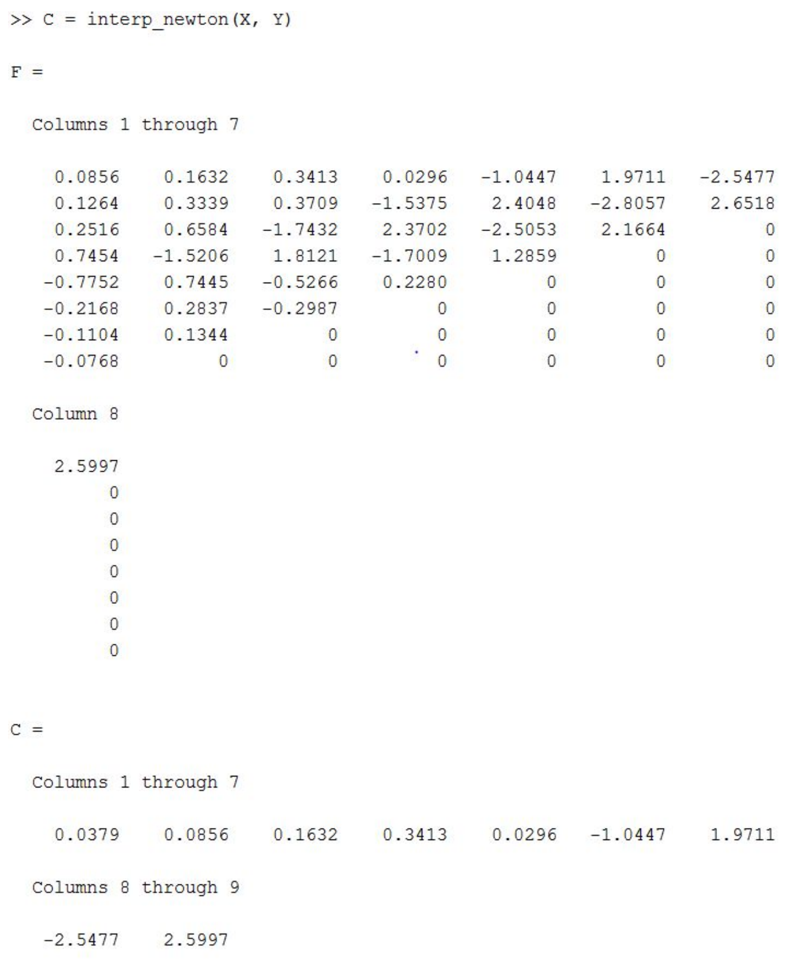
Problem Set 2: Polynomial Interpolation
Jack Weissenberger

1. The lagrange basis: Prove

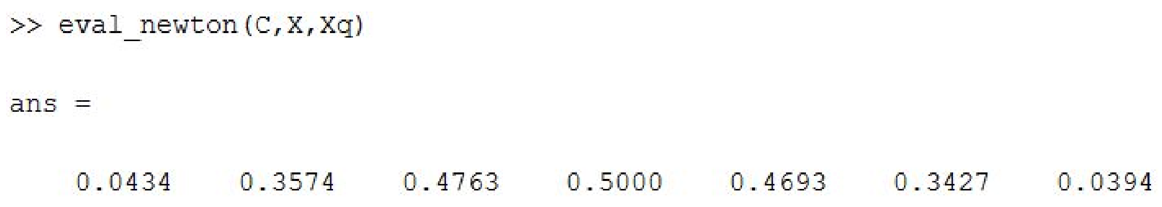
$$\sum_{i=0}^{n} L_k(x) = 1.$$

If x0, x1, ..., xn are n+1 distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial P(x) of degree of at most n exists. Because the interpolating polynomial is unique, if f(x) = 1, the interpolation would perfectly fit the function and therefore the sum of Lk must be equal to one.

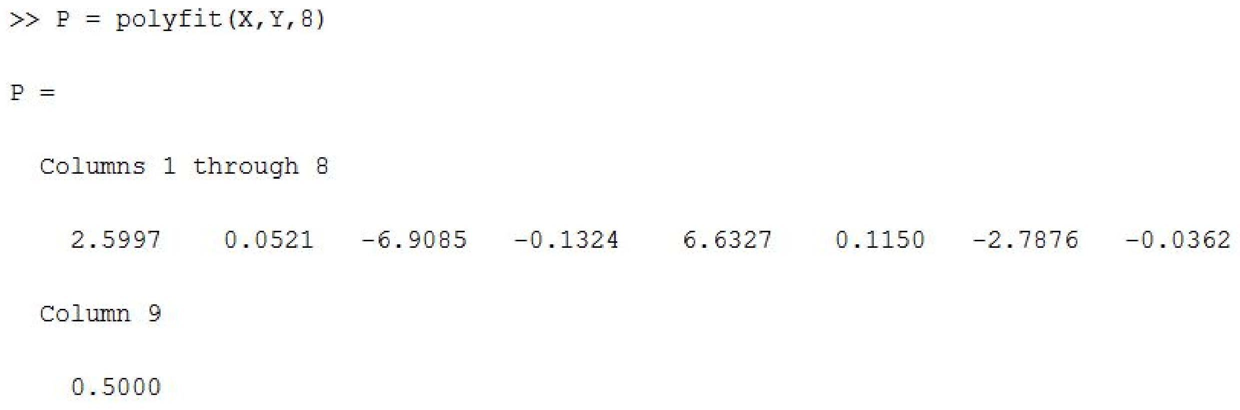
2. In my actual program I don't have F printing out, but wanted to show you my calculations. The vector C contains the coefficients of the polynomial after F



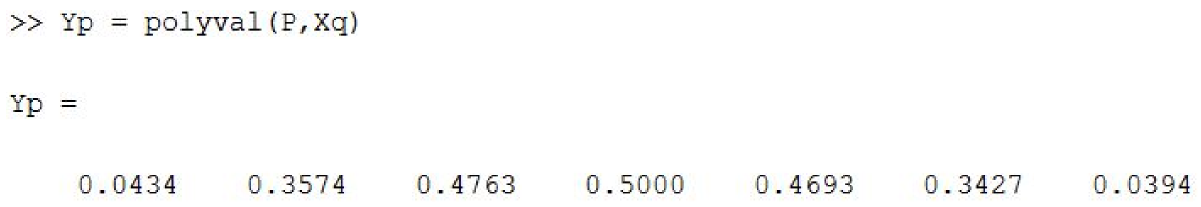
```
>> C = interp_newton(X, Y)

F =

  Columns 1 through 7

    0.0856    0.1632    0.3413    0.0296   -1.0447    1.9711   -2.5477
    0.1264    0.3339    0.3709   -1.5375    2.4048   -2.8057    2.6518
    0.2516    0.6584   -1.7432    2.3702   -2.5053    2.1664         0
    0.7454   -1.5206    1.8121   -1.7009    1.2859         0         0
   -0.7752    0.7445   -0.5266    0.2280         0         0         0
   -0.2168    0.2837   -0.2987         0         0         0         0
   -0.1104    0.1344         0         0         0         0         0
   -0.0768         0         0         0         0         0         0

  Column 8

    2.5997
         0
         0
         0
         0
         0
         0
         0


C =

  Columns 1 through 7

    0.0379    0.0856    0.1632    0.3413    0.0296   -1.0447    1.9711

  Columns 8 through 9

   -2.5477    2.5997
```



```
>> eval_newton(C,X,Xq)

ans =

    0.0434    0.3574    0.4763    0.5000    0.4693    0.3427    0.0394
```



```
>> P = polyfit(X,Y,8)

P =

  Columns 1 through 8

    2.5997    0.0521   -6.9085   -0.1324    6.6327    0.1150   -2.7876   -0.0362

  Column 9

    0.5000
```

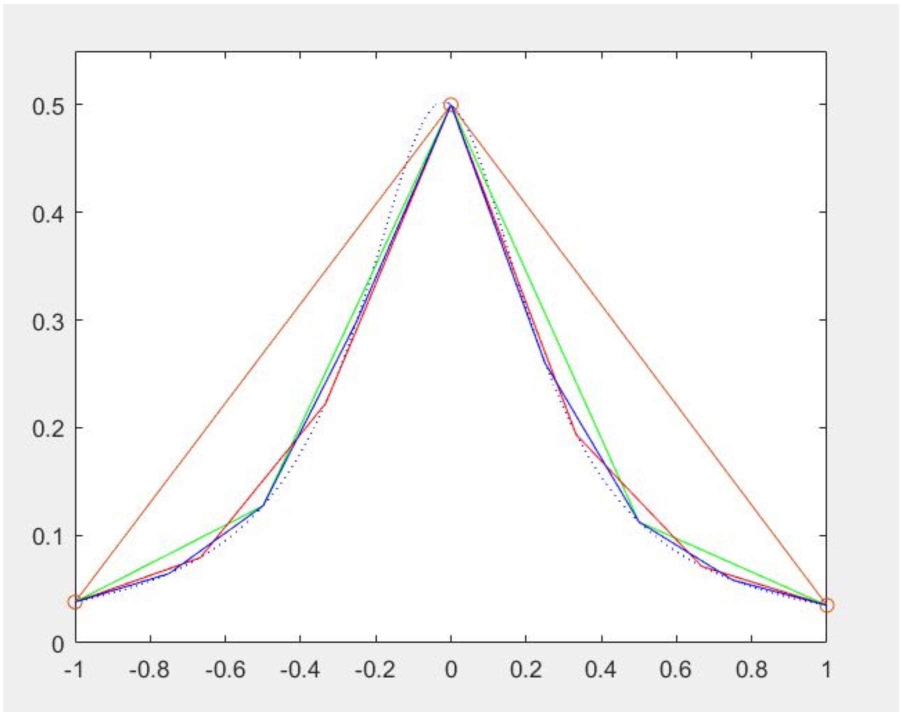


```
>> Yp = polyval(P,Xq)

Yp =

    0.0434    0.3574    0.4763    0.5000    0.4693    0.3427    0.0394
```

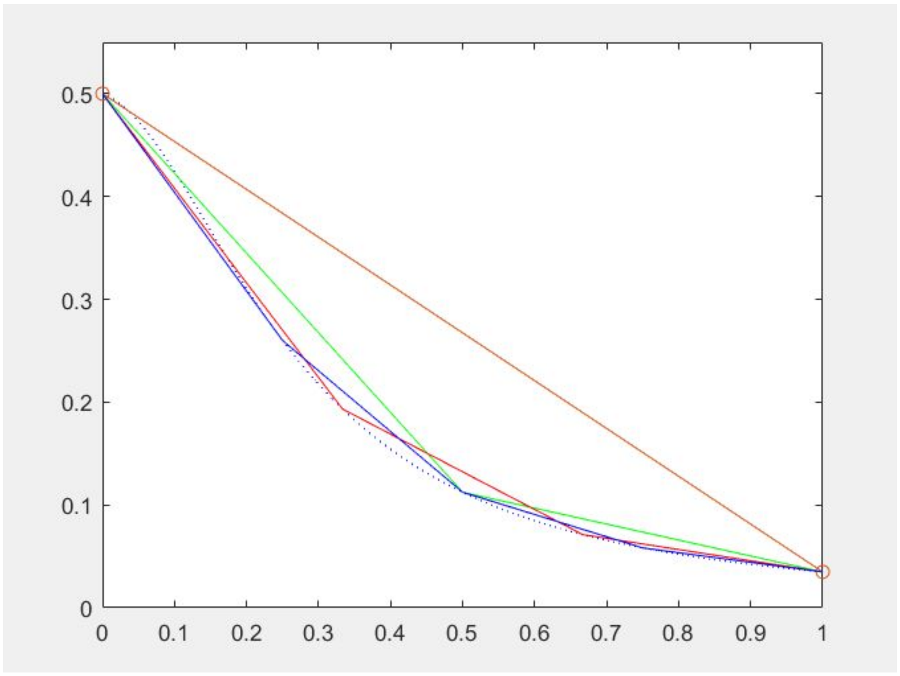
As you can see, the eval_newton and polyfit functions produce exactly the same interpolations when polyfit has a degree of 8. This proves that you can simplify the newton basis into the monomial basis.

3. Cubic Splines

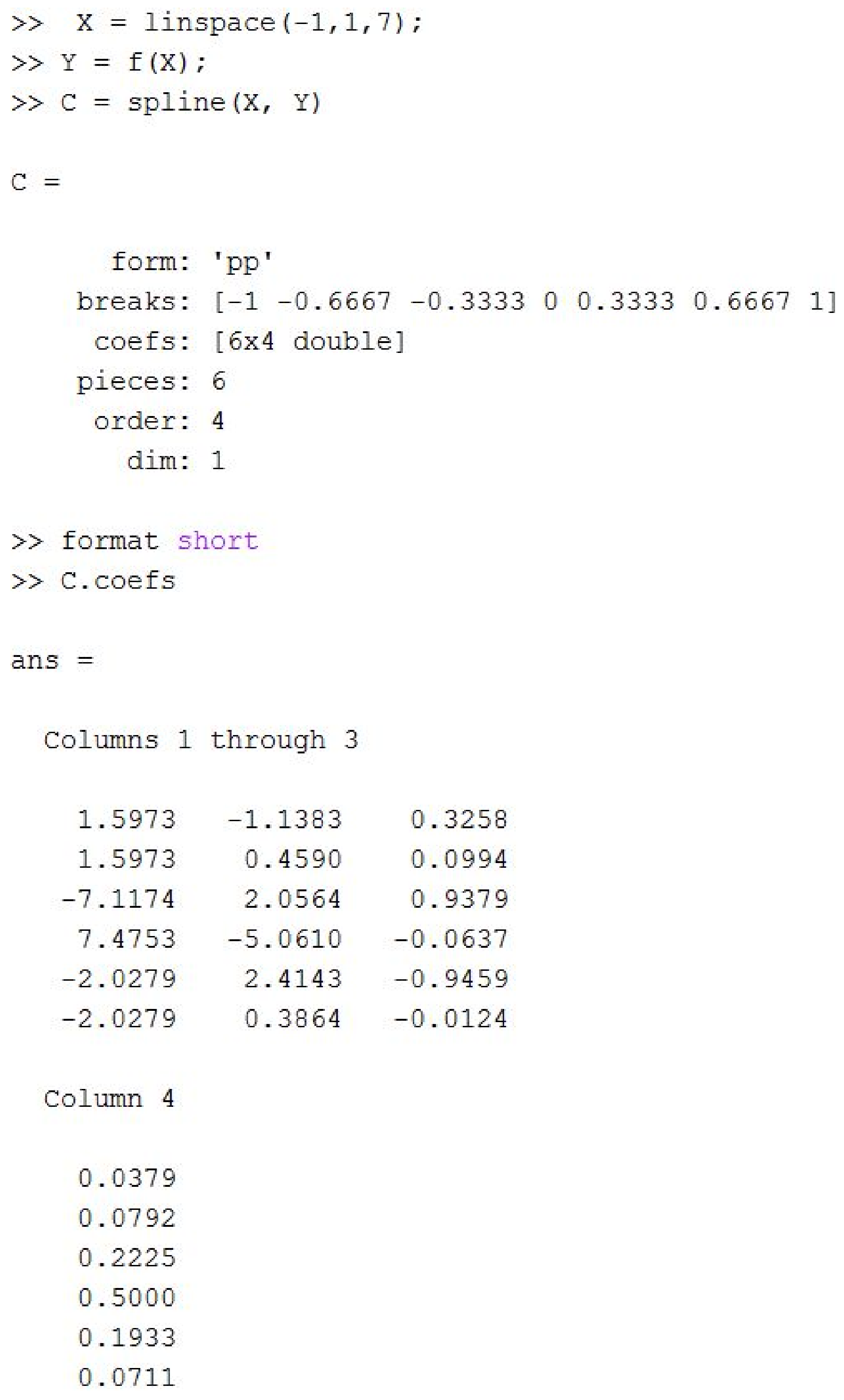
Here is the graph of the Cubic Spline of the function f(x) = 1/(1+exp(x)+25x^2)
The orange outside line is n = 3
The green is n = 5
The red is n= 7
The blue is n = 9
The dotted line is the actual function



This can be a little hard to see so here is the range from 0 1

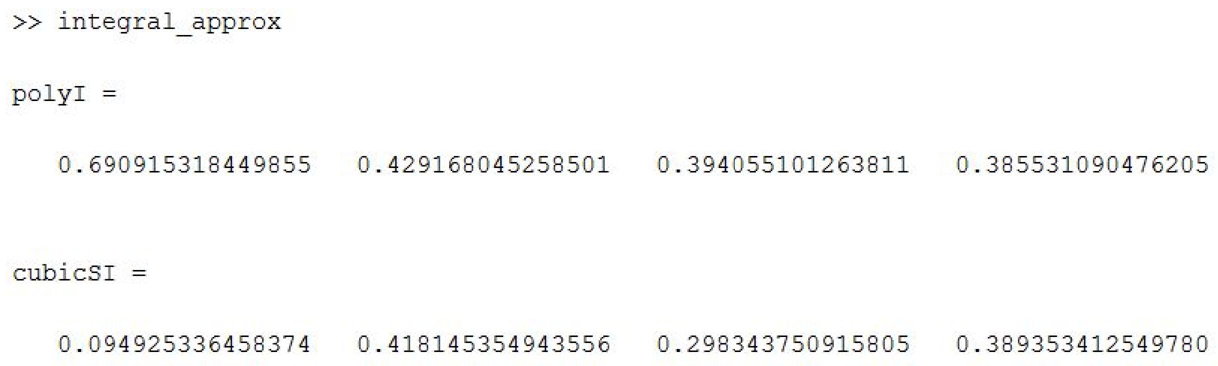


Here are the coefficients of the polynomials for n = 7



```
>>  X = linspace(-1,1,7);
>> Y = f(X);
>> C = spline(X, Y)

C = 

      form: 'pp'
    breaks: [-1 -0.6667 -0.3333 0 0.3333 0.6667 1]
     coefs: [6x4 double]
    pieces: 6
     order: 4
       dim: 1

>> format short
>> C.coefs

ans =

  Columns 1 through 3

    1.5973   -1.1383    0.3258
    1.5973    0.4590    0.0994
   -7.1174    2.0564    0.9379
    7.4753   -5.0610   -0.0637
   -2.0279    2.4143   -0.9459
   -2.0279    0.3864   -0.0124

  Column 4

    0.0379
    0.0792
    0.2225
    0.5000
    0.1933
    0.0711
```

4. Integral approx



```
>> integral_approx

polyI =

   0.690915318449855   0.429168045258501   0.394055101263811   0.385531090476205


cubicSI =

   0.094925336458374   0.418145354943556   0.298343750915805   0.389353412549780
```

These are the integral approximations from for the polynomial interpolation (polyI) and for cubic splines interpolation (cubicSI). The values from left to right are for n = 3, 5, 7, 9. I used the polyint() function to integrate the vectors containing the coefficients and then used diff(polyval()) to evaluate the integrals. I chose to use polyfit() to calculate the interpolating polynomial instead of my own functions interp_newton and eval_newton to calculate the interpolating polynomial because polyfit produces a polynomial in the monomial basis which is very easy to integrate over. Whereas the the newton functions produce a polynomial in the newton basis, which is not as straightforward to integrate over.

I trust the last two approximations from polynomial interpolation and only the last value from cubic splines. This is because there is still a relatively large difference between the last two values of cubic splines and the polynomial interpolation starts having a small difference between the interpolations in the last two points. Of these best approximations I only slightly trust the first two decimal places of the last approximations for both cubicSI and polyI. This is because they vary so much over the approximations. To gain more digits that I would trust I would need to increase n in the cubic spline interpolation to better fit the curve. This might not necessarily work for polynomial interpolation because high degree polynomials tend to overfit the function and provide bad approximations.