1 $\{(x_i, y_i)\}$

$= \{(0,3),(2,3),(3,5),(6,4)\}$

(a) $$p(x) = 3\cdot\frac{(x-2)(x-3)(x-6)}{(0-2)(0-3)(0-6)} + 3\cdot\frac{(x-0)(x-3)(x-6)}{(2-0)(2-3)(2-6)}$$

$$+ 5\cdot\frac{(x-0)(x-2)(x-6)}{(3-0)(3-2)(3-6)} + 4\cdot\frac{(x-0)(x-2)(x-3)}{(6-0)(6-2)(6-3)}$$

(b) $c_0 = \frac{3}{-36} = \frac{-1}{12}$, $c_1 = \frac{3}{2\cdot -1\cdot -4} = \frac{3}{8}$

$c_2 = \frac{5}{3\cdot 1\cdot -3} = -\frac{5}{9}$, $c_3 = \frac{4}{6\cdot 4\cdot 3} = \frac{1}{18}$

so

$$p(x) = \frac{-1}{12}(x-2)(x-3)(x-6) + \frac{3}{8}(x)(x-3)(x-6)$$

$$-\frac{5}{9}(x)(x-2)(x-6) + \frac{1}{18}(x)(x-2)(x-3)$$

(c) $$p(x) = \left(-\frac{x^3}{12} + \frac{11}{12}x^2 - 3x + 3\right) + \left(\frac{3}{8}x^3 - \frac{27}{8}x^2 + \frac{27}{4}x\right)$$

$$+\left(-\frac{5}{9}x^3 + \frac{40}{9}x^2 - \frac{20}{3}x\right) + \left(\frac{x^3}{18} - \frac{5}{18}x^2 + \frac{x}{3}\right)$$

$$= -\frac{5}{24}x^3 + \frac{41}{24}x^2 - \frac{31}{12}x + 3$$

(d) See screenshots.

(e) See screenshots.
The two curves coincide.

Question 1d and e

Here is the code from class on Day 13, which uses the monomial basis:

```
x = np.array([2, 8, 54, 15])  # data points to interpolate
y = np.array([3,7,-1,2])
nplus1 = len(x)

# construct the matrix A
A = np.empty((nplus1,nplus1))
for k in range(nplus1):
    A[:,k] = x**k
print(A)

# find the coeffs
c = np.linalg.solve(A,y)
print('c =',c)

# construct p
xx = np.linspace( x.min(), x.max(), 500 )  # evaluation points
p = np.zeros_like(xx)
for k in range(nplus1):
    p += c[k]*xx**k

plt.plot(x,y,'ro') # dots for the data points
plt.plot(xx,p);
plt.title('The interpolant to the $\{x_j,y_j\}$ built with the monomials $\{1,x,x^2,x^3\}$');
```

Here is the modifcation of the above to use the Newton basis:

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([(0,3), (2,3), (3,5), (6,4)])

x = data[:,0]  # data points to interpolate
y = data[:,1]
nplus1 = len(x)
n = nplus1 - 1

# construct the matrix A corresponding to the Newton basis
# Columns are the basis functions evaluated at the nodes, x
A = np.empty((nplus1,nplus1))
A[:,0] = 1
for k in range(n):
    A[:,k+1] = A[:,k]*(x-x[k])
```

different matrix

```
print('A:'); print(A)

# find the coeffs
c = np.linalg.solve(A,y)
print('c:'); print(c)

# construct p
xx = np.linspace( x.min(), x.max(), 500 )  # evaluation points
p = np.zeros_like(xx)
phi = np.ones_like(xx)
for k in range(nplus1):
    p += c[k]*phi
    phi *= (xx-x[k])
```

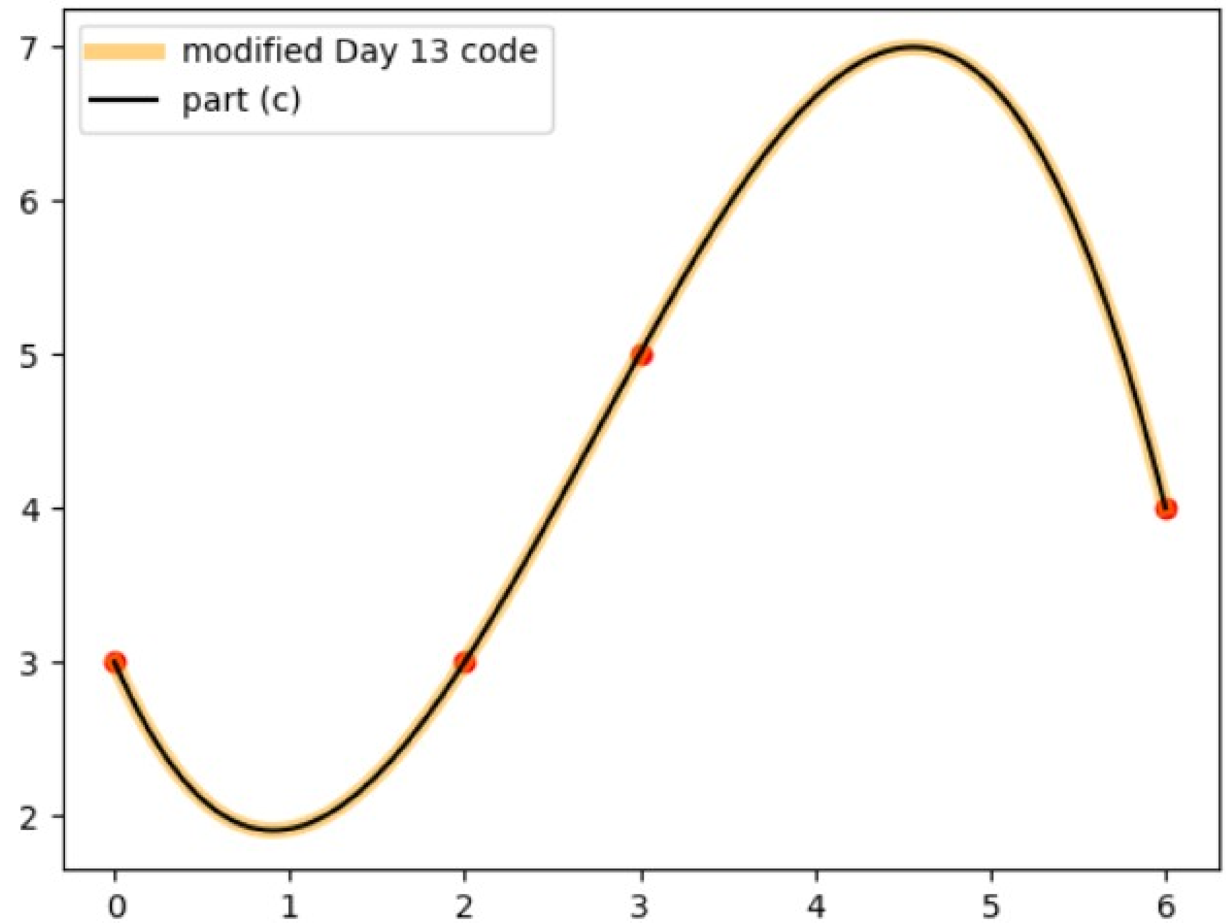
different basis functions

```
plt.plot(x,y,'ro') # dots for the data points
plt.plot(xx,p,'orange',lw=5,alpha=0.5,label='modified Day 13 code');

# finally add the graph of the result of the hand-calculation in part (c):
plt.plot(xx,-5/24*xx**3 + 41/24*xx**2 - 31/12*xx + 3,'k',label='part (c)')
```

from part (c)

```
plt.legend();
```

```
A:
[[ 1.  0. -0.  0.]
 [ 1.  2.  0. -0.]
 [ 1.  3.  3.  0.]
 [ 1.  6. 24. 72.]]
c:
[ 3.00000000e+00  2.96059473e-16  6.66666667e-01 -2.08333333e-01]
```



We visually confirm that the two ways of computing the interpolant give the same result.

2

THEOREM 4.6

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct points in $[a,b]$ and $f \in C^{n+1}[a,b]$, then for each $x \in [a,b]$, there exists a number $\xi = \xi(x) \in (a,b)$ such that

$$f(x) = p(x) + \frac{f^{(n+1)}(\xi(x)) \prod_{j=0}^{n} (x - x_j)}{(n+1)!} \tag{4.11}$$

Proof:

Case 1 Suppose that $x = x_k$ for some k, $0 \le k \le n$. Then, $f(x_k) = p(x_k)$ and (4.11) is satisfied for $\xi(x_k)$ arbitrary on (a,b).

First dispensing with the easy cases when x happens to be a node of the interpolation.
There by definition p(x)=f(x) and the product in 4.11 has a zero factor giving agreement.

Case 2 Suppose that $x \ne x_k$ for $k = 0, 1, 2, \ldots n$. Define as a function of t,

$$g(t) = f(t) - p(t) - (f(x) - p(x)) \prod_{i=0}^{n} \frac{t - x_i}{x - x_i}.$$

Off the bat I have no idea where this is going, but we can certainly define this function, g.
Let's keep in mind that we have fixed x in the definition of g. For a different x,
we have a different g.

Since $f \in C^{n+1}[a,b]$, $p \in C^{\infty}[a,b]$, and $x \ne x_i$ for any i, it follows that $g \in C^{n+1}[a,b]$.

because a sum or product or quotient of C^m functions is a C^m function if there's no zero in the denominator - which there isn't because we're in Case 2.

For $t = x_k$,

$$g(x_k) = f(x_k) - p(x_k) - (f(x) - p(x)) \prod_{i=0}^{n} \frac{x_k - x_i}{x - x_i} = 0 - 0 = 0.$$

Again, not sure where we're going with this, but this is true because f(x_k) = p(x_k) and for any k there is a zero factor in the product.

Moreover,

$$g(x) = f(x) - p(x) - (f(x) - p(x)) \prod_{i=0}^{n} \frac{x - x_i}{x - x_i} = 0.$$

because all the factors in the product are 1.

Thus, $g \in C^{n+1}[a,b]$ vanishes at the $(n+2)$ points $x_0, x_1, \ldots, x_n, x$. If $g(x_0) = 0$ and $g(x_1) = 0$, there is an x_0^*, $x_0 < x_0^* < x_1$, such that $g'(x_0^*) = 0$. (This is

by Rolle's theorem because g is in C^1[a,b].
Similarly there's an x1* in (x1,x2) such that g'(x1*)=0, etc.,
and we have (n+1) points where g' vanishes.
Recursively, we are guaranteed g'' is zero at at least n points, g''' ant at least n-1 points,
and finally g^(n+1) is zero at at least (n+2)-(n+1) = 1 point in (a,b) - call it $\xi = \xi(x)$.

Evaluating $g^{(n+1)}(t)$ at ξ gives

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - p^{(n+1)}(\xi) - [f(x) - p(x)] \left(\frac{d^{n+1}}{dt^{n+1}} \prod_{i=0}^{n} \frac{t - x_i}{x - x_i} \right)_{t=\xi}, \qquad (*)$$

just differentiating the definition of g (n+1) times and evaluating at \xi.

so

$$0 = f^{(n+1)}(\xi) - (f(x) - p(x)) \frac{(n+1)!}{\prod_{i=0}^{n}(x - x_i)}. \qquad (**)$$

(**) follows from (*) because
i. the (n+1)th derivative of a degree-n polynomial such as p is zero, and
ii. the denominators of the factors in the products can be brought outside the differentiation, and the (n+1)th derivative with respect to t of $(t - x_0)(t - x_1)\ldots(t - x_n)$

is (n+1)!

Therefore,

$$f(x) = p(x) + \frac{f^{(n+1)}(\xi(x)) \prod_{i=0}^{n}(x - x_i)}{(n+1)!}.$$

which is our goal (4.11), by just solving (**) for f(x).

3

$$\left(-1, \tfrac{1}{e}\right), \ (0,1), \ (1,e)$$

Using the Lagrange form,

$$p(x) = \frac{1}{e}\frac{(x-0)(x-1)}{(-1-0)(-1-1)} + 1\,\frac{(x-(-1))(x-1)}{(0-(-1))(0-1)} + e\,\frac{(x-(-1))(x-0)}{(1-(-1))(1-0)}$$

$$= \frac{1}{e}\frac{x(x-1)}{2} + \frac{(x+1)(x-1)}{-1} + e\,\frac{(x+1)\cdot x}{2}$$

and

$$p\left(\tfrac{1}{2}\right) = \frac{1}{e}\frac{\left(\frac{1}{2}\right)\left(\frac{-1}{2}\right)}{2} + \frac{\frac{3}{2}\cdot\left(\frac{-1}{2}\right)}{-1} + e\,\frac{\left(\frac{3}{2}\right)\left(\frac{1}{2}\right)}{2}$$

$$= \frac{-1}{8e} + \frac{3}{4} + \frac{3}{8}e = 1.7233707555\ldots$$

Now, actually, $\sqrt{e} = 1.6487212707\ldots$

so error $= p\left(\tfrac{1}{2}\right) - \sqrt{e} = \boxed{0.0746494848\ldots}$

error.

Theorem says

$$|\text{error}| \le \left|\overbrace{\left(\tfrac{1}{2}+1\right)\cdot\tfrac{1}{2}\cdot\left(\tfrac{1}{2}-1\right)}^{-3/8}\right| \cdot \left|\frac{f^{(3)}(c)}{3!}\right| \quad \text{for some } c \in [-1,1]$$

Here $f(x) = e^x$, $f'(x) = e^x$, $f''(x) = e^x$, $f'''(x) = e^x$

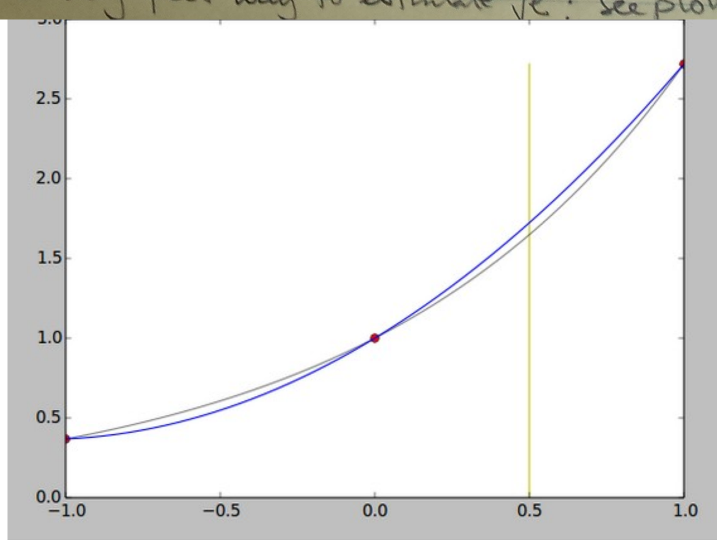
So $\displaystyle\max_{x\in[-1,1]} |f'''(x)| = \max_{x\in[-1,1]} e^x = e.$

and $\displaystyle |\text{error}| \le \frac{3}{8}\frac{e}{6} = \frac{e}{16} = \boxed{0.169892614\ldots}$

error bound.

The error is within the bound $(0.0747 < 0.16)$.

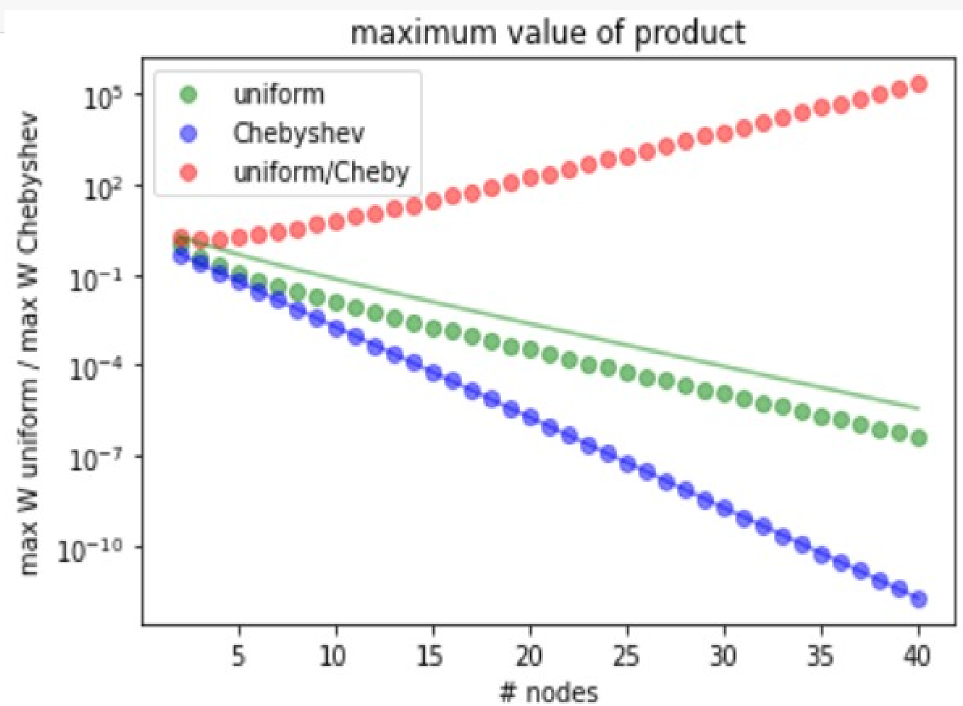
(BTW, this is a very poor way to estimate $\sqrt{e}$: see plot.)



4

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1.,1.,2000)

print( ' n    maxu     maxc   maxu/maxc' )

ulabel,clabel,rlabel = 'uniform','Chebyshev','uniform/Cheby'
for n in range(2,41):

    Xu = np.linspace(-1.,1.,n)              # uniformly spaced nodes
    Xc = np.cos( (2*np.arange(n)+1)/n*np.pi/2 ) # Chebyshev nodes

    # evaluate the products for both node choices
    Pu = np.ones_like(x)
    Pc = np.ones_like(x)
    for k in range(n):
        Pu *= x-Xu[k]
        Pc *= x-Xc[k]

    maxu = np.abs(Pu).max()
    maxc = np.abs(Pc).max()

    print( '{:2d} {:8.1e} {:8.1e} {:8.1e}'.format( n, maxu, maxc, maxu/maxc ) )
    #print abs(Pc2(x)).max(), 2.**-(n-1)
    plt.semilogy(n,maxu,'go',alpha=0.5,label=ulabel)
    plt.semilogy(n,maxc,'bo',alpha=0.5,label=clabel)
    plt.semilogy(n,maxu/maxc,'ro',alpha=0.5,label=rlabel)
    ulabel,clabel,rlabel = None,None,None # so we only have one entry for each in legend
nvalues = np.arange(2,41)
plt.semilogy(nvalues,2.**(-(nvalues-1)),'b',alpha=0.5)    # we've read that it's 2^-(n-1) for Chebyshev
guess = [np.prod([i/n for i in range(1,n+1)])*2**(n+1)/n for n in range(2,41)] # a bound for functional form for uniform nodes
plt.semilogy(nvalues,guess,'g',alpha=0.5)
plt.xlabel('# nodes')
plt.ylabel('max W uniform / max W Chebyshev')
plt.title('maximum value of product')
plt.legend();
```

```
 n    maxu     maxc   maxu/maxc
 2  1.0e+00  5.0e-01  2.0e+00
 3  3.8e-01  2.5e-01  1.5e+00
 4  2.0e-01  1.3e-01  1.6e+00
 5  1.1e-01  6.3e-02  1.8e+00
 6  6.9e-02  3.1e-02  2.2e+00
 7  4.4e-02  1.6e-02  2.8e+00
 8  2.8e-02  7.8e-03  3.6e+00
 9  1.9e-02  3.9e-03  4.8e+00
10  1.3e-02  2.0e-03  6.5e+00
11  8.5e-03  9.8e-04  8.7e+00
12  5.8e-03  4.9e-04  1.2e+01
13  4.0e-03  2.4e-04  1.6e+01
14  2.8e-03  1.2e-04  2.3e+01
15  1.9e-03  6.1e-05  3.2e+01
16  1.3e-03  3.1e-05  4.4e+01
17  9.4e-04  1.5e-05  6.2e+01
18  6.6e-04  7.6e-06  8.7e+01
19  4.7e-04  3.8e-06  1.2e+02
20  3.3e-04  1.9e-06  1.7e+02
21  2.3e-04  9.5e-07  2.4e+02
22  1.7e-04  4.8e-07  3.5e+02
23  1.2e-04  2.4e-07  4.9e+02
24  8.4e-05  1.2e-07  7.0e+02
25  6.0e-05  6.0e-08  1.0e+03
26  4.3e-05  3.0e-08  1.4e+03
27  3.1e-05  1.5e-08  2.0e+03
28  2.2e-05  7.5e-09  2.9e+03
29  1.6e-05  3.7e-09  4.2e+03
30  1.1e-05  1.9e-09  6.0e+03
31  8.1e-06  9.3e-10  8.7e+03
32  5.8e-06  4.7e-10  1.2e+04
33  4.2e-06  2.3e-10  1.8e+04
34  3.0e-06  1.2e-10  2.6e+04
35  2.2e-06  5.8e-11  3.7e+04
36  1.6e-06  2.9e-11  5.3e+04
37  1.1e-06  1.5e-11  7.7e+04
38  8.1e-07  7.3e-12  1.1e+05
39  5.8e-07  3.6e-12  1.6e+05
40  4.2e-07  1.8e-12  2.3e+05
```



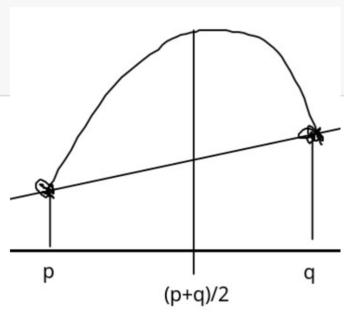
From the graphs, it appears that the inf norm of the product goes to zero exponentially with n for both uniform and Chebyshev nodes, but faster for Chebyshev nodes, so that the latter are exponentially better than uniform nodes. A straightforwardly obtained bound for uniform nodes is shown as the green curve.

5

5(a)

Answers to this were extraordinarily more complicated than what I had in mind. Here's all I meant: a quadratic function realizes the bound of Theorem 4.13 in that the maximum deviation exactly equals the given bound with h = width of subinterval.

```
from resources306 import *
x,a,b,c,p,q = sp.symbols('x,a,b,c,p,q',real=True)
f = a*x**2 + b*x + c
l = (q-x)/(q-p)*f.subs({x:p}) + (x-p)/(q-p)*f.subs({x:q})  # linear interpolant with nodes at x = p,q
dev = l - f
ddev = sp.diff(dev,x)
xmaxdev = sp.solve(ddev,x)[0]
print('Location of extreme deviation:')
display(xmaxdev)

maxabsdev = sp.Abs(sp.factor(sp.simplify(dev.subs({x:xmaxdev}))))
print('abs extreme deviation:')
display(maxabsdev)

D2f = sp.diff(f,x,x)
print('f'':')
display(D2f) # it's constant: it's its own max

theorem_4_13_bound = sp.simplify( sp.Abs((q-p)**2/8*D2f) )
print('Theorem 4.13 bound:')
display( theorem_4_13_bound )
```




```
Location of extreme deviation:
```

$$\frac{p}{2} + \frac{q}{2}$$

```
abs extreme deviation:
```

$$\frac{(p-q)^2\,|a|}{4}$$

f''

$$2a$$

```
Theorem 4.13 bound:
```

$$\frac{(p-q)^2\,|a|}{4}$$

(b)

From part (a) we know that for a quadratic $f(x) = ax^2 + bx + c$ on an interval $[p, q]$ of width $h = q - p$, the maximum absolute deviation is $ah^2/4$. Note that this is independent of both b and c.

On the other hand, $f'(x) = 2ax + b$, a linear function, so its extremes on $[p, q]$ are at the ends, p and q. That is $||Df||_\infty = \max\{|ap + b|, |aq + b|\}$ where "max" means the larger of the two numbers.

Since b and hence $||Df||_\infty$ can be made arbitarily large without affecting the maximum deviation, we conclude that there are quadratics that hugely fail to attain the bound $\frac{1}{2}h||Df||_\infty$ of Remark 4.33 (p242).

Moreover, the following experiment strongly suggests that a sharp bound for **quadratic** f is in fact $\frac{1}{2}$ of the bound in the Remark.

```
# Empirical test of sharpness (or not) of Remark 4.33 inequality.
# compute the ratio LHS / RHS for a large number of (quadratic) examples
n = 100000
a,b = np.random.randn(2,n)
p,q = -1,1  # choices don't actually matter
maxabsdf = np.maximum( np.abs(2*a*p + b), np.abs(2*a*q + b) )
bound = np.abs(p-q)/2*maxabsdf
maxabsdev = (p-q)**2/4*np.abs(a)
ratio = maxabsdev/bound
ratio.max()
```

```
0.49999922976923794
```

And I cannot think of any (non-quadratic) function that attains the bound in the Remark.

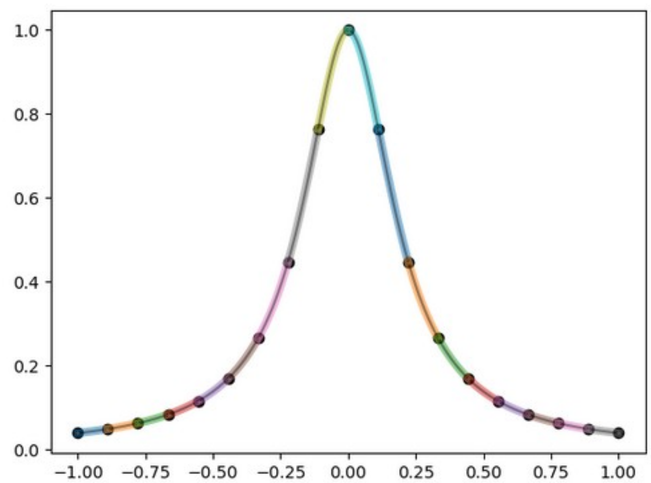
6. Cubic spline interpolation with "natural" end conditions

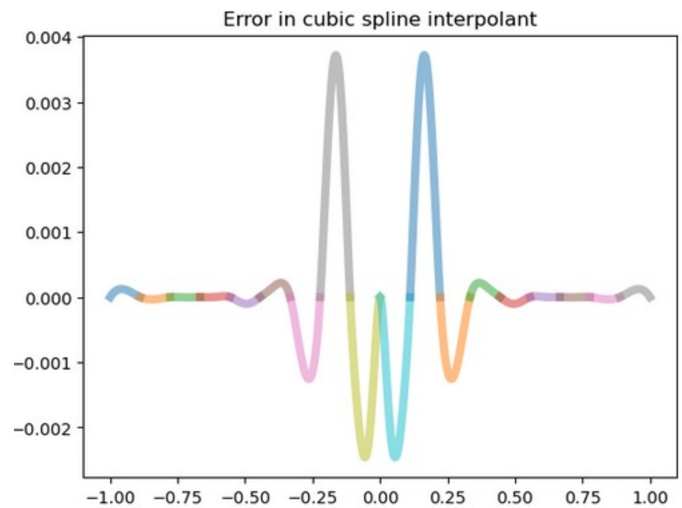
Code exactly as in class except that the data “y” is samples of Runge function, and the number of nodes is 19:

```
import numpy as np
import matplotlib.pyplot as plt
np.set_printoptions(linewidth=300)

a = -1
b = 1
nplus1 = 19  # number of nodes

n = nplus1 - 1
print(n,'subintervals')

x = np.linspace(a,b,nplus1)
xsq  = x**2  # x^2 and x^3 for convenience
xcub = x**3

def runge(x): return 1/(1+25*x**2) # Runge function
y = runge(x)

# we have n subintervals, and on each a 4-dof cubic
function

A = np.zeros( (4*n,4*n) )

for letter in 'abcd':
    for j in range(n):
        print('\t'+letter+str(j),end='')
rhs = np.zeros( 4*n )
row = 0

# left endpoint interpolation
for j in range(n):
    rhs[ row ] = y[j]
    A[ row, j ]     = 1
    A[ row, j + 1*n ] = x[j]
    A[ row, j + 2*n ] = xsq[j]
    A[ row, j + 3*n ] = xcub[j]
    row += 1

# right endpoint interpolation
for j in range(n):
    rhs[ row ] = y[j+1]
    A[ row, j       ] = 1
    A[ row, j + 1*n ] = x[j+1]
    A[ row, j + 2*n ] = xsq[j+1]
    A[ row, j + 3*n ] = xcub[j+1]
    row += 1

# continuous derivative at interior nodes
for j in range(n-1):
    rhs[ row ] = 0
    A[ row, j + 1*n ] = 1
    A[ row, j + 2*n ] = 2*x [j+1]
    A[ row, j + 3*n ] = 3*xsq[j+1]
    A[ row, j + 1*n + 1 ] = -1
    A[ row, j + 2*n + 1 ] = -2*x [j+1]
    A[ row, j + 3*n + 1 ] = -3*xsq[j+1]

    row += 1

# continuous 2nd derivative at interior nodes
for j in range(n-1):
    rhs[ row ] = 0
    A[ row, j + 2*n ] = 2
    A[ row, j + 3*n ] = 6*x[j+1]

    A[ row, j + 2*n + 1 ] = -2
    A[ row, j + 3*n + 1 ] = -6*x[j+1]

    row += 1

# two more conditions
# example: second derivative zero at ends

j = 0
rhs[row] = 0
A[ row, j + 2*n ] = 2
A[ row, j + 3*n ] = 6*x[j]
row += 1

j = n-1
rhs[row] = 0
A[ row, j + 2*n ] = 2
A[ row, j + 3*n ] = 6*x[j+1]
row += 1


abcd = np.linalg.solve(A,rhs)
print()
print(abcd)
def plot(x,y,abcd):
    #plt.figure(figsize=(10,1))
    #plt.plot(x,y,'yo',markersize=30,clip_on=False)
    plt.plot(x,y,'ko',clip_on=False)
    nplus1 = len(x)
    n = nplus1 - 1
    for j in range(n):
        xx = np.linspace(x[j],x[j+1],100)
        a,b,c,d = abcd[[j,j+n,j+2*n,j+3*n]]
        plt.plot(xx, a + b*xx + c*xx**2 +
d*xx**3,lw=5,alpha=0.5)
        plt.plot(xx,runge(xx),'k',alpha=0.4)
    plt.plot(x,y,'wo',markersize=0.2,clip_on=False)
#plt.figure(figsize=(10,2))
plot(x,y,abcd)
plt.savefig('temp.svg')
```

Below left is the interpolant, and on the right is the deviation from the Runge function, where we can see the maximum deviation is less than 0.004. From the illustrations in the question, we can see this is even better than the global polynomial interpolant using Chebyshev nodes, which has a maximum error more than 4 times greater.

If we also care about error in the derivative, the cubic spline is better in that regard too. (Chebyshev error plot has steeper slopes.)





7.

Approximate quarter circle with a Bezier cubic curve

```python
from nsm import *

def bezier(P,t):  # columns of P are P0, P1, P2, P3
    P0,P1,P2,P3 = P.T
    s = 1 - t
    x =  s**3*P0[0] + 3*s**2*t*P1[0] + 3*s*t**2*P2[0] + t**3*P3[0]
    y =  s**3*P0[1] + 3*s**2*t*P1[1] + 3*s*t**2*P2[1] + t**3*P3[1]
    return x,y

def deviation_from_circle(x,y):
    dev = x**2 + y**2 - 1
    return dev.min(),dev.max()

def doit(qmin,qmax,nq,draw_control_points=True):
    fig, (ax0, ax1) = plt.subplots(1, 2, gridspec_kw={'width_ratios': [5, 2]})
    for q in np.linspace(qmin,qmax,nq):
        P = np.array([[1,1,q,0],
                      [0,q,1,1]]) # columns of P are P0, P1, P2, P3
        ax0.set_aspect(1)
        t = np.linspace(0,1,500)
        if draw_control_points:
            ax0.plot(P[0,:],P[1,:],'go',alpha=0.75)
            ax0.plot(P[0,:],P[1,:],'m-',alpha=0.5)
        x,y = bezier(P,t)
        ax0.plot( x, y,'b-')
        ax0.plot(-x, y,'b-')
        ax0.plot(-x,-y,'b-')
        ax0.plot( x,-y,'b-')

        ax1.plot(t,x**2+y**2,'-',label='{:6.4f}'.format(q))
        ax1.set_title('radius'); ax1.set_xlabel('t')
        print('q: {:6.4f}  deviation range {:9.5f} {:9.5f}'.format(q,*deviation_from_circle(x,y)))
    ax1.legend();
    fig.tight_layout()

```

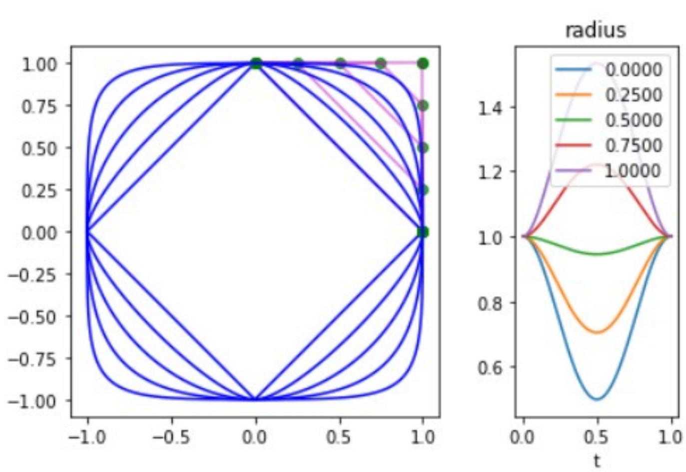
My criterion

My goal is to minimize the maximum deviation from the circle, $\max_{t\in[0,1]} |x(t)^2 + y(t)^2 - 1|$.

For each round of the process, I'm going to try 5 values of the free parameter, and zero in on the optimal value.

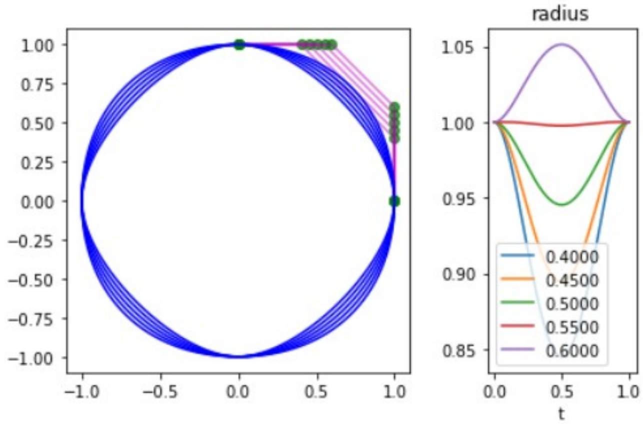
```python
doit(0,1,5)
```

```
q: 0.0000  deviation range  -0.50000   0.00000
q: 0.2500  deviation range  -0.29492   0.00000
q: 0.5000  deviation range  -0.05469   0.00000
q: 0.7500  deviation range   0.00000   0.22070
q: 1.0000  deviation range   0.00000   0.53125
```



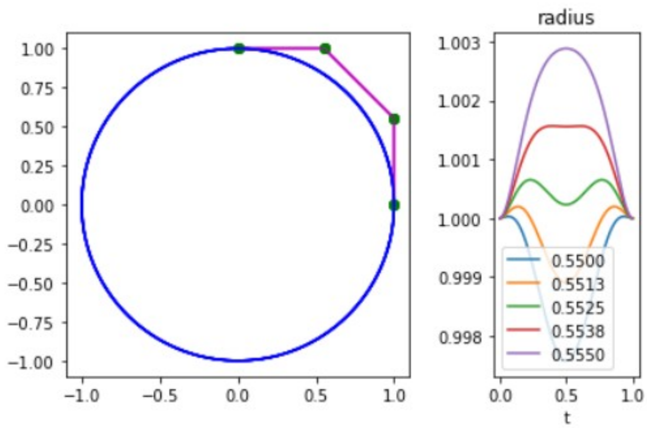
```
doit(.4,.6,5)
```

```
q: 0.4000  deviation range  -0.15500   0.00000
q: 0.4500  deviation range  -0.10555   0.00000
q: 0.5000  deviation range  -0.05469   0.00000
q: 0.5500  deviation range  -0.00242   0.00003
q: 0.6000  deviation range   0.00000   0.05125
```




```
doit(.55,.555,5)
```

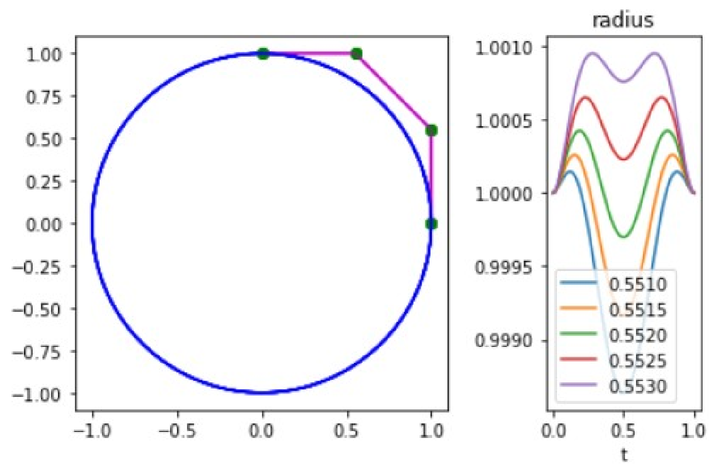
```
q: 0.5500  deviation range  -0.00242   0.00003
q: 0.5513  deviation range  -0.00110   0.00020
q: 0.5525  deviation range   0.00000   0.00065
q: 0.5538  deviation range   0.00000   0.00157
q: 0.5550  deviation range   0.00000   0.00288
```



Could be improved a little bit further, but I'll stop here.
The deviation from a perfect circle when “q”=0.552 is less than 0.025%. (r = 1+eps, r^2 - 1 ~ 2eps.)

```
doit(.551,.553,5)
```

```
q: 0.5510  deviation range  -0.00136   0.00014
q: 0.5515  deviation range  -0.00083   0.00026
q: 0.5520  deviation range  -0.00030   0.00042
q: 0.5525  deviation range   0.00000   0.00065
q: 0.5530  deviation range   0.00000   0.00095
```




```
doit(.552,.552,1,False)
```

```
q: 0.5520  deviation range  -0.00030   0.00042
```