For this project, I have made an implementation of the paper "Repulsive Curves" (2020) by Chris Yu, Henrik Schumacher, and Keenan Crane (https://doi.org/10.48550/ARXIV.2006.07859). I implemented it with the application of embedding non-planar graphs in 3D in mind.

Outline of README

This document will include: **(1)** Execution instructions with details about some examples, **(2)** an overview of the paper (with discussions of how my implementation related to each part), and **(3)** a summary of all the files involved in implementation.

1 Execution and Example Guide

Building this project follows a typical cmake/make build routine. Once inside the project directory, type:

```
mkdir build
cd build
cmake ..
make
```

Once built, you can execute the code from inside the `build/` by running on a given OBJ file (containing a low-resolution curve, which for the current parameters set up is about 5-20 vertices):

```
./repulsive_curves [path to curve.obj]
```

There is a thorough description in the main.cpp file of parameters that can be tweaked to adjust the behaviour. They are also talked about later in this document.

Since I am personally most interested in the application to taking non-planar graphs and embedding them in 3D space, all the examples that I worked with were for this application and my code is tailored to this application.

If you just run the following, the program will allow you to animate embedding the Petersen graph in 3D space.

```
./repulsive_curves
```

This is my personal favourite graph.

You can also try using an alternate 2D embedding of the Petersen graph with the file `petersen_alternate.obj` and additionally try out the complete graph of 5 vertices with `K5_graph.obj`. I found it cool to watch how in-plane relaxations for `K5` happen very quickly (the border rounds out into a circle), but the algorithm takes a bit before figuring out that it can obtain a much lower energy by using the 3rd dimension. Though, this works faster if you start by perturbing the mesh.

I also included a 3,2 bipartite graph in the file `K_3-2.obj`, which was an interesting test case. The algorithm seems to settle on a local minimum which has much lower energy than the initial state but is not the global minimum. This is likely a cause of there being a small number of points which results in most of the edges being fixed to have a constant length (any edge connected to 2 or more other edges must have a constrained length as per the paper). The large number of length constraints makes the jump from the local minima to the global much harder.

No changes to the parameters in the main file are needed to run these 4 examples.

2 Paper Summary and Implementation Overview

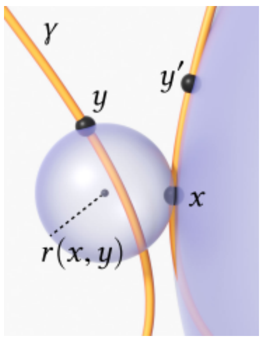
The Goal

Given a geometry file of a curve, this paper broadly aims to animate the curve "repelling" from itself, meaning that portions of the curve that are close together are pushed apart. You can imagine it like having tangled up ear buds and having the parts of the wire which are close together push apart until the ear buds untangle, falling into a "more relaxed state".

The paper does this repulsion through the minimization of the "tangent-point energy".

Tangent-point Energy

Imagine taking 2 points on the curve and drawing a sphere such that the sphere is tangent to the first point and contains the second point (the picture below is taken from the paper and shows this sphere). Then, you get one form of the tangent point energy by taking the reciprocal of this radius to some power and integrating over all pairs of points on the curve.



The radius is given by the following formula (screenshot taken from the paper):

$$r(x,y) = \frac{|\gamma(x)-\gamma(y)|^2}{|T(x)\times(\gamma(x)-\gamma(y))|}$$

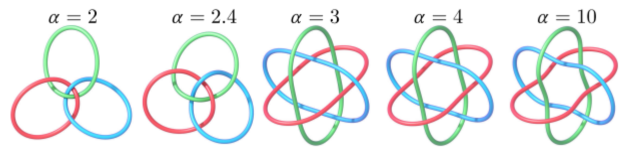
However, the paper uses a generalization the tangent point energy by putting the denominator of the radius formula to the power of a new variable α, and the numerator to a new variable β (instead of 2). Thus, when we take the reciprocal and integrate it, we get the following formula for the generalized tangent point energy.

$$\mathcal{E}^{\alpha}_{\beta}(\gamma) := \iint_{M^2} k^{\alpha}_{\beta}(\gamma(x),\gamma(y),T(x))\,dx_\gamma dy_\gamma, \quad k^{\alpha}_{\beta}(p,q,T) := \frac{|T\times(p-q)|^{\alpha}}{|p-q|^{\beta}}.$$

In the discrete setting, we get the following formula with sums instead of integrals:

$$\hat{\mathcal{E}}^{\alpha}_{\beta}(\gamma) = \sum\sum_{I,J\in E, I\cap J=\varnothing}(\hat{k}^{\alpha}_{\beta})_{IJ}\ell_I\ell_J, \quad (\hat{k}^{\alpha}_{\beta})_{IJ} := \tfrac{1}{4}\sum_{i\in I}\sum_{j\in J}k^{\alpha}_{\beta}(\gamma_i,\gamma_j,T_I).$$

The choice of α and β can be thought of as variables that influence how much weight is put on local interactions rather than global interactions and how much bending the curve will have to do in order to avoid these interactions. This is visually demonstrated in this figure from the paper:



In my implementation, I use $\alpha = 3$ and $\beta = 6$, as recommended in the paper.

Picking a Smart Descent Direction

The part of the paper which I find most interesting is the way that it actually minimizes this energy.

Typical gradient descent involves taking a step in the opposite direction of the gradient in each iteration. The reason for doing this is that the gradient points in the direction of steepest local increase of the function. However, we can re-imagine the gradient by asking "what do we mean by steep". Typically the gradient is defined using the L2 inner product to "define steepness". Namely, we say that the gradient of an energy E is the function which has the property that the L2 inner product of E with any other function f gives you the directional derivative of E in the direction f.

With a very careful choice of the inner product, we can redefine what we mean by "steepness" in gradient descent in a way where we pick a smarter direction of descent than just the negative of the L2 gradient. In particular, the paper uses the fractional Sobolev gradient, which is what you get from the fractional Sobolev inner product. Altogether, this results in what you call "fractional Sobolev descent" denoted by H^s where s is a parameter set to $(\beta-1)/\alpha$. (Note: σ is a variable used in many formulas and it is equal to $s-1$)

Considering a discrete curve, we can write this inner-product as $\langle a,b\rangle_{H^s_\gamma} = a^T\overline{A}b$ for a matrix $\overline{A}$ (which I constructed in the paper implementation in `build_A_bar.cpp`). Furthermore, this matrix satisfies $\overline{A}g = d\varepsilon^{\alpha}_{\beta}$ where $d\varepsilon^{\alpha}_{\beta}$ is the derivative of the tangent-point energy $\varepsilon^{\alpha}_{\beta}$ with respect to each vertex coordinate (which I constructed in `loss_derivative.cpp`), and $g\in\mathbb{R}^{3|V|}$ is the discrete fractional Sobolev gradient - which will be used to calculate the direction of descent.

Differentiating $\varepsilon^{\alpha}_{\beta}$

Descending With Constraints

If we just minimize the tangent-point energy as-is, the curve will just keep expanding and become arbitrarily large. To avoid this behaviour (and other weird behaviours), we can invoke a number of different constraints. In my implementation, I added a constraint so that the total length of the curve stays constant, and another constraint so that at points where 3 or more edges are connected, the length of each of those edges remains fixed (this helps with applications to graphs).

To actually account for constraints, we use the following process

(which is applied at the end of the file `repulsion_iteration.cpp`):

First, we project the Sobolev gradient g (described earlier) onto the tangent of the constraint space. The tangent of the constraint space is a matrix C with k rows (for the k constraints) and $3|V|$ columns (for each variable that we differentiate by). I computed all the derivatives by hand on paper, then checked my results with Mathematica, and finally programmed it into the file `constraint_derivative.cpp`.

Then, in `repulsion_iteration.cpp`, the matrix $\overline{A}$, $d\varepsilon^{\alpha}_{\beta}$ (both previously described) and C are combined to make the saddle point system:

$$\begin{bmatrix}\overline{A} & C^T\\ C & 0\end{bmatrix}\begin{bmatrix}\tilde{g}\\ \lambda\end{bmatrix} = \begin{bmatrix} d\mathcal{E}^{\alpha}_{\beta}|^{T}_{\gamma}\\ 0\end{bmatrix}$$

λ is a vector of k Lagrange multipliers and $\tilde{g}$ is the new descent direction after the projection.

I solved this equation in `repulsion_iteration.cpp` with an iterative LU solver (issues arose when partial pivoting was used, so a full pivoting LU solver was used instead).

Then, as suggested in the paper, backtracking line search was used to find the appropriate step size. However, backtracking line search requires 2 parameters (one for the Armijo rule, and one for the rate at which the time step should decay), which were not given. Thus, I experimented and found that 0.01 and 0.9 worked well, but sometimes the algorithm worked better when these had different values (in `main.cpp`, the parameters are called `a_const` and `b_const` and more is described there).

Moreover, backtracking line search was modified to account for constraints. First, the descent direction is iteratively projected onto the constraint space.

This projection is done by solving the following saddle system for x:

$$\begin{bmatrix} \overline{A} & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ \mu \end{bmatrix} = \begin{bmatrix} 0 \\ -\Phi(\tilde{\gamma}) \end{bmatrix}$$

(μ consists of more Lagrange multipliers)

Then, I update the vertex positions with $\tilde{\gamma} = \tilde{\gamma} + x$, and solving the system again repeatedly until the constraints evaluated at the new vertex position, $\Phi(\tilde{\gamma})$, is below some tolerance (which you can set with a variable called `tolerance` in `main.cpp`), or until a certain number of iterations has passed (set with a variable called `max_iters` in `main.cpp`).

Finally, the new vertex positions are checked against both the Armijo rule and the constraint tolerance (in case the max number of iterations was exceeded before the tolerance got low enough). If the vertex positions don't satisfy both, the step size is decreased by a factor of `b_const` and everything here is repeated again.

Finally, when the Armijo rule and the constraint tolerance are met, the vertex positions are updated in the viewport and the descent iteration is complete.

Note that since my code is designed for graph embeddings, it doesn't matter whether the descent step preserves the isotopy class of the curve (i.e. we don't care whether the curve steps over itself, in fact, we want that to happen if it means a lower energy).

3 File Overview

For all the following files, additional information about specific parameters can be found in the header files.

`main.cpp`

This was adapted from the `main.cpp` file from the registration assignment. At the top of the file, there are some parameters that you can set, as mentioned many times above. `main.cpp` first reads in a triangle mesh which has a x-y-z axis for easier viewing. Then, it imports the curve OBJ file, using a function in `read_curve_file.cpp` which I made. Then, `init_curve.cpp` is called, which is described below, and based on the user's key presses, iterations of the descent algorithm can be run (calling `repulsion_iteration.cpp`) or the mesh can be reset to its original state.

`read_curve_file.cpp`

Standard file IO methods are used to read an OBJ file and extract vertices and edges.

`init_curve.cpp`

This file primarily constructs 2 variables which I came up with to avoid recomputing things that only depend on the topology of the mesh in `repulsion_iteration.cpp`. Namely, it constructs a vector of vectors `Ac` which contains indices of all the edges disjoint from a given edge (more details in the header file). It also constructs a vector of vectors `E_adj` which for a given index of a point, gives you all the indices of the edges which that point is a part of. Finally, `init_curve.cpp` computes the lengths of every edge in the curve in its initial state, which is later used in `repulsion_iteration.cpp` to evaluate different length constraints.

`repulsion_iteration.cpp`

First, different variables are constructed containing data about the mesh which will be used later within the current iteration. For example, the current edge lengths `L` and the unit tangent vectors of each edge `T` are constructed. Then, $\overline{A}$ (explained previously) is constructed through a series of steps. First, weight matrices W and $W0$ are constructed with the file `build_weights.cpp` (described below). Then, `build_A_bar.cpp` is called to build $\overline{A}$. Next, the derivative of the energy is constructed with `loss_derivative.cpp`. Finally, the constraint derivative C (described previously) is constructed with `constraint_derivative.cpp`. All of these new variables are used to construct and solve a saddle system which gives us the initial descent direction. Now, the modified backtracking line search described previously is implemented. Finally, once the vertices have been updated, the barycenter of the new mesh is calculated and the mesh is shifted so that the barycenter of the new mesh is at the origin (this is something I added myself).

`build_weights.cpp`

Two weight matrices W and $W0$ are built as described in the paper. They will be used in `build_A_bar.cpp` to build $\overline{A}$. W corresponds to the discrete high-order fractional Sobolev inner product. W0 corresponds to the discrete low-order term (which will build a matrix B0 which will act like a regularizer for the more important matrix B).

`build_A_bar.cpp`

The $\overline{A}$ matrix is constructed in multiple steps. First, a matrix B is constructed from W which represents the higher-order inner product matrix. Then, $B0$ is constructed from $W0$ which represents the lower-order term. Then, we build a matrix $A = B + B0$. Finally, $\overline{A}$ is constructed by concatenating A 3 times along the diagonal, to make a block diagonal matrix $\overline{A}$.

`loss_derivative.cpp`

To save on computation time when computing the derivative of the energy, I derived the derivative of the energy by hand (it took multiple pages), simplified the result, and then checked the answer with Mathematica. Note that Mathematica couldn't be applied straight away since it returns a very long unsimplifed solution which would be almost impossible to copy into C++ without making mistakes. The derivation of the derivative is in the project file, but it is not formatted in a way that is easy to read (it is mainly just there to display that I have actually done it) (see `messy_derivative_calculations.pdf`). Computing the derivative in `loss_derivative.cpp` just involved typing up the equations for all the different cases once they were computed by hand.

`constraint_derivative.cpp`

Similar to the derivative of energy, the derivative of constraints was done on paper, then checked on Mathematica, and finally converted from math to code.