Programming Assignment 03

Lagrange Interpolating Polynomials

Nick Handelman

2/14/2018

1 Project Description

For this project, I developed an Octave program that implements Lagrange's Interpolating Polynomial Formula. The program can build the formula for degrees 1,2,3,...,∞. In the program, the formula is applied specifically to the 4 equations in problem #1 in section 3.2, with x = 0.45. The linear interpolation is performed using x-coordinates: 0.0 and 0.6. The quadratic interpolation is performed using x-coordinates: 0.0, 0.6, 0.9. The output of the program includes, for each of the 4 equations, the function value of x, linearly interpolated value of x, quadratically interpolated value of x, relative errors, and graphs of these points and their related equations.

1.1 Lagrange Interpolating Polynomial Formulas

$$P_n(x) = \sum_{k=0}^{n} f(x_k) L_{n,k}(x) \tag{1}$$

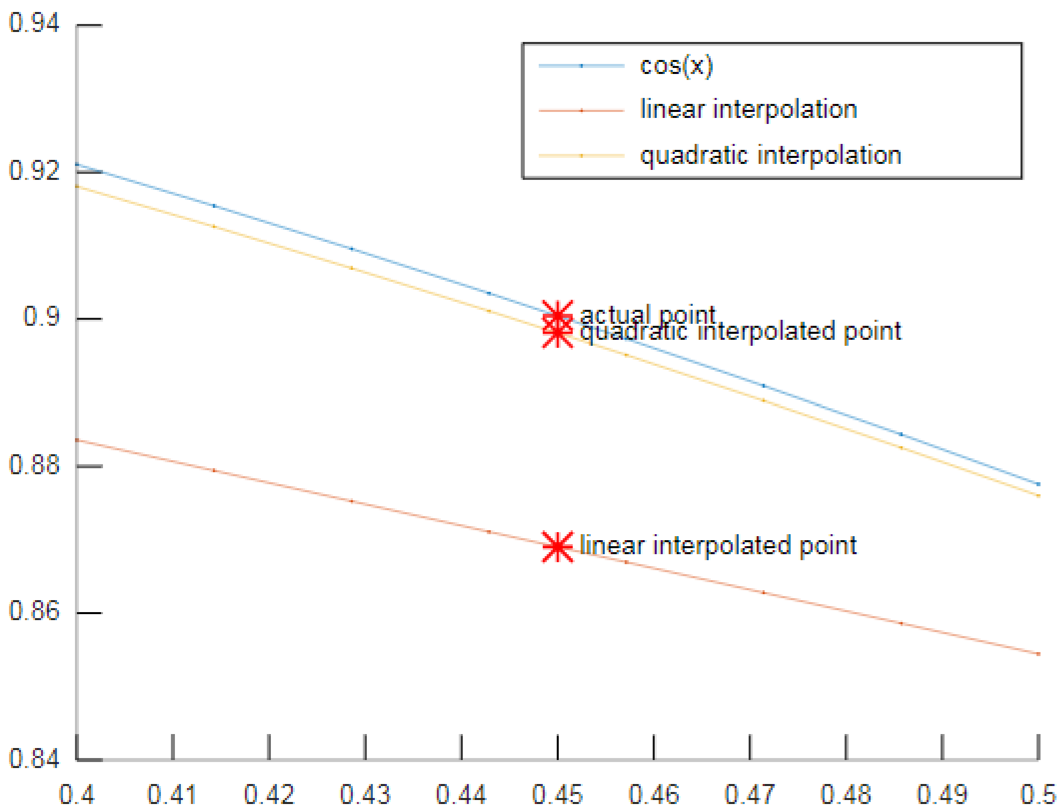
$$L_{n,k} = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x_k-x_n)}, k = 1, 2, ..., n \tag{2}$$

1.2 Program Description

lagrange.m (Appendix B) implements the Lagrange Interpolating Polynomial Formulas (1) and (2). This function builds the combined formula in a string based on the values received for the parameters and converts the string to a function that is returned to lagrange.m's caller. The formulas certainly could have been implemented directly, but with this method, I could see the actual equation (1). Also, the built-in Octave function func2str() doesn't have an analog in other programming languages I have worked with, so I thought it would be interesting to implement equations (1) and (2) in this way.
proj3.m (Appendix A) calls lagrange.m to build the degree 1 and degree 2 interpolating equations. It then calculates the interpolated point, the relative error, outputs the results and graphs the actual equation and the interpolating polynomials.

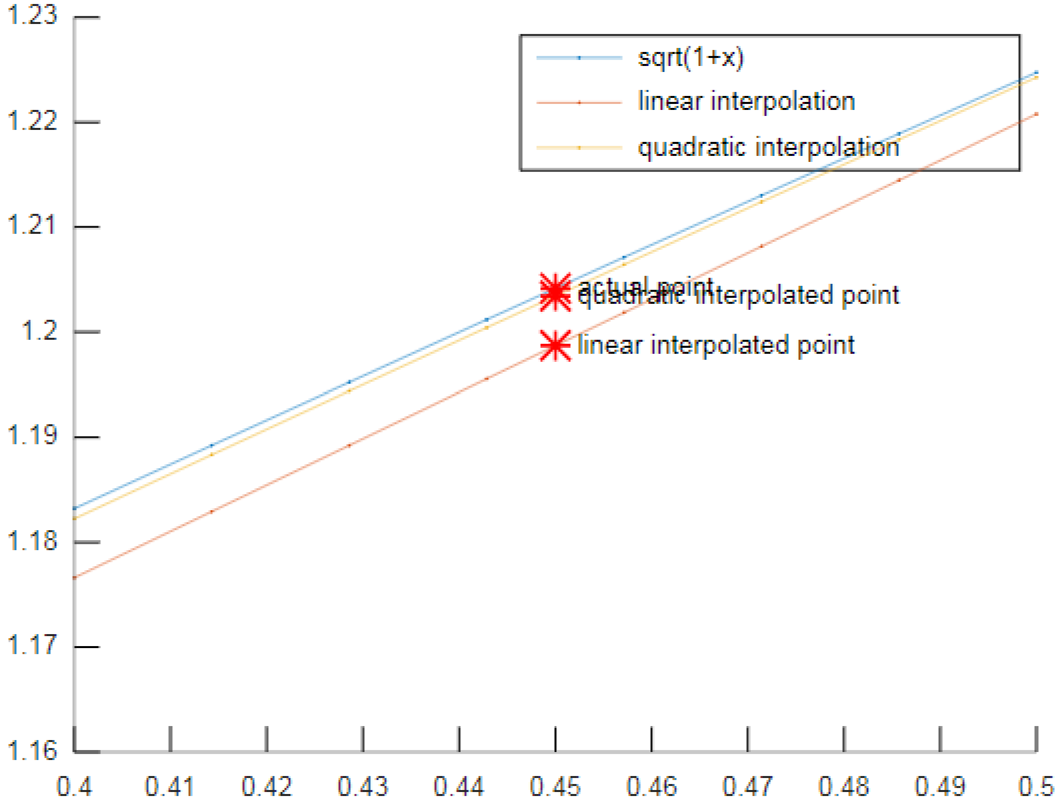
2 Analysis

2.1 $f(x) = \cos(x)$



Program Output
f(x) = cos(x). Actual: f(0.45) = 0.90045.
Linear Interpolation: P(0.45) = 0.86902. Relative Error: 0.034899.
Quadratic Interpolation: P(0.45) = 0.89812. Relative Error: 0.0025859.

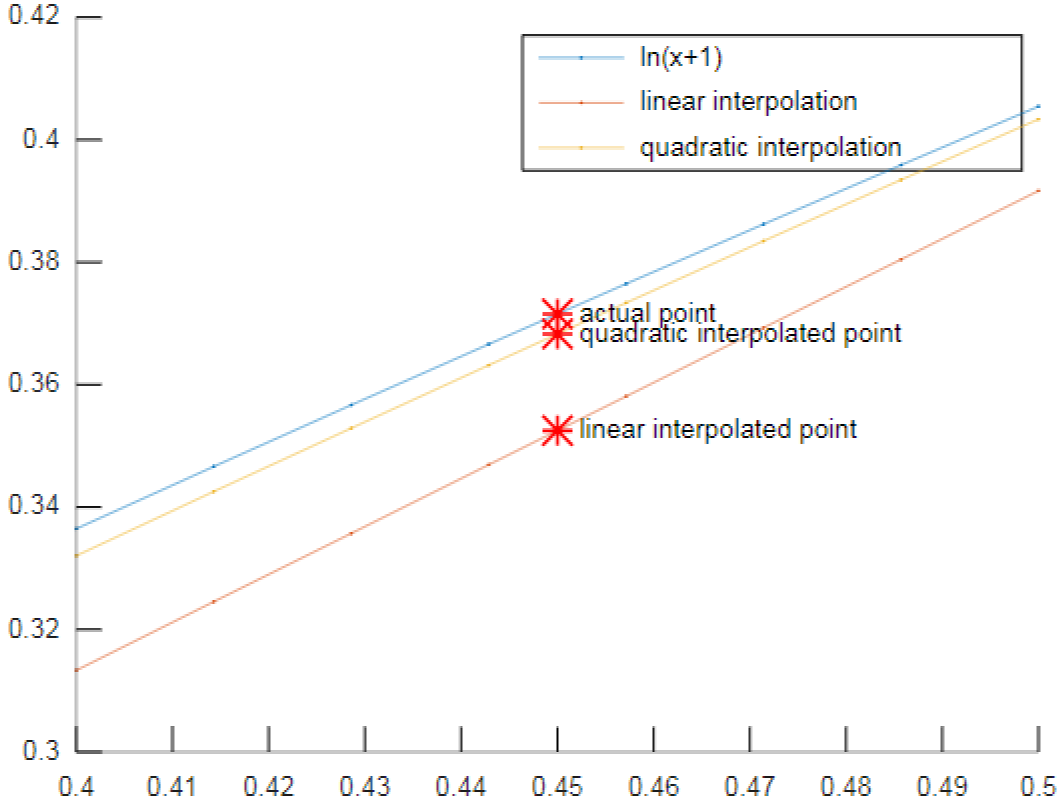
2.2 $f(x) = \sqrt{1+x}$



Program Output

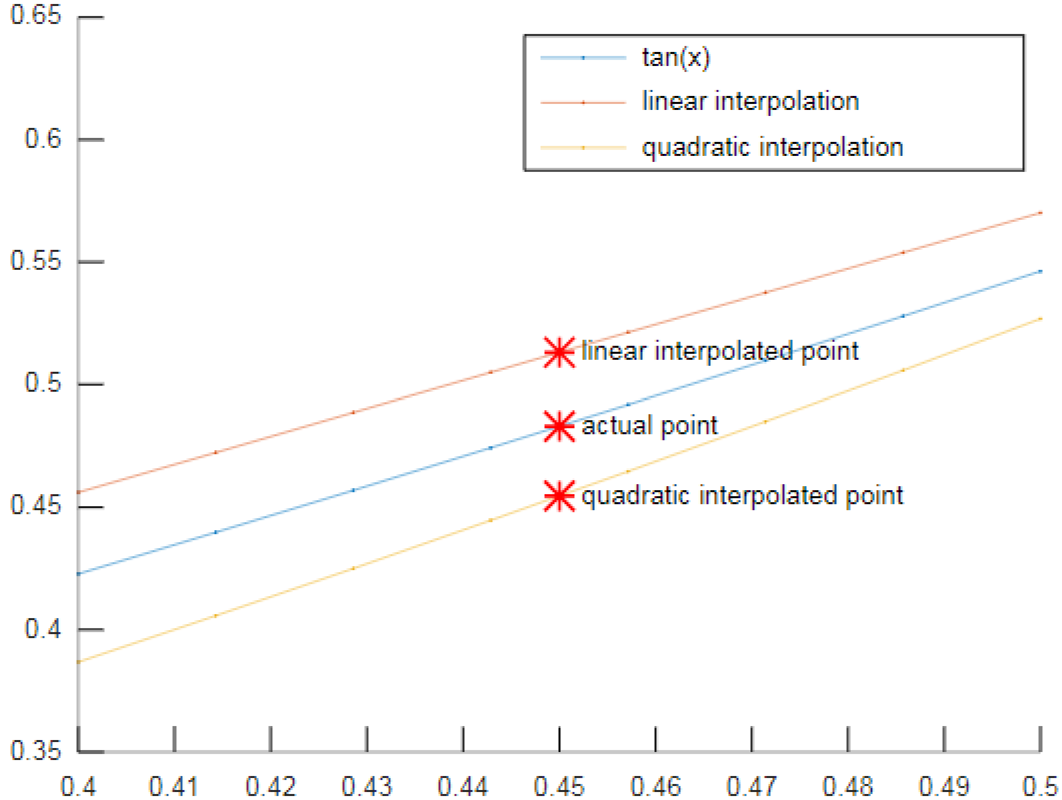
f(x) = sqrt(1+x). Actual: f(0.45) = 1.2042.
Linear Interpolation: P(0.45) = 1.1987. Relative Error: 0.0045347.
Quadratic Interpolation: P(0.45) = 1.2034. Relative Error: 0.00060174.

2.3 $f(x) = ln(x + 1)$



Program Output
f(x) = ln(x+1). Actual: f(0.45) = 0.37156.
Linear Interpolation: P(0.45) = 0.35251. Relative Error: 0.051287.
Quadratic Interpolation: P(0.45) = 0.3683. Relative Error: 0.0087941.

2.4 $f(x) = \tan(x)$



Program Output

f(x) = tan(x). Actual: f(0.45) = 0.48306.

Linear Interpolation: P(0.45) = 0.51312. Relative Error: 0.062229.

Quadratic Interpolation: P(0.45) = 0.45462. Relative Error: 0.058862.

3 Conclusions

For both interpolations of all equations, the error was less than 0.07 relative to the actual value. $f(x) = \sqrt{1+x}$ had the lowest relative errors for both interpolation methods: 0.0045347 for linear and 0.00060174 for quadratic. $f(x) = \tan(x)$ had the highest relative errors for both interpolation methods: 0.062229 for linear and 0.058862 for quadratic. In all equations, quadratic interpolation was more accurate than linear interpolation. I expect this since quadratic interpolation uses more data to interpolate than linear interpolation.

Appendix A proj3.m

```
x_interp = 0.45; %estimate f(x) at x = 0.45
xn1 = [0.0, 0.6]; %x coordinates for linear interpolation
xn2 = [0.0, 0.6, 0.9]; %x coordinates for quadratic interpolation

function do_work (func_name, func, x_interp, xn1, xn2)

  actual = func(x_interp);
  yn1 = func(xn1);
  yn2 = func(xn2);

  linear_interp_func = lagrange(1, x_interp, xn1, yn1);
  quadratic_interp_func = lagrange(2, x_interp, xn2, yn2);

  approx_deg1 = linear_interp_func(x_interp);
  approx_deg2 = quadratic_interp_func(x_interp);

  rel_error_deg1 = abs(approx_deg1 - actual)/actual;
  rel_error_deg2 = abs(approx_deg2 - actual)/actual;

  fprintf('f(x) = %s. Actual: f(%.2d) = %.5d.\n', func_name, x_interp, actual);
  fprintf('Linear Interpolation: P(%.2d) = %.5d. Relative Error: %.5d.\n', x_interp, appro
  fprintf('Quadratic Interpolation: P(%.2d) = %.5d. Relative Error: %.5d.\n\n', x_interp, a

  figure('name', func_name, 'NumberTitle','off');
  hold on

  fplot(func, [.4, .5]);
  fplot(linear_interp_func, [.4, .5]);
  fplot(quadratic_interp_func, [.4, .5]);

  plot(x_interp, actual, '*', 'MarkerSize', 5, 'MarkerEdgeColor', 'r');
  plot(x_interp, approx_deg1, '*','MarkerSize', 5, 'MarkerEdgeColor', 'r');
  plot(x_interp, approx_deg2, '*','MarkerSize', 5, 'MarkerEdgeColor', 'r');

  text(x_interp, actual, '   actual point', 'Interpreter', 'tex');
  text(x_interp, approx_deg1, '   linear interpolated point');
  text(x_interp, approx_deg2, '   quadratic interpolated point');

  legend(func_name, 'linear interpolation', 'quadratic interpolation');

endfunction

%f(x) = cos x
do_work('cos(x)', @(x) cos(x), x_interp, xn1, xn2);

%f(x) = sqrt(1+x)
do_work('sqrt(1+x)', @(x) sqrt(1+x), x_interp, xn1, xn2);

%f(x) = ln(x+1)
do_work('ln(x+1)', @(x) log(x+1), x_interp, xn1, xn2);

%f(x) = tan x
```

```
do_work('tan(x)', @(x) tan(x), x_interp, xn1, xn2);
```

Appendix B lagrange.m

```
%n is the degree of the Lagrange polynomials
%x_interp is the x value for interpolation
%xn is the array of data points
%yn is the array of function values at the data points
%build a string of the lagrange interpolation formula for the given degree and points
%convert the string to an anonymous function that can be graphed
function [interp_func] = lagrange(n, x_interp, xn, yn)

func_str = '@(x)';
for i=1:n+1
    func_str = strcat(func_str, num2str(yn(i)));
    for j=1:n+1
        if(i ~= j)
            func_str = strcat(func_str, '.*(x-', num2str(xn(j)), ').*', num2str(1/(xn(i)-xn
        endif
    endfor
    func_str = strcat(func_str, '+');
endfor

func_str = func_str(1:end-1);
interp_func = str2func(func_str);

endfunction
```