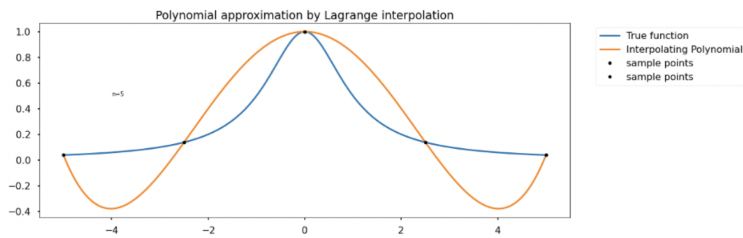
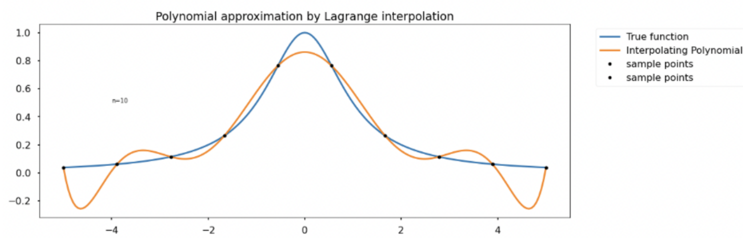
# APPM4650 Homework7

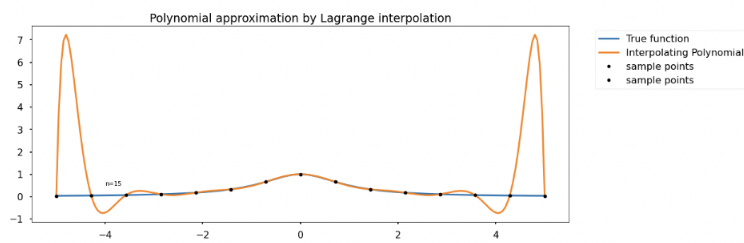
Olivia Golden

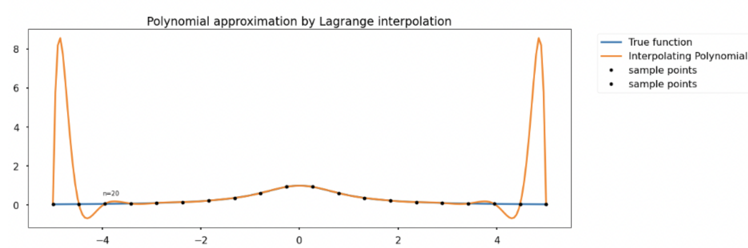
October 15, 2021

1. (a)

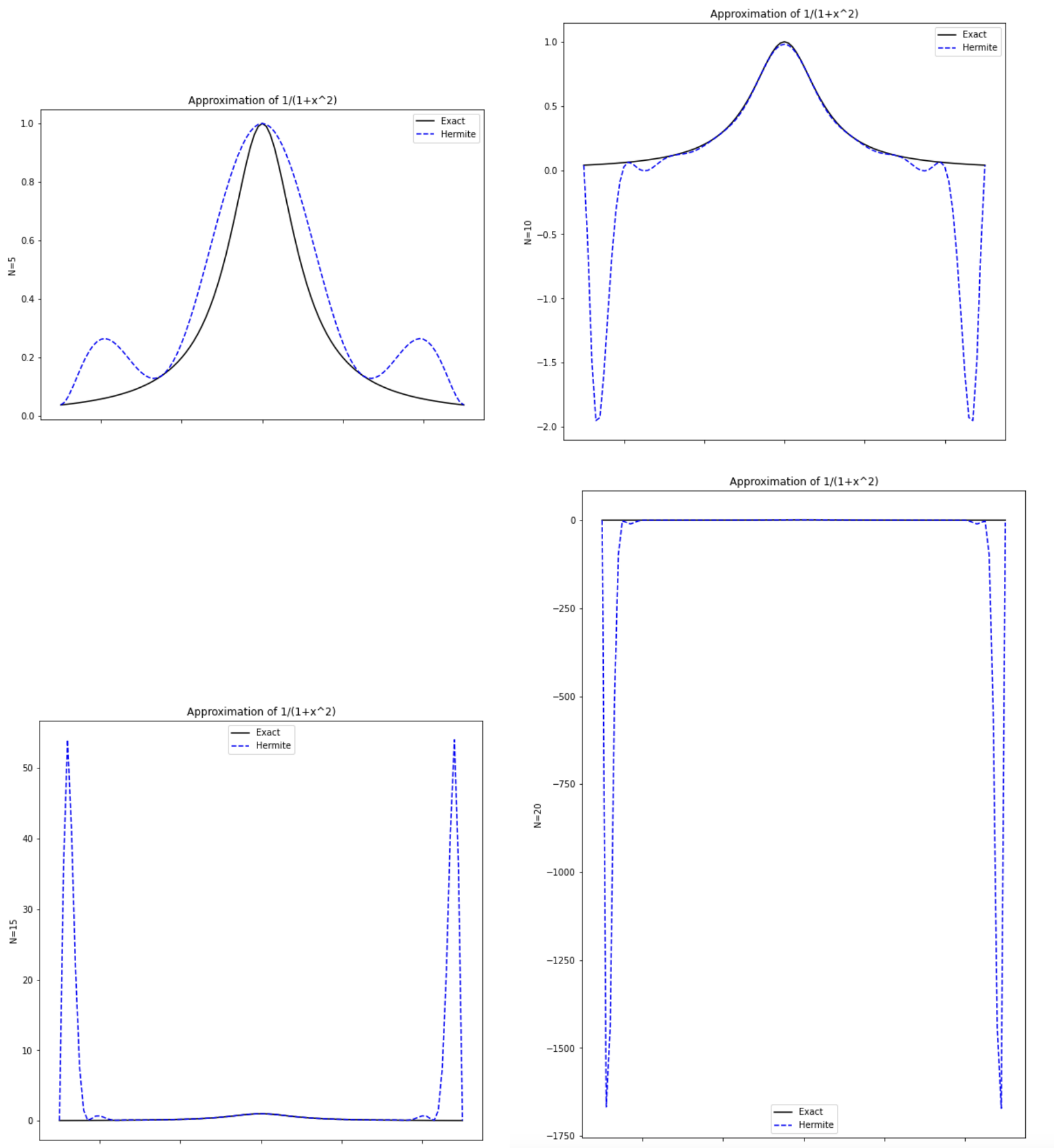


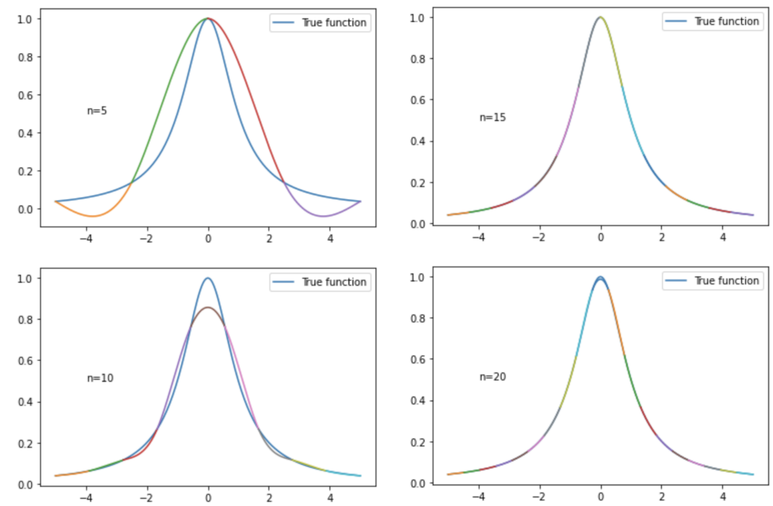



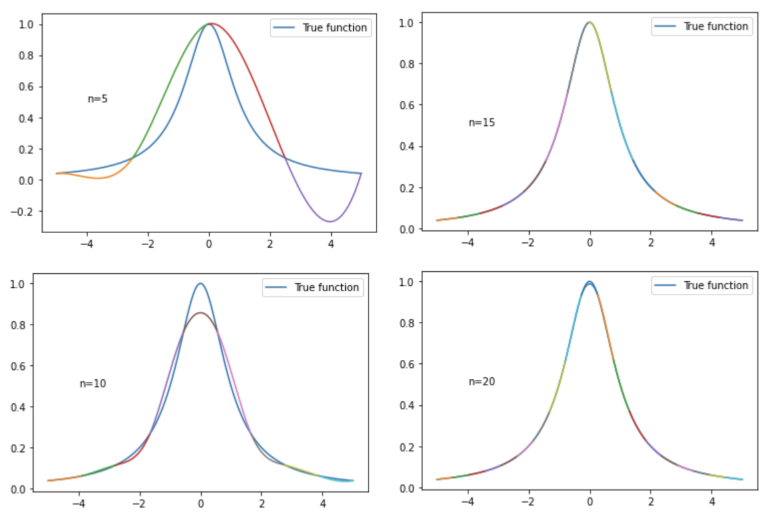
(b)



Approximation of 1/(1+x^2)
Exact
Hermite
N=5
Approximation of 1/(1+x^2)
Exact
Hermite
N=10
Approximation of 1/(1+x^2)
Exact
Hermite
N=15
Approximation of 1/(1+x^2)
Exact
Hermite
N=20



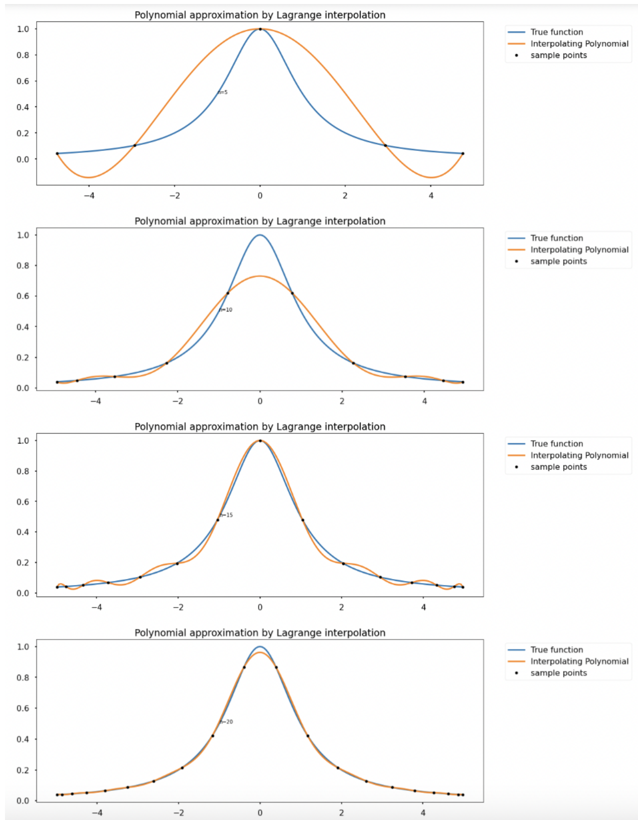

(c)




(d)

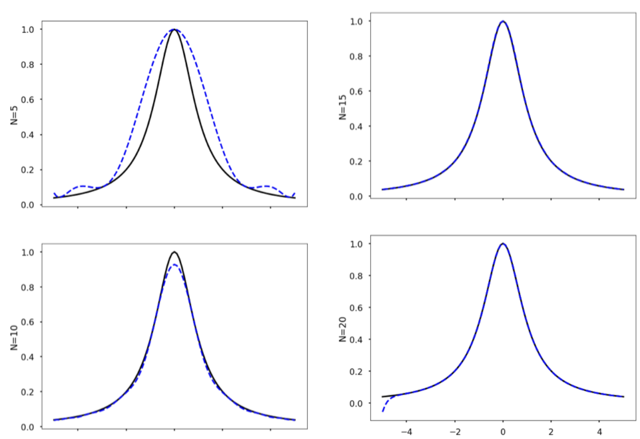
Here, the best performing method is the Natural Cubic Spline. It behaves better at the endpoints since it incorporates multiple polynomials.

2. (a)

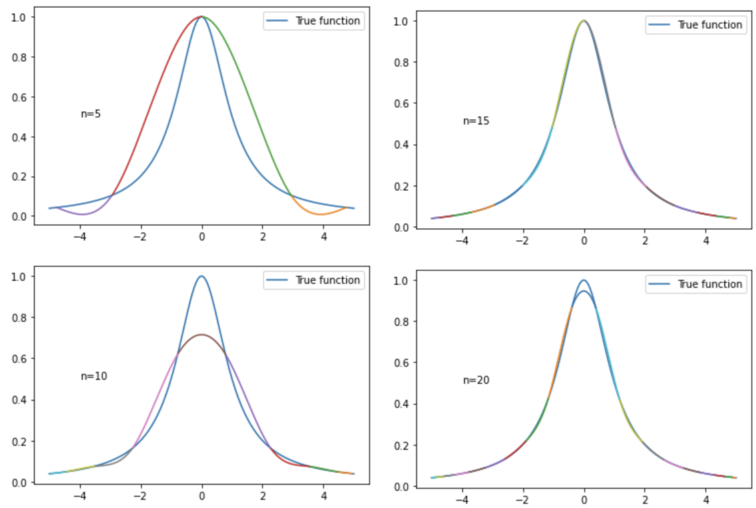


Polynomial approximation by Lagrange interpolation
True function
Interpolating Polynomial
sample points
Polynomial approximation by Lagrange interpolation
True function
Interpolating Polynomial
sample points
Polynomial approximation by Lagrange interpolation
True function
Interpolating Polynomial
sample points
Polynomial approximation by Lagrange interpolation
True function
Interpolating Polynomial
sample points

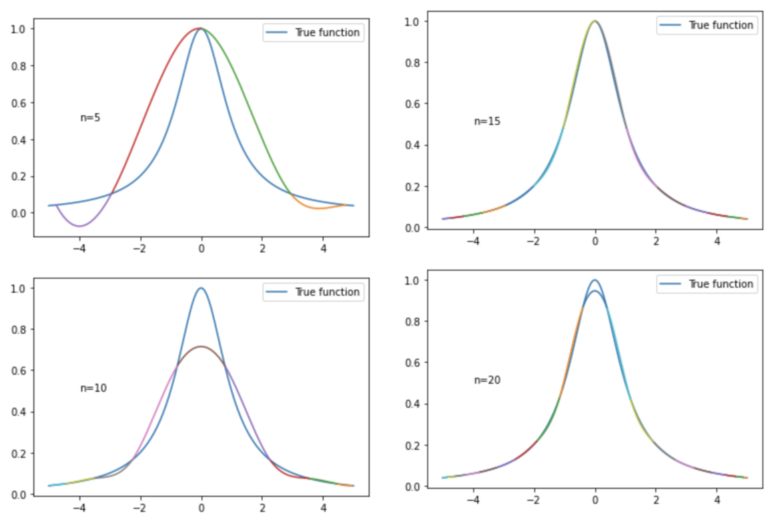

(b)



N=5
N=15
N=10
N=20




(c)




(d)

As seen above, both the Lagrange and Hermite interpolations behave better at the endpoints. However, the cubic splines method are not as affected.

3. In order for the spline to be naturally periodic, the start and end points need to be continuous. Therefore, the conditions are $S_1'(x_0) = S_n'(x_n)$ and $S_1''(x_0) = S_n''(x_n)$ under the assumption that $y_0 = y_n$.

4. $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$

$b = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$

$A^T Ax = A^T b$

$x = (A^T A)^{-1}(A^T b)$

$A^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$

$A^T A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$ $(A^T A)^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$

$A^T b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$x = (A^T A)^{-1}(A^T b) = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{2} \end{bmatrix}$

5. $A = \begin{bmatrix} 1 & 3 \\ 6 & -1 \\ 4 & 0 \\ 2 & 7 \end{bmatrix}$ $c = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ $b = \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \\ b_4^2 \end{bmatrix}$

Multiply by Diagonal Matrix to get regular least squares solution $(Ax-c = b)D => DAx - Dc = Db$

where $D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$

This gives the least squares equations $DAx = Dc$

$DA = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 6 & -1 \\ 4 & 0 \\ 2 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 12 & -2 \\ 20 & 0 \\ 6 & 21 \end{bmatrix} = A'$

$Dc = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 15 \\ 12 \end{bmatrix} = c'$

Solving the least squares equation $A'^T A'x = A^T c'$

$x = (A'^T A')^{-1} A'^T c'$

$A'^T = \begin{bmatrix} 1 & 12 & 20 & 6 \\ 3 & -2 & 0 & 21 \end{bmatrix}$ $A'^T A' = \begin{bmatrix} 581 & 105 \\ 105 & 454 \end{bmatrix}$

$(A'^T A')^{-1} = \begin{bmatrix} \frac{454}{252749} & -\frac{15}{36107} \\ -\frac{15}{36107} & \frac{83}{36107} \end{bmatrix}$

$A'^T c' = \begin{bmatrix} 1 & 12 & 20 & 6 \\ 3 & -2 & 0 & 21 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \\ 15 \\ 12 \end{bmatrix} = \begin{bmatrix} 421 \\ 247 \end{bmatrix}$

$$(A'^T A')^{-1} A'^T c' = \begin{pmatrix} \frac{165199}{252749} \\ \frac{14186}{36107} \end{pmatrix} \approx \begin{pmatrix} 0.6536 \\ 0.3928 \end{pmatrix}$$

# 1 Appendix

Question 1 and 2

1. 
```
import math
import numpy.linalg as la

def driver_lagrange_1(n):
    '''
    Demonstrating approximation via interpolation
    '''
    f = lambda x: 1/(1+x**2)

    #n = 4 # number of sample points to use

    # choose a basis - comment out the other


    #xs = np.linspace(-5,5,n)
    #ys = f(xs)
    x_i=lambda i: 5*np.cos(((2*i-1)*math.pi)/(2*n))
    xs=[]
    ys=[]
    for i in range(1,n+1):
        xs.append(x_i(i))
        ys.append(f(x_i(i)))

    basis = lagrange_basis(xs)

    coeffs = get_interpolation_coefficients(xs, ys, basis)
    polynomial_text = ' + '.join([f'{c:.2f}x^{i}' for i, c in enumerate(coeffs)])

    # test the function on a finer grid
    zs = np.linspace(xs[0], xs[-1], 201)
    zs_eval = interp_eval(zs, coeffs, basis)

    plt.figure(3, figsize=(16,6))

    plt.plot(zs, f(zs), label='True function')
    plt.plot(zs, zs_eval, label='Interpolating Polynomial')
    #plt.plot(xs, f(xs), 'k.', label='sample points')
    plt.plot(xs, ys, 'k.', label='sample points')
    c=str(n)
    plt.text(-1, .5, 'n='+c)

    plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
```

```
    plt.title('Polynomial approximation by Lagrange interpolation')
    plt.show()


def lagrange_polynomial(x, x_i, other_points):
    # f_i(x_j) = 0 for i =/= j
    # f_j(x_j) = 1
    # phi(x) = product of (x - x_j)/(x_i - x_j)
    y = 1 # start with 1 when making a product in a loop
    for x_j in other_points:
        y *= (x - x_j)/(x_i - x_j)
    return y

def remove_point_from_list(x_i, points):
    return [x_j for x_j in points if x_j != x_i]

def lagrange_basis(points):
    return [lambda x, x_i=x_i, other_points=remove_point_from_list(x_i, points):
                lagrange_polynomial(x, x_i, other_points)
            for x_i in points]

def get_interpolation_coefficients(xs, ys, basis):
    M = np.array([[phi(x) for phi in basis] for x in xs])
    return la.solve(M, ys)

def interp_eval(zs, coeffs, basis):
    return sum(c*phi(zs) for c, phi in zip(coeffs, basis))


for i in range(5,21,5):
    driver_lagrange_1(i)
```

2.

```
    from math import factorial

def driver_hermite_1():
    function_derivative_pairs = [
        [lambda x: 1/(1+x**2), lambda x: (-2*x)/(x**2+1)**2]]
    for f, df in function_derivative_pairs:
        zs = np.linspace(-5, 5, 101)
        exact = f(zs)
        Ns = [5,10,15,20]
        fig, axes = plt.subplots(len(Ns), 2, figsize=(20, 30), sharex=True)

        for i, N in enumerate(Ns):
            #xs = np.linspace(-5, 5, N)
            #ys = f(xs)
```

```
            x_i=lambda i: 5*np.cos(((2*i-1)*math.pi)/(2*N))
            xs=[]
            ys=[]
            ds=[]
            l = lambda x: 1/(1+x**2)
            for t in range(1,N+1):
                xs.append(x_i(t))
                ys.append(l(x_i(t)))
                ds.append(df(x_i(t)))

            #ds = df(xs)
            print(ds)
            hermite_eval = hermite_interp_eval(zs, xs, ys, ds)
            axes[i][0].plot(zs, exact, ’k-’, label=’Exact’)
            axes[i][0].plot(zs, hermite_eval, ’b--’, label=’Hermite’)
            axes[i][1].semilogy(zs, np.abs(exact - hermite_eval), ’b--’, label=’Hermite
            axes[i][0].set_ylabel(f’N={N}’)

            #axes[i][1].set_ylim(1e-16, 1e0)
            #axes[0][0].set_title(f’Approximation of 1/(1+x^2)’)
            #axes[0][1].set_title(’Error’)
            #axes[0][0].legend()

def hermite_interp_eval(zs, xs, ys, ds):
    ’’’
    Interpolate the values ys and the derivative values ds at
    the domain values xs then return the evaluation of the
    interpolant at the values zs.
    ’’’
    assert len(xs) == len(ys)
    assert len(xs) == len(ds)
    repeated_values = np.repeat(xs, 2)
    inverleaved_values = np.array(list(zip(ys, ds))).flatten()
    # get monomial coefficients from Newton divided difference table
    coeffs = newton_interp(repeated_values, inverleaved_values)
    return newton_eval(zs, coeffs)

def newton_interp(xs, fs):
    ’’’
    Interpolate the values in the array fs at the points xs. If xs are
    repeated, the repeated values must be adjacent. It is easiest to provide
    them in increasing order. If xs are repeated, then the provided f values
    are to be interpreted as derivatives.
    ’’’
    assert len(xs) == len(fs)
    n = len(xs)
```

```
    # check derivative value
    order = [0]
    for i in range(1,n):
        if xs[i-1] == xs[i]:
            order += [order[i-1]+1]
        else:
            order += [0]
    ds = [fs[0]]
    for i in range(1,n):
        ds += [fs[i-order[i]]]
    # divided difference table
    for col in range(1, n):
        for row in range(n-1, col-1, -1):
            if order[row] >= col:
                ds[row] = fs[row-(order[row]-col)]/factorial(col)
            else:
                ds[row] = (ds[row] - ds[row-1])/(xs[row] - xs[row-col])

    # Horner's Rule
    for i in range(len(ds)-1-1,-1,-1):
        for j in range(i,len(ds)-1):
            ds[j] -= xs[i]*ds[j+1]

    return ds

def newton_eval(zs, coeffs):
    '''
    Evaluate a polynomial given coefficients from newton_interp
    '''
    ret = coeffs[-1]
    for c in reversed(coeffs[:-1]):
        ret = ret*zs + c
    return ret
driver_hermite_1()
```

3.
```
def natural_cubic_spline(x,y,n):
    h=[0]*(n+1)
    a=[0]*(n+1)
    for i in range(n):
        h[i]=(x[i+1]-x[i])
    for i in range(1,n):
        a[i]=(((3/h[i])*(y[i+1]-y[i]))-((3/h[i-1])*(y[i]-y[i-1])))
    l=[1]*(n+1)
    u=[0]*(n+1)
    z=[0]*(n+1)
```

```
    for i in range(1,n):
        l[i]=(2*(x[i+1]-x[i-1])-(h[i-1]*u[i-1]))
        u[i]=(h[i]/l[i])
        z[i]=((a[i]-(h[i-1]*z[i-1]))/l[i])
    l[n]=(1)
    z[n]=(0)
    c=[0]*(n+1)
    b=[0]*(n+1)
    d=[0]*(n+1)
    c[n]=0
    for j in range(n-1,-1,-1):
        c[j]=z[j]-(u[j]*c[j+1])
        b[j]=(y[j+1]-y[j])/h[j]-(h[j]*(c[j+1]+2*c[ j])/3)
        d[j]=(c[j+1]-c[j])/(3*h[j])
    return(y,b,c,d)

for q in range(5,21,5):
    j = lambda x: 1/(1+x**2)
    xs1 = np.linspace(-5,5,201)
    xs=np.linspace(-5,5,q)
    ys=j(xs)
    '''
    x_i=lambda i: 5*np.cos(((2*i-1)*math.pi)/(2*q))
    xs=[]
    ys=[]
    l = lambda x: 1/(1+x**2)
    for t in range(1,q+1):
        xs.append(x_i(t))
        ys.append(l(x_i(t)))
    '''
    plt.plot(xs1,j(xs1), label='True function')
    (a,b,c,d)=natural_cubic_spline(xs,ys,q-1)
    for i in range(q):
        #print(i)
        #print(xs[i:i+1])
        e=lambda x: a[i]+b[i]*(x-xs[i])+(c[i]*(x-xs[i])**2)+(d[i]*(x-xs[i])**3)
        if ((i+1)!=q):
            xs_new=np.linspace(xs[i], xs[i+1], 201)
            plt.plot(xs_new, e(xs_new))
    h=str(q)
    plt.text(-4, .5, 'n='+h)
    plt.legend()
    plt.show()
```

4. def clamped_cubic_spline(x,y,n, f1, f2):

```
    h=[0]*(n+1)
    a=[0]*(n+1)
    for i in range(n):
        h[i]=(x[i+1]-x[i])
    a[0]=(3*(y[1]-y[0]))/h[0]-3*f1
    a[n]=3*f2-(3*(y[n]-y[n-1]))/h[n-1]
    for i in range(1,n):
        a[i]=(((3/h[i])*(y[i+1]-y[i]))-((3/h[i-1])*(y[i]-y[i-1])))
    l=[2*h[0]]*(n+1)
    u=[.5]*(n+1)
    z=[a[0]/l[0]]*(n+1)
    for i in range(1,n):
        l[i]=(2*(x[i+1]-x[i-1])-(h[i-1]*u[i-1]))
        u[i]=(h[i]/l[i])
        z[i]=((a[i]-(h[i-1]*z[i-1]))/l[i])
    l[n]=h[n-1]*(2*-u[n-1])
    z[n]=(a[n]-(h[n-1]*z[n-1])/l[n])
    c=[0]*(n+1)
    c[n]=z[n]
    b=[0]*(n+1)
    d=[0]*(n+1)

    for j in range(n-1,-1,-1):
        c[j]=z[j]-(u[j]*c[j+1])
        b[j]=(y[j+1]-y[j])/h[j]-(h[j]*(c[j+1]+2*c[j])/3)
        #print((y[1]-y[0])/h[0]-(h[0]*(c[1]+2*c[0])/3))
        d[j]=(c[j+1]-c[j])/(3*h[j])
    #print(b[0])
    return(y,b,c,d)

for q in range(5,21,5):
    j = lambda x: 1/(1+x**2)
    j1= lambda x: (-2*x)/((x**2+1)**2)
    xs1 = np.linspace(-5,5,201)
    xs=np.linspace(-5,5,q)
    ys=j(xs)
    '''
    x_i=lambda i: 5*np.cos(((2*i-1)*math.pi)/(2*q))
    xs=[]
    ys=[]
    l = lambda x: 1/(1+x**2)
    for t in range(1,q+1):
        xs.append(x_i(t))
        ys.append(l(x_i(t)))
    '''
```

```
plt.plot(xs1,j(xs1), label='True function')
(a,b,c,d)=clamped_cubic_spline(xs,ys,q-1, j1(-5), j1(5))
for i in range(q):
    e=lambda x: a[i]+b[i]*(x-xs[i])+c[i]*(x-xs[i])**2+d[i]*(x-xs[i])**3
    if ((i+1)!=q):
        xs_new=np.linspace(xs[i], xs[i+1], 201)
        plt.plot(xs_new, e(xs_new))
h=str(q)
plt.text(-4, .5, 'n='+h)
plt.legend()
plt.show()
```