UM EECS 487

Lab 10: Splines I - Join The Dots!

[Interpolation and Approximation]

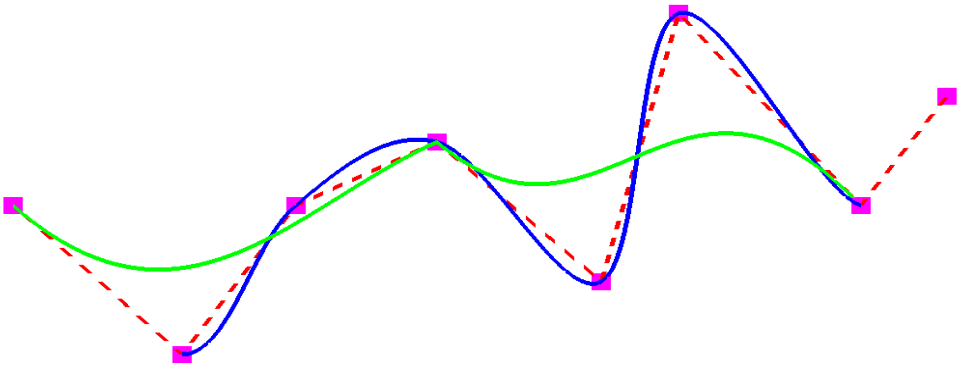

Fig 1: Screenshot showing one particular arrangement of points and [**red**] linear interpolation, [**blue**] Catmull-Rom spline interpolation, and [**green**] Bézier curve approximation

In this lab we will experiment with fitting, interpolating, and approximating curves through a given set of points. When you compile and run the code you will find a set of 8 points (highlighted by pink squares) in a row which you can position these points by dragging them with the mouse. The `keyb()` and `disp()` functions in `main.c` manage calls to functions that perform the fitting:

- pressing '1' binds to `draw_linear()` which draws the linear interpolant,
- pressing '2' binds to `draw_catmull_rom()` which computes and draws the Catmull-Rom spline interpolant, and
- pressing '3' binds to `draw_bezier()` which computes and draws the Bézier curve approximating the points.

Stubs for the above functions are provided in `draw.c` and your task is to implement the body of them.

Instructions

- Get comfortable with moving the points with the mouse.
- Start with the `main.c` file. Look at the `keyb()` and `disp()` functions which manage flags and calls to draw the curves.
- All the points are stored in the global array `points[NUM_POINTS][4]`. While these are stored as 4-vectors, only the first two elements in each matter, since we are working in 2D. The only reason that these are 4-vectors is for compatibility with the rudimentary math library we have been using in previous labs.
- When you implement one of the curves as instructed below, if you don't see anything on the screen when you run the program, try pressing the toggle keys as described above.
- Implement `draw_linear()` in `draw.c`. This should linearly interpolate between successive points.

- Implement `draw_catmull_rom()` in `draw.c`. This should interpolate all points except the first and last. For each pair of successive points, calculate the coefficients of the spline, and then approximate the section between them with a number of short line-segments. You are free to choose the number of samples for the parameter *u* between the points to create these line segments.
- Implement `draw_bezier()` in `draw.c` to approximate the curve between four successive points. Calculate the coefficients of the spline for point-sets {0,1,2,3} and {3,4,5,6}. Choose the number of samples for the curve parameter *u*, and approximate the section with a number of short line-segments.
- Move the points around and examine the advantages and limitations of each spline with respect to continuity (C^0 continuity), smoothness (C^1 continuity or differentiability), interpolating vs. approximating behavior, and computational effort.
- To gain a better understanding of spline methodology, approximate the section between the first two points and the section between the last two points with quadratic splines for the Catmull-Rom case. These quadratic curves must have the same slope as the rest of the Catmull-Rom spline at the point where they meet it.
- [Optional] You will notice that for making large, sweeping curves, the amount of short line segments drawn, by *u*, may be insufficient for the curve to appear "smooth." Implement, or just think about, some way in which this could be remedied, that is, implement some form of adaptive sampling. This is related to how much the curve is "curving" over a single step.

Matrix-Vector Helper functions

A crude implementations of a few vector and matrix operations is provided in `matvec.h` and `matvec.c` Matrices are `Glfloat[4][4]` and vectors are `Glfloat[4]`. All these functions support overwrites, i.e., the result vector can be the same as one of the inputs.

Function Name	***Action***
`copy_vec`	Copies a vector to another
`copy_mat`	Copies a matrix to another
`swap_vec`	Swaps the two given vectors
`zero_mat`	Fills the supplied matrix with 0.0
`identity_mat`	Makes the supplied matrix a 4x4 Identity matrix
`normalize_vec`	Assigns the second vector the normalized version of the first vector
`dot`	Returns result of dot product
`cross`	Assigns the third vector the cross product of the first two vectors

`lin_comb2`
`lin_comb3`
`lin_comb4`
	Assigns the linear combination of the first 2/3/4 vectors to the vector that appears as the last argument. A linear combination of a bunch of vectors is simply the sum obtained by adding these vectors, each scaled by a scalar constant, e.g., position vectors are linear combinations of the basis vectors, with the coordinates being the scalar coefficients.

`transpose`	Assigns the second matrix the transposition of the first matrix
`mat_vec_mult`	Assigns the second vector the multiplication of the first vector by the given matrix

Function Name	***Action***
`invert_mat`	Non-destructively invert the given matrix
`printf_mat`	Write the given matrix to the output stream, useful for debugging