hw2_Ray

February 18, 2024

vertical: t horizantal: i

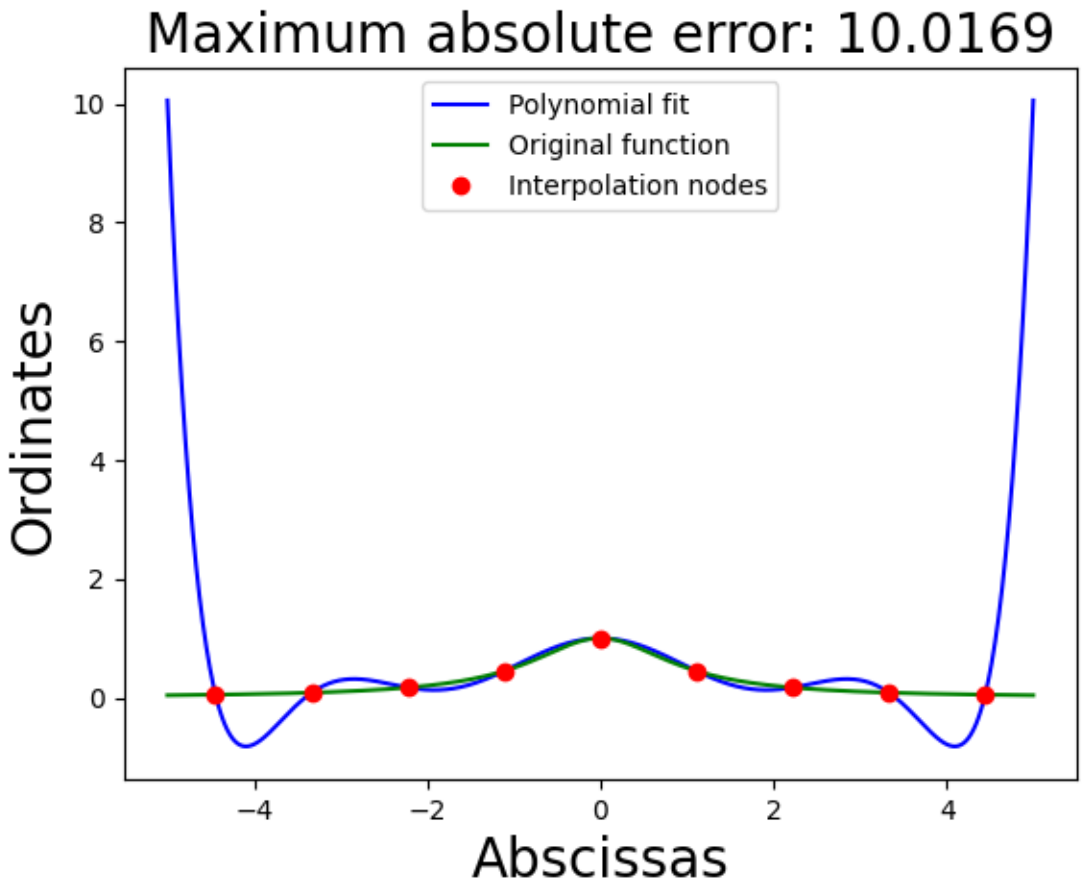
1 Exercise 1

[13]:
```
def PolyNewton(absc, ordi, x):
    absc = array(absc)
    ordi = array(ordi)
    if shape(absc) != shape(ordi):
        raise ValueError("Abscissas and ordinates have different length!")
    if len(shape(absc)) != 1:
        raise ValueError("Abscissas and ordinates must be 1D objects!")

    Npts = len(absc)
    coef = zeros((Npts, Npts))

    # first vertical line: f[x_1] ... f[x_n] = y_1 ...  y_n
    coef[:, 0] = ordi # y values
    # print(coef)
    for t in range(1, Npts): # update next level by combining two
        for i in range(Npts - t):
            coef[i][t] = (coef[i + 1][t - 1] - coef[i][t - 1]) / (absc[i + t] -␣
 ↪absc[i])

    product = ones((Npts,) + shape(x))
    for i in range(1, Npts):
        for t in range(i):
            product[i] *= (x - absc[t])

    return dot(coef[0], product)
```

[14]:
```
from pylab import *
ion()
from scipy.linalg import solve

ion()
function = lambda x: 1./(1+x**2)
interval = [-5, 5]
```

```
Npts = 9
L = interval[1]-interval[0]
x = linspace(interval[0], interval[1], 1001)


# constant interval abscissas
a = linspace(interval[0] + L/Npts/2, interval[1] - L/Npts/2 , Npts)
o = function(a)
p = PolyNewton(a, o, x)

figure(1)
plot(x, p,           'b-',
     x, function(x), 'g-',
     a, o,           'or')
xlabel("Abscissas", fontsize=20)
ylabel("Ordinates", fontsize=20)
legend(["Polynomial fit", "Original function", "Interpolation nodes"],
       loc="upper center")
title("Maximum absolute error: {:7.4f}".format(amax(fabs(p - function(x)))),␣
 ↪fontsize=20 )
```

[14]: Text(0.5, 1.0, 'Maximum absolute error: 10.0169')



2 E2

[15]:
```python
def get_Chebishev_zeros(n: int):
    import math
    result = []
    for i in range(1, n + 1):
        result.append(math.cos(((2 * i - 1) * math.pi) / (2 * n)))
    return result
```

To rescale the variable of a function defined on the interval $[-1, 1]$ so that it is defined on a new interval $[a, b]$, we can use a linear transformation. The idea is to map the old interval $[-1, 1]$ to the new interval $[a, b]$ by finding a function that takes any x in $[-1, 1]$ and gives a corresponding value in $[a, b]$.

The transformation can be achieved using the formula:

$$x' = \frac{b-a}{2}x + \frac{a+b}{2}$$

Here, x' is the new variable corresponding to x, and x is the original variable in the interval $[-1, 1]$.

This formula linearly transforms x from the interval $[-1, 1]$ to x' in the interval $[a, b]$.center of $[-1, 1]$) to the center of $[a, b]$, which is $\frac{a+b}{2}$.

This linear transformation ensures that $x = -1$ maps to $x' = a$ and $x = 1$ maps to $x' = b$, effectively rescaling and repositioning the function $f(x)$ from the interval $[-1, 1]$ to the interval $[a, b]$.

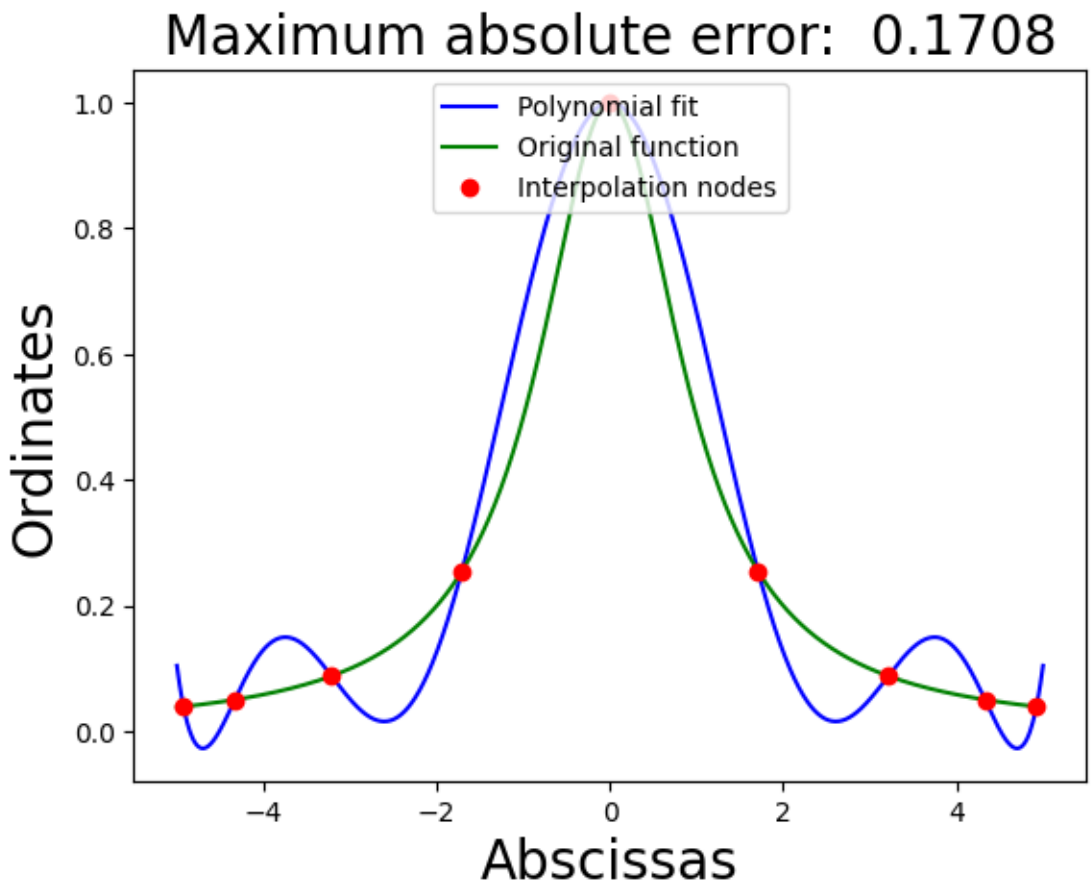
To calculate the result backward, we have

$$x = (\frac{2}{b-a}) \cdot (x' - \frac{a+b}{2})$$

[16]:
```
def get_Chebishev_zeros_scaled(a: int, b: int, n: int):
    raw_zeros = get_Chebishev_zeros(n = n)
    return [((b - a) / 2) * i + ((a + b) / 2) for i in raw_zeros]
```

[19]:
```
ion()
function = lambda x: 1./(1+x**2)
interval = [-5, 5]
Npts = 9
L = interval[1]-interval[0]
x = linspace(interval[0], interval[1], 1001)


# constant interval abscissas
a = get_Chebishev_zeros_scaled(a = interval[0], b = interval[1], n = Npts)
o = [function(i) for i in a]
p = PolyNewton(a, o, x)

figure(1)
plot(x, p,           'b-',
     x, function(x), 'g-',
     a, o,           'or')
xlabel("Abscissas", fontsize=20)
ylabel("Ordinates", fontsize=20)
legend(["Polynomial fit", "Original function", "Interpolation nodes"],
       loc="upper center")
title("Maximum absolute error: {:7.4f}".format(amax(fabs(p - function(x)))),␣
 ↪fontsize=20 )
```

[19]:
```
Text(0.5, 1.0, 'Maximum absolute error:  0.1708')
```



3 E3

[20]:
```
def SplineCoefficients(x_values, y_values):
    """Computes a matric containing the coefficients of the polynomials
        forming a set of natural cubic splines interpolating the points (absc,␣
 ↪ordi).

        Input:

        absc, ordi: 1D arrays or iterable python objects that convert to 1D
                    array. They must have the same length. absc must be ordered
                    in growing order.

        Output:

        A matrix with four columns and as many rows as the elements in absc
        minus one. If c0, c1, c2, c3 are the elements along the i-th row, the
        corresponding interpolating polynomial is
```

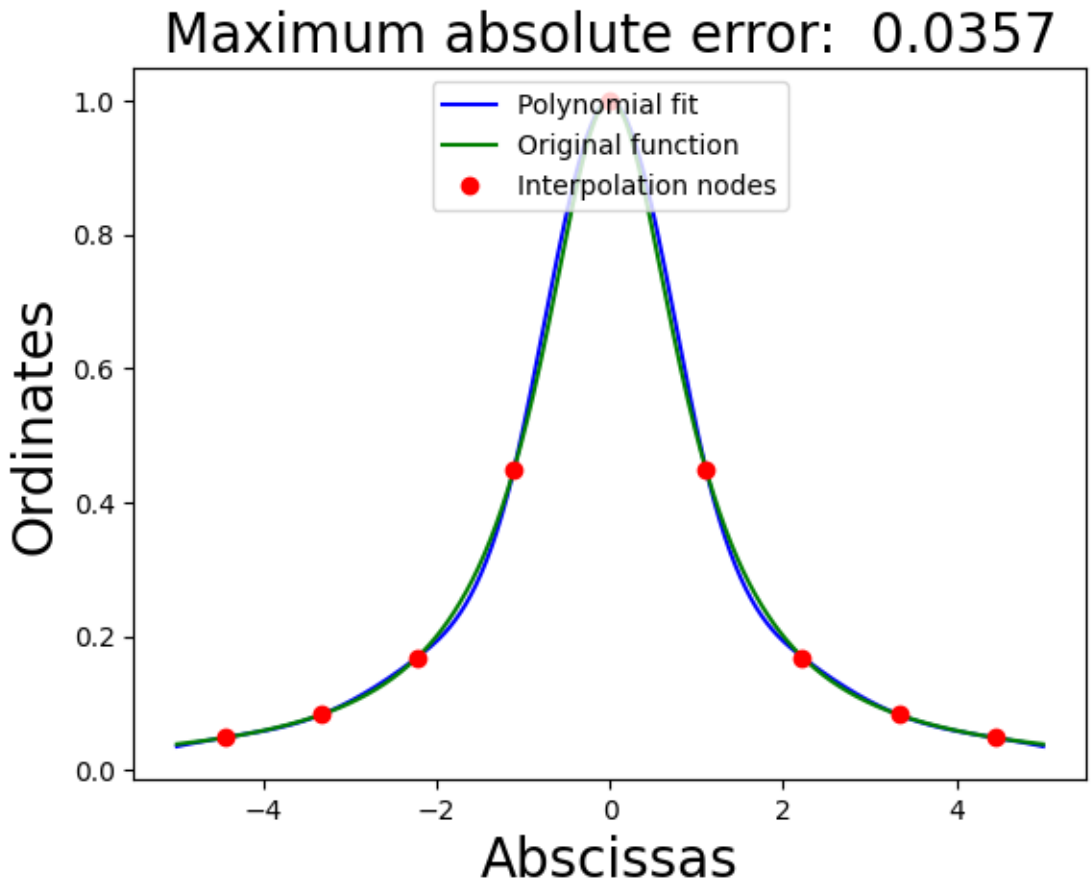
```
        S_i(x)=c0 + c1*(x - absc[i]) + c2*(x - absc[i])**2 + c3*(x - absc[i])**3

    """
    # Delete 'pass' and put here your code.
    # To solve eq. 3.24 in the book, use
    #    c = solve(A,K)
    # where A is the matrix containing the deltas and K is the vector
    # of the know solutions (you must prepare them both).
    # 'solve' has been imported at the beginning of this code.

    num_points = len(x_values)
    coefficients = np.zeros((num_points - 1, 4))

    # Calculate differences between consecutive x and y values
    delta_x = np.diff(x_values)
    delta_y = np.diff(y_values)

    # Initialize the matrix and vector for solving cubic spline coefficients
    matrix = np.zeros((num_points, num_points))
    vector = np.zeros(num_points)

    # Boundary conditions
    matrix[0, 0] = matrix[-1, -1] = 1

    # Set up the equations for the cubic spline (excluding boundaries)
    for i in range(1, num_points - 1):
        matrix[i, i - 1] = delta_x[i - 1]
        matrix[i, i] = 2 * (delta_x[i - 1] + delta_x[i])
        matrix[i, i + 1] = delta_x[i]
        vector[i] = 3 * (delta_y[i] / delta_x[i] - delta_y[i - 1] / delta_x[i -␣
↪1])

    # Solve for cubic spline coefficients
    c_coefficients = np.linalg.solve(matrix, vector)

    # Compute a, b, and d coefficients from c
    for i in range(num_points - 1):
        coefficients[i, 0] = y_values[i]  # a
        coefficients[i, 1] = (delta_y[i] / delta_x[i]) - (delta_x[i] / 3) * (2␣
↪* c_coefficients[i] + c_coefficients[i + 1])  # b
        coefficients[i, 2] = c_coefficients[i]  # c
        coefficients[i, 3] = (c_coefficients[i + 1] - c_coefficients[i]) / (3 *␣
↪delta_x[i])  # d

    return coefficients
```

```python
def __evaluate_spline(coeff, absc, x):
    """
    Evaluates the value of a cubic spline at a given point x.

    Parameters:
    - coeff: Coefficients of the cubic spline segments.
    - absc: Abscissas (x-values) at which the spline coefficients are defined.
    - x: The point at which to evaluate the spline.

    Returns:
    - The value of the spline at x.
    """
    # Find the right interval for x
    index = np.searchsorted(absc, x, side='right') - 1

    # Clamp the index to the valid range of [0, len(absc) - 2]
    index = max(min(index, len(absc) - 2), 0)

    # Extract the coefficients for the relevant spline segment
    a, b, c, d = coeff[index]

    # Calculate the relative position of x within the spline segment
    xi = absc[index]
    delta_x = x - xi

    # Evaluate the spline polynomial
    spline_value = a + b * delta_x + c * delta_x ** 2 + d * delta_x ** 3

    return spline_value


def evaluate_spline(coeff, absc, x):
    """Evaluates at x the the natural spline interpolant.

Input:

coeff:  matrix containinig the coefficients of the cubic interpolants.
        This is the output of 'SplineCoefficients'.

absc:   the same abscissas passed to 'SplineCoefficients' to compute 'coeff'.

x:      1D array or a number

Output: 1D array of same length as x, or a number if x is a number.

    """
```

```
    # x is a number
    if shape(x) == ():
        p = __evaluate_spline(coeff, absc, x)
    # x is a 1D array, or list, or tuple
    else:
        x = array(x)
        p = zeros_like(x)
        for i in range(len(x)):
            p[i] = __evaluate_spline(coeff, absc, x[i])
    return p
```

[22]:
```
ion()
function = lambda x: 1./(1+x**2)
interval = [-5, 5]
Npts = 9
L = interval[1]-interval[0]
x = linspace(interval[0], interval[1], 1001)


# constant interval abscissas
a = linspace(interval[0] + L/Npts/2, interval[1] - L/Npts/2 , Npts)
o = function(a)
coeff = SplineCoefficients(a,o)
p = evaluate_spline(coeff,a,x)

figure(1)
plot(x, p,           'b-',
     x, function(x), 'g-',
     a, o,           'or')
xlabel("Abscissas", fontsize=20)
ylabel("Ordinates", fontsize=20)
legend(["Polynomial fit", "Original function", "Interpolation nodes"],
       loc="upper center")
title("Maximum absolute error: {:7.4f}".format(amax(fabs(p - function(x)))),␣
 ↪fontsize=20 )
```

[22]:
```
Text(0.5, 1.0, 'Maximum absolute error:  0.0357')
```



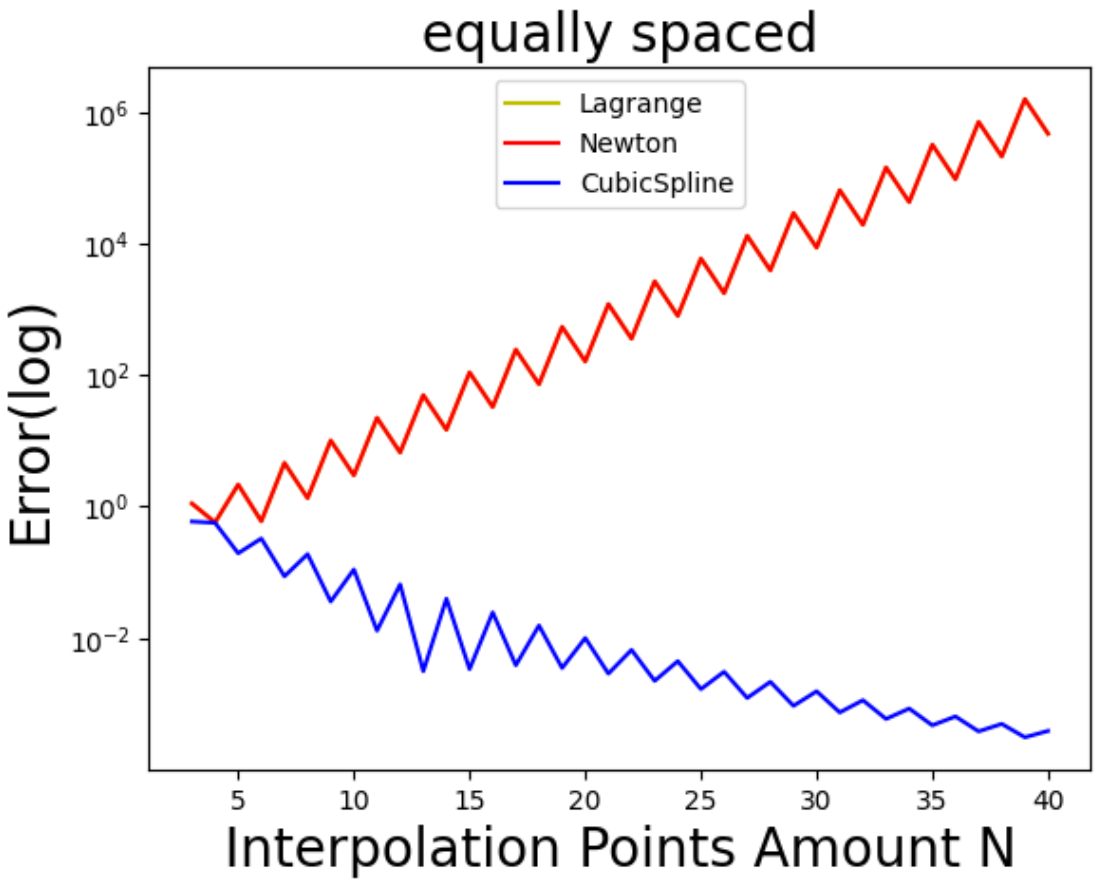
[24]:
```
def PolyLagrange(absc, ordi, x):
    """Evaluates at x the polynomial passing through the points having
abscissas 'absc' and ordinates 'ordi' using Lagrange's method.

Input:

absc, ordi: 1D arrays or iterable python objects that convert to 1D
            array. They must have the same length.

x: 1D array or a number

Output: 1D array of same length as x, or a number if x is a number.

    """
    absc = array(absc)
    ordi = array(ordi)
    if shape(absc) != shape(ordi):
        raise ValueError("Abscissas and ordinates have different length!")
```

```
    if len(shape(absc)) != 1:
        raise ValueError("Abscissas and ordinates must be 1D objects!")
    Npts = len(absc)
    product = ones((Npts,) + shape(x))
    for i in range(Npts):
        for j in range(Npts):
            if i == j: continue
            product[i] *= (x - absc[j]) / (absc[i] - absc[j])
    return dot(ordi, product)
```

4 E4

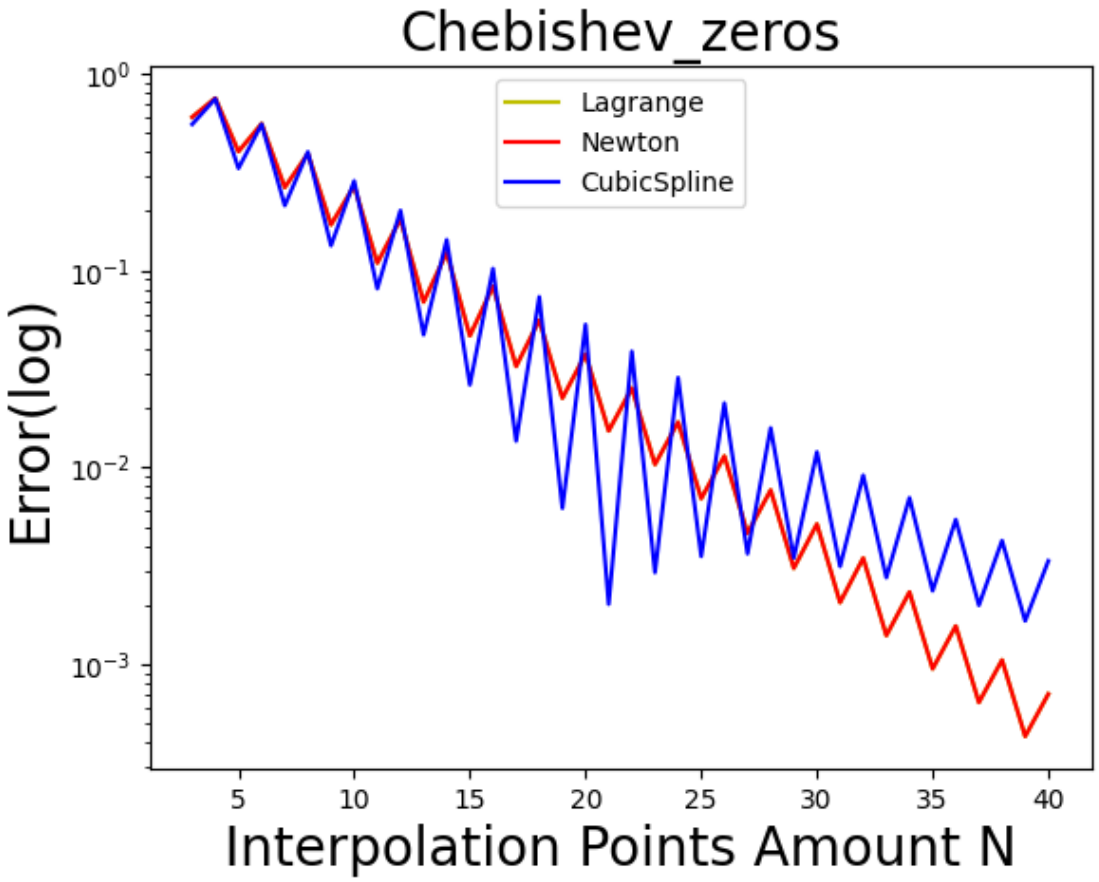
[36]:
```
ion()
function = lambda x: 1. / (1 + x ** 2)
interval = [-5, 5]
Npts = 20
L = interval[1] - interval[0]
x = linspace(interval[0], interval[1], 1001)

max_e_l = [] # Lagrange
m_e_n = [] # Newton
m_e_s = [] # Spline
N = range(3, 41)

for Npts in N:
    a = linspace(interval[0] + L/Npts/2, interval[1] - L/Npts/2 , Npts)
    # a = sort(get_Chebishev_zeros_scaled(interval[0], interval[1], Npts))
    o = function(a)

    p_Lagrange = PolyLagrange(a, o, x)

    p_Newton = PolyNewton(a, o, x)

    coeff = SplineCoefficients(a, o)
    p_Spline = evaluate_spline(coeff, a, x)

    max_e_l.append(amax(fabs(p_Lagrange - function(x))))
    m_e_n.append(amax(fabs(p_Newton - function(x))))
    m_e_s.append(amax(fabs(p_Spline - function(x))))

plot(N, max_e_l, 'y-',
     N, m_e_n, 'r-',
     N, m_e_s, 'b-')
xlabel("Interpolation Points Amount N", fontsize=20)
ylabel("Error(log)", fontsize=20)
legend(["Lagrange", "Newton", "CubicSpline"],
```

```
       loc="upper center")
title("equally spaced", fontsize=20 )
yscale('log')
```



[35]:
```
ion()
function = lambda x: 1. / (1 + x ** 2)
interval = [-5, 5]
Npts = 20
L = interval[1] - interval[0]
x = linspace(interval[0], interval[1], 1001)

max_e_l = [] # Lagrange
m_e_n = [] # Newton
m_e_s = [] # Spline
N = range(3, 41)

for Npts in N:
    # a = linspace(interval[0] + L/Npts/2, interval[1] - L/Npts/2 , Npts)
    a = sort(get_Chebishev_zeros_scaled(interval[0], interval[1], Npts))
```

```
    o = function(a)

    p_Lagrange = PolyLagrange(a, o, x)

    p_Newton = PolyNewton(a, o, x)

    coeff = SplineCoefficients(a, o)
    p_Spline = evaluate_spline(coeff, a, x)

    max_e_l.append(amax(fabs(p_Lagrange - function(x))))
    m_e_n.append(amax(fabs(p_Newton - function(x))))
    m_e_s.append(amax(fabs(p_Spline - function(x))))

plot(N, max_e_l, 'y-',
     N, m_e_n, 'r-',
     N, m_e_s, 'b-')
xlabel("Interpolation Points Amount N", fontsize=20)
ylabel("Error(log)", fontsize=20)
legend(["Lagrange", "Newton", "CubicSpline"],
       loc="upper center")
title("Chebishev_zeros", fontsize=20 )
yscale('log')
```



4.1 Conclusion

Looking at the graphs:

The graphs show the errors from different interpolation methods. For the Lagrange and Newton methods, the errors look the same because these methods give identical results when the same starting points (x values) are used, as seen in Exercise 1. In the first graph, where the interpolation points are spaced evenly, the error for the Lagrange and Newton methods goes up, but the error for the Cubic Spline method goes down. This happens because of something called Runge's phenomenon, which happens with high-degree polynomials at evenly spaced points and causes big errors. But the Cubic Spline method doesn't have this issue because it uses separate low-degree polynomials for different parts of the curve. It also keeps the curve smooth where the parts meet by matching up their second derivatives. This smooth joining stops the wild swings that high-degree polynomials can have.

In the second graph, using Chebyshev points, all methods show a steady decrease in error. This indicates that Chebyshev points avoid the issues seen with evenly spaced points. However, the error with the Cubic Spline method fluctuates more and ends up being larger than with the Newton and Lagrange methods.

In general, Chebyshev points are better for polynomial interpolation like Lagrange and Newton

because they reduce error more effectively. Cubic splines are not as affected by the spacing of points because they work in smaller sections and maintain certain smoothness conditions, which is why they do well with evenly spaced points too.