MAT357 Project 2

Adam De Broeck

January 4, 2024

The Idea

The purpose of this interpolation method is to try to create decent-looking interpolation curves from a recurrence relation that uses Euler's method (semi-implicit) in order to trace out an appropriate curve. Because of this, this method is not a traditional function, piecewise or other. Instead, the function places the "cursor" at the first point — this method assumes that the points have some sort of order to them — and lets it "fall" into the other points. A similar concept might be placing a satellite nearby a planetary body. The satellite would "fall" into the planet's gravity well and as such, move closer to the planet.

This method is not bound by a standard cartesian based function and could likely be approximated via a piecewise parametric function, however, such a function would be worthy of a project by itself and is outside of the scope of this method. Instead, this method will produce a list of points that describe the line created by the interpolation — see Possible Extensions for more information.

The benefits of this method are largely seen in the amount of customizability that the interpolation function has to create curves of varying shapes and sizes. Additionally, while the main goal of this method is experimental in nature, simplicity is another key goal. Unlike Lagrange polynomials, this method can easily add points to either the middle, or the end without re-computing the entire function. This method also doesn't deal with the increasing complexity required for an NxN matrix via a spline. In the end, the performance target of this method is not to beat existing methods but rather stay within a linear $O(N)$ growth pattern.

The Computation

The basis of this idea stems from a semi-implicit implementation of Euler's method and all code was written in `C++` — graph plotting was performed via `matplot++` and `gnuplot` while vector functions were used from the `GLM` header library. The goal of this method is to start at the first point and progressively move to subsequent points while changing "target" after reaching each one. The functional representation of this is as follows:

$$\vec{a_i} = \frac{\vec{x_i} - \vec{t_i}}{\|\vec{x_i} - \vec{t_i}\|^2}$$
$$\vec{v}_{i+1} = \vec{v_i} + h\vec{a_i}$$
$$\vec{x}_{i+1} = \vec{x_i} + h\vec{v_i}$$

Where $\vec{t_i}$ is the position of the current target point and h is the current step size.

The first notable issue that came up before even getting to the algorithm itself was the step accuracy needed in order to properly detect when the interpolation had reached a target point. Typically, with floating point values one must check if a value is within a specific error range, `epsilon`. This is because floating point values using the `IEEE754` standard do not have perfect accuracy and thus equality operations cannot be used. When choosing the `epsilon` value, a small value in the thousandths would normally suffice. However, due to the nature of the ever-increasing second derivative value (velocity), it is incredibly easy to "overshoot" this value and fail to validly pass the target point by phasing over the point in one step.

In order to fix this error and be able to use values of a wide variety, a dynamic step size was introduced. After each step of integration, the step size is recalculated based on the current step's calculated velocity. This uses the equation $\frac{1}{10\|\vec{v_i}\|}$ in order to adjust the step size. In code this is as follows:

```
// Modify future step size
stepSize = STEP_SIZE_MOD * std::min(0.1, 1.0 / (10.0 * glm::length(vel)));
```

The `std::min` serves to prevent the step sizes from ever dynamically re-sizing to a large value — this can cause erratic behavior. Additionally, the `STEP_SIZE_MOD` preprocessor definition is used as a scalar to overall tune the steps sizes used by the interpolation method.

Unintentionally Stable Orbits

The first step to making this semi-implicit Euler integration work was to apply a gravitational pull to the cursor using the acceleration function $\vec{a_i} = \frac{\vec{x_i} - \vec{t_i}}{\|\vec{x_i} - \vec{t_i}\|^2}$. The code for this is as follows:

```
// Get vector from current position to current well
glm::vec<2, double> diff = currentWell - pos;
double length = glm::length(diff);

// Modify the diffs magnitude to act as an acceleration
// toward the current gravity well
diff = glm::normalize(diff);
diff *= (1.0 / (length));
```

After tweaking coefficients by hand and testing a wide variety of values a pattern emerged. No matter the strength of the gravitational pull, whenever only one force acted on cursor the resulting line had a tendency toward creating extremely stable elliptical orbits (albeit rotating) around the target point. This is of course undesirable as the plotted line must reach the point to continue. Below is an example of this behavior:

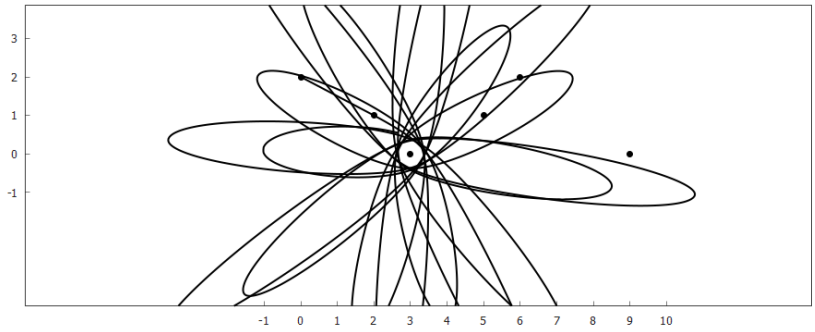


Figure 1: Example of stable rotating orbit around point #3.

Problems at the Event Horizon

There were a couple of methods used in order break up the stable orbiting behavior exhibited earlier. One such technique was to implement a sort of "drag" to the cursor. By limiting the cursor's max speed, the typical orbiting behavior could be broken due to the cursor's existing velocity being capped and subsequently overpowered by the point that is exhibiting a pull. This was implemented in code like so:

```
// Impose a speed limit
if(glm::length(vel) > SPEED_LIMIT)
{
    vel = glm::normalize(vel);
    vel *= SPEED_LIMIT;
}
```

In this case, `SPEED_LIMIT` is a constant preprocessor definition that is set to `1.0` by default. This speed limit was effective in preventing the cursor from getting stuck in elliptical orbits around a point, however, it wasn't enough as a new problem arose. Below is the resulting attempt which implements a speed limit on the cursor:

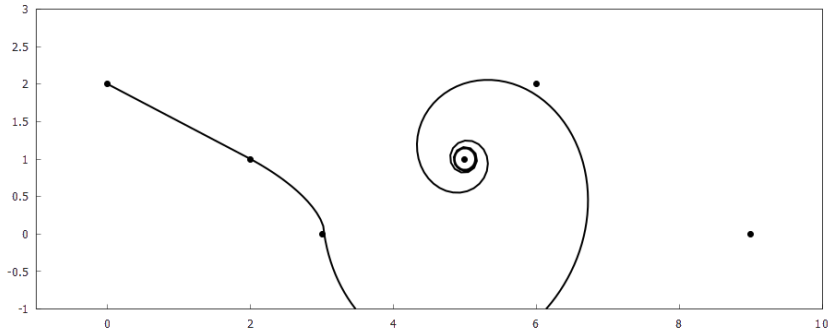


Figure 2: Example of the cursor getting stuck around point #4.

Unfortunately, this technique by itself doesn't solve the problem of the cursor getting stuck in an orbit around the object. It does however exhibit an interesting property which is present in certain celestial bodies with extreme mass for their nearly infinitesimally small (relatively) size — black holes. Interestingly enough, the cursor's perfectly circular orbit behaves similar to light that ends up stuck at the event horizon of a black hole. The cursor's

arbitrary speed limit acts the same as the maximum theoretical travel speed of light, and as such, the same property is exhibited — i.e., the object cannot move any closer to the body with a gravitational pull as this would violate the imposed speed limit.

Looking Ahead

In order to break up the new orbiting nature of the cursor, an additional point can be used simultaneously to act as an accelerator or brake at the right moments. By adding in a portion of the gravitational pull of the next point in the list of points, the cursor gets nudged off course which results in it almost never getting stuck in a loop. This has the added benefit of allowing a smoother transition between points due to the multi-point influence. The new code for this secondary point is listed below:

```
//...
glm::vec<2, double> diff2 = {0.0, 0.0};
double length2 = 0.0;
if(nextWellExists)
{
    diff2 = nextWell - pos;
    length2 = glm::length(diff2);
}

//...
if(nextWellExists)
{
    diff2 = glm::normalize(diff2);
    diff2 *= (1.0 / (length2));
}
```

This finally solves the issues with the cursor getting stuck at points, but it doesn't exactly create desirable looking interpolation by itself. See below:

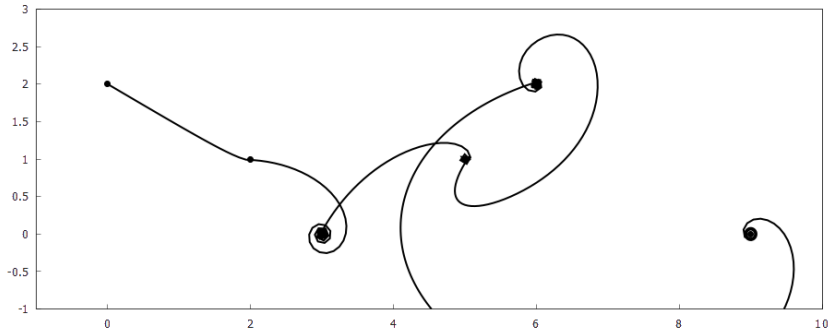


Figure 3: Example cursor completing line.

From here on out the remaining tweaks are additional coefficients and changes to specific values in order to tune the interpolation method to get more smooth results. One of the biggest contributors to getting a more smooth curve was changing the overall pull strength, the ratio between pull force from current and next gravity well targets, as well as setting the second gravity well to push the cursor instead of pull. This resulted in many knobs that could be hand tuned in order to get the best look for the interpolation. The code changes for these are below:

```
// Settings
#define MAX_ITERATIONS 3000     /* Default 3000 */
#define PULL_STRENGTH 4.0       /* Default 4.0  */
#define OPPOSING_FORCES true    /* Default true */
#define FORCES_RATIO 2.0        /* Default 2.0  */
#define SPEED_LIMIT 1.0         /* Default 1.0  */
#define STEP_SIZE_MOD 0.5       /* Default 0.5  */
#define REQUIRED_DISTANCE 0.1   /* Default 0.1  */
```

```
// Accumulate velocity from gravity wells
vel += PULL_STRENGTH * (FORCES_RATIO / (FORCES_RATIO + 1.0)) * 2.0 *
           (diff1 * stepSize) +
       (OPPOSING_FORCES ? -1.0 : 1.0) *
       (PULL_STRENGTH * (1.0 - (FORCES_RATIO / (FORCES_RATIO + 1.0))) *
           2.0 * (diff2 * stepSize));
```

Broken Symmetry

After generating graphs for the other point sets it became clear that there was broken symmetry with this method — this was especially noticeable for graph #4. Below are representations of the graph when the point set is traversed forward:

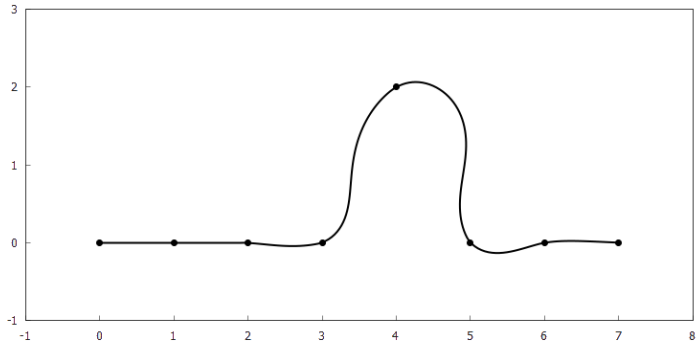


Figure 4: Example graph leaning right.

Due to the nature of the "momentum" that the cursor carries as it moves through the points, the resulting graph tends to have a bit of a lean to it. This can be seen as soon as the point set is traversed in the opposite direction.

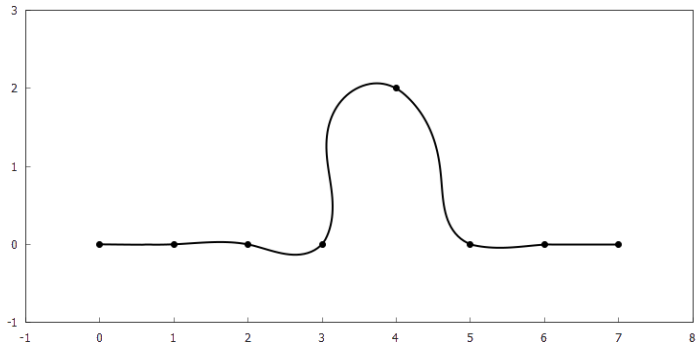


Figure 5: Example graph leaning left.

Typically, the intention for this interpolation method should be for it be uniform regardless of direction. A quick fix for this problem was easily achievable by averaging out the results of both sets. In order to do this, however, the resulting sets of points needed to be equal size. Due to the nature of how the sets are calculated (They continue to step forward until either the maximum number of steps have been reached, or the last point is reached by the cursor) both traversals needed to have the same number of points. The quick — and admittedly dirty — solution to this was to drop the last results of whichever set was larger and average them anyway. After being blended, the first and last points are added back into the results to ensure the function remains continuous from point 0 to point n. The final image is as follows:

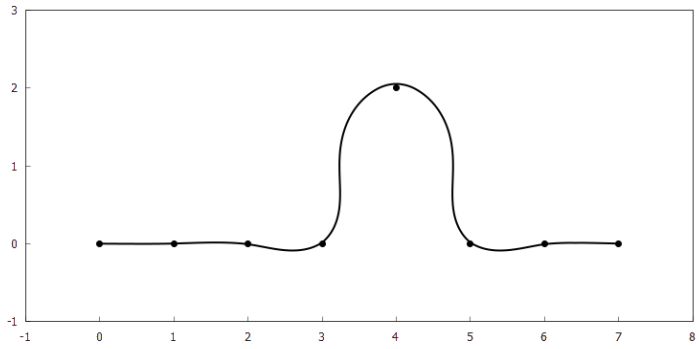
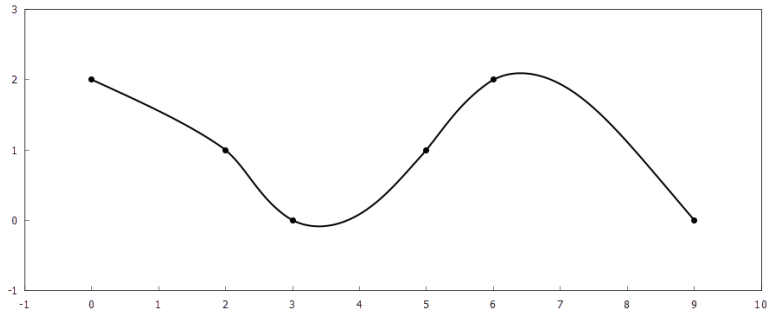


Figure 6: Example averaged final graph.

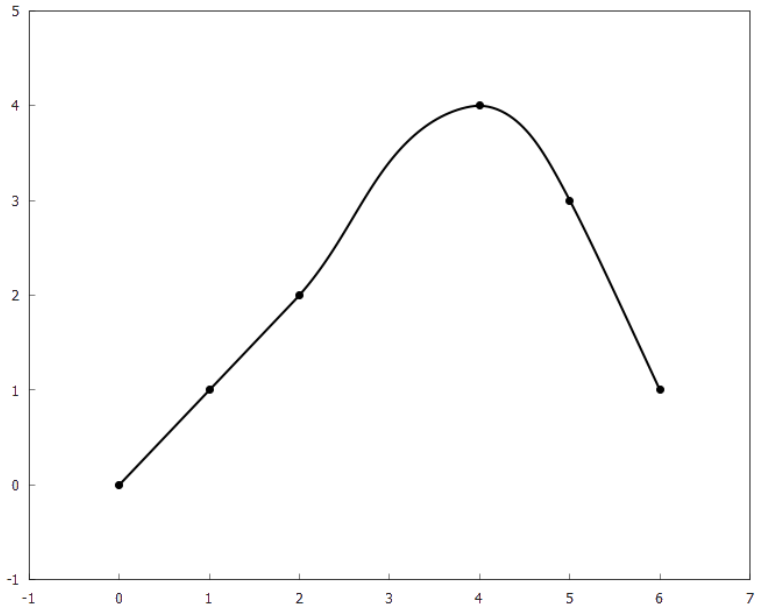
It's important to note that the intention of this method is not to perfectly pass through the set of input points but rather to get as close as is reasonable while still making a "good" looking graph.

The Results

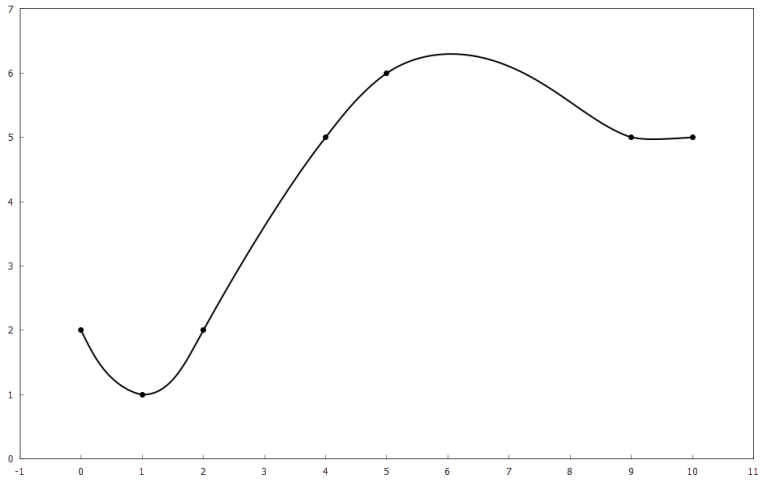
Graph #1



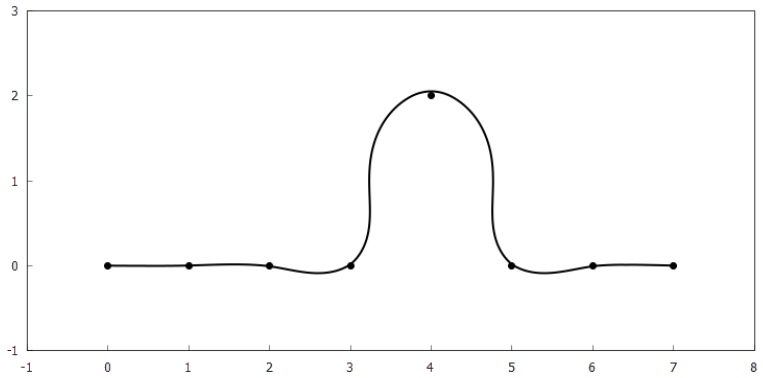
Graph #2



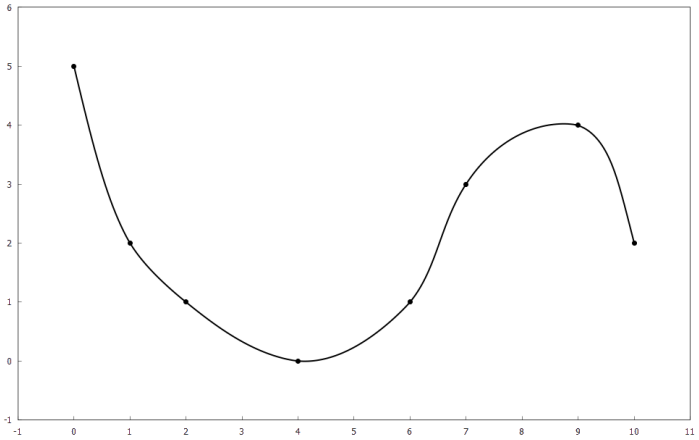
Graph #3



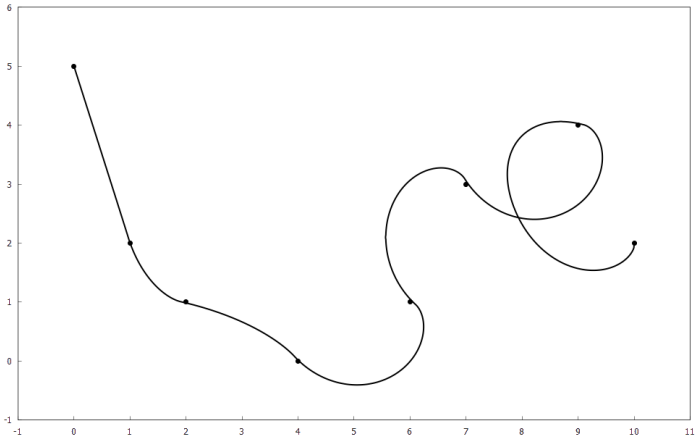
Graph #4



Graph #5



Bonus Graph




```
// Settings
#define MAX_ITERATIONS 3000     /* Default 3000 */
#define PULL_STRENGTH 0.18      /* Default 4.0  */
#define OPPOSING_FORCES false   /* Default true */
#define FORCES_RATIO 100.0      /* Default 2.0  */
#define SPEED_LIMIT 0.5         /* Default 1.0  */
#define STEP_SIZE_MOD 0.5       /* Default 0.5  */
#define REQUIRED_DISTANCE 0.1   /* Default 0.1  */
#define USE_AVERAGE false
```

Possible Issues

In the general case (default values), even when the points are shuffled randomly a decent interpolation curve is generated — this is only a couple of very similar sets of points, however. It should be noted that the averaging feature can cause significantly distorted values whenever the forward and backward set of resulting points are extremely dissimilar. That being said, the feature should only be used after a known decently symmetric interpolation has been found.

There's also the untested possibility that due to the somewhat hard coded values of the step function, this method won't scale as a set of points scales — i.e., the larger the scale of the points the straighter the lines, and the smaller the scale of the points the more likely for the resulting set to be erratic. As an attempt to tune/prevent small scale sets from becoming erratic quickly, the `STEP_SIZE_MOD` preprocessor command has been provided. Functions that have many sharp edges / turns may also not perform well with the current default settings or may not fully complete properly.

Additionally, this method may not scale well in extremely large point sets due to the step by step nature of the semi-implicit Euler integration being used. A more traditional interpolation may be desired if speed is necessary. Furthermore, the curves generated are rather "organic" and cannot be accurately represented reasonably with functions. There's also no way to insert some value t in order to get an output (like a parametric function).

Possible Extensions

As a direct solution to the previously mentioned issue of not being a direct function, a hashmap lookup table may be used in order quickly interpolate some existing x or y to the closest value on the line. This could be done by caching both the x and y representations in a hashmap, sorting the key values to get the closest 2 key representations and interpolating between their corresponding values. Similarly, if the desired input is some uniform variable t, then saving and caching the time steps used in order to arrive at the current value of $\vec{x_i}$ could be used instead of x / y keys. This would allow a uniform lookup of t to get a linearly interpolated value of x and y.

As for the averaging issue experienced earlier where some forwards and backwards set results are too different to average, another method could be used instead. Dropping the last values and simply inserting the first and last

point is crude and often times leads to minor errors / cuts in the continuity of this method's second degree integration ($\vec{v_i}$).

Finally, using only 2 future points to calculate the next acceleration of the cursor works well, but this method could be extended to use 3, 4, or as many points as one desires. It's possible some interesting behavior may emerge from the latter if combined with higher dimension functions.