \mathbb{R}$ (think about why!) and find:

$$M_{opt}^{similarity} = \frac{1}{\mu} \sum_{i=1}^{n} w_i \begin{pmatrix} \hat{q}_i^\top \\ (-\hat{q}_i^\perp)^\top \end{pmatrix} \begin{pmatrix} \hat{p}_i & -\hat{p}_i^\perp \end{pmatrix} \tag{12}$$

where

$$\mu = \sum_{i=1}^{n} w_i \|\hat{p}_i\|^2 \tag{13}$$

If we force M only to encode rotations without any scaling, then (12) still works, but we replace μ with

$$\tilde{\mu} = \sqrt{\left(\sum_{i=1}^{n} w_i (\hat{q}_i \cdot \hat{p}_i) \right)^2 + \left(\sum_{i=1}^{n} w_i (\hat{q}_i \cdot \hat{p}_i^\perp) \right)^2} \tag{14}$$

Here, for vector $\vec{a} = (a_1, a_2)$ we denote $\vec{a}^\perp = (-a_2, a_1)$.

MLS simply applies the formulas we developed above with a clever choice of the w_i values. We're going to render an image by drawing a $k \times k$ grid of quads with evenly-spaced texture coordinates. In our code the grid of texture coordinates is stored in the variable `textureCoords`; you should *not* change the values in this array. We'll deform the grid to a new set of positions (stored in `imageCoords`) using the MLS algorithm and render the deformed image using textured triangles; see `display()` to see our basic setup.

The user shift-clicks the control points $\vec{p}_i$ to make red handles appear (shift-clicking those red handles again makes them disappear); the indices of these points in `imageCoords` are stored in the set `active`. The user then can drag the handles to new targets $\vec{q}_i$, the positions of which are stored in `imageCoords`. This much functionality is implemented for you in the skeleton code.

Your job is to implement `deformImage()`, which writes new values for elements of the array `imageCoords`. The basic algorithm goes like this:

- Iterate over each texture coordinate $\vec{p}$ in the array `textureCoords`.

- If $\vec{p}$ is one of the $\vec{p}_i$'s, then its position in `imageCoords` is $\vec{q}_i$ and can be left alone. Otherwise proceed with the steps below.
- Compute weights $w_i = \|\vec{p}_i - \vec{p}\|^{-2\alpha}$ for the provided constant $\alpha > 0$. Recall that $\vec{p}_i$ denotes the texture coordinate of the ith point in `active`. Notice what this choice of w_i is doing: Control points that are close to $\vec{p}$ get more weight than those that are far away.
- Compute M_{opt} and $\vec{t}_{opt}$ using the weights w_i you just computed and the formulas above.
- Replace the position in `imageCoords` with $\vec{q} = M_{opt}\vec{p} + \vec{t}_{opt}$.

In other words, each (sampled) point on the base texture computes its own least-squares map and uses that map to find a target.

Problem 5 (50 points)**.** *Implement MLS deformation starting from the skeleton code. You must implement the affine, similarity, and rotation-only versions of the algorithm, using command line options stored in* `deformationType` *to determine what to do in the application.*

Problem 6 (5 points extra credit)**.** *When does our choice of w_i in MLS break down? Can you suggest a replacement? Describe your solution in the written homework; no need to add it to your code, as the implementation change might (hint!) require a using a structure other than a grid mesh–a considerable change!*