Numerical Analysis

Final Project

Jan Kretschmann

Comparing Interpolation Methods

- Description and Example Data Set
- Interpolating Known Functions
- Comparing Errors
- Conclusion

Methods of Interpolation

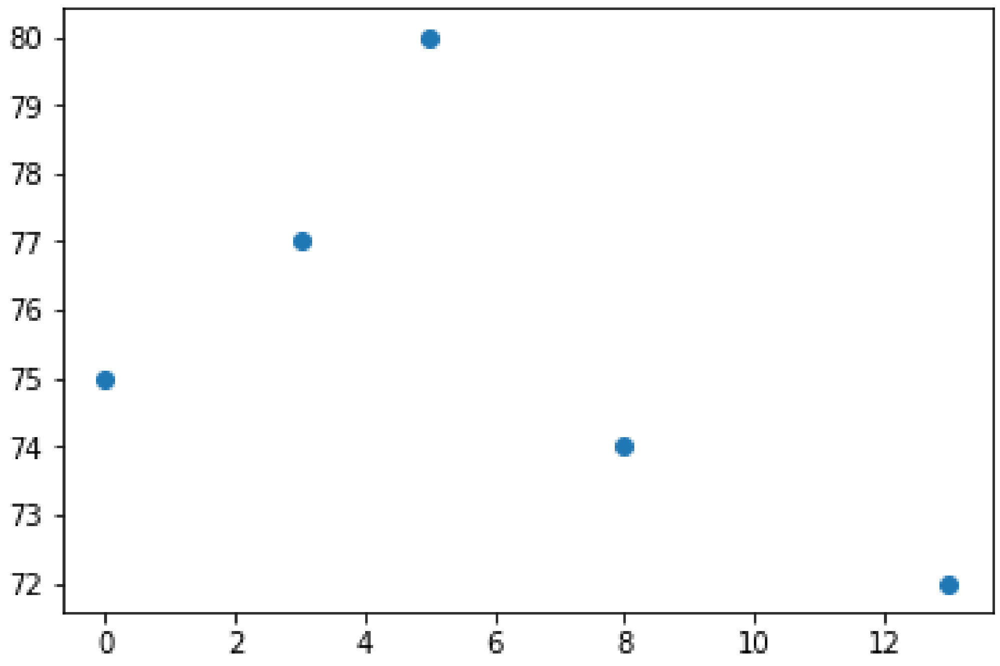
- Linear Interpolation
- Lagrange Interpolation
- Neville's Method
- Divided Differences
- Cubic Splines

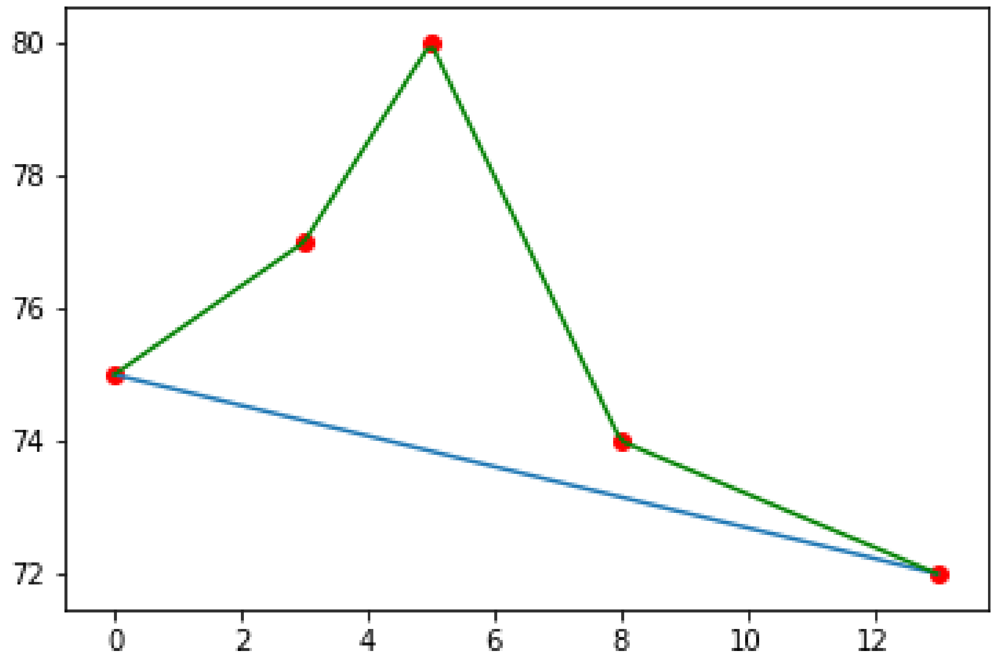
- Example Data Set

x	0	3	5	8	13
y	75	77	80	74	72

Linear Interpolation

- Straight Line through 2 points
  - $y = mx + b$
- Piecewise vs Non-Piecewise





Linear Interpolation

```
In [5]: def piecewise_linear_interpolation_value(data, x):
            # Find appropriate indices for the dataset
            index = np.where(data[:,0] >= x)[0][0]
            return linear_fct(x, data[index-1], data[index])

        def linear_interpolation_value(data, x):
            return linear_fct(x, data[0], data[-1])

        def linear_fct(x, p1, p2):
            m = (p2[1] - p1[1])/(p2[0] - p1[0])
            b = p1[1] - p1[0]*m
            return m*x + b
```

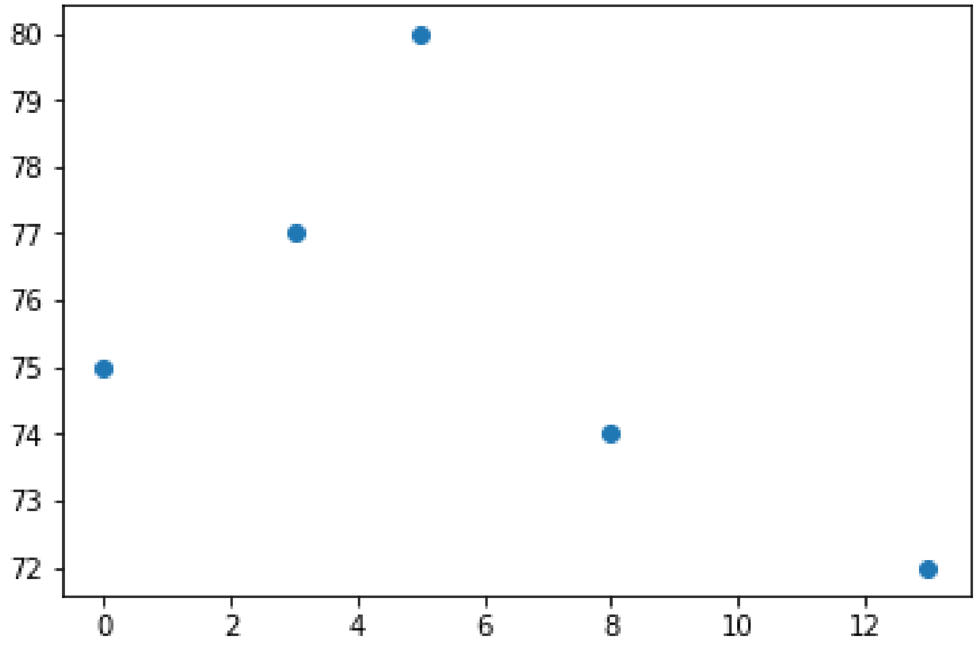
Lagrange Interpolation

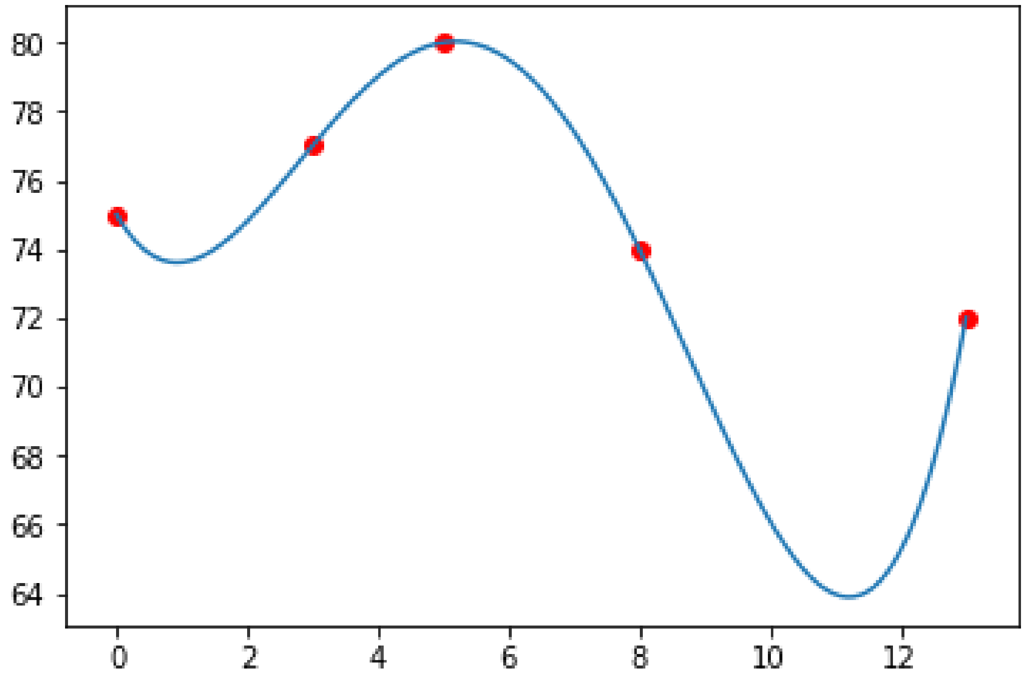
- Algebraic Polynomial fitting every Point
  - $L_{n,k}(x) = \frac{(x-x_0)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x_k-x_n)}$
- $L_{n,k}(x_i) = 0 \ \forall i \neq k$ and $L_{n,k}(x_k) = 1$
- $P_n(x) = \sum_{k=1}^{n} f(x_k) L_{n,k}(x)$
- Problem: numerically instable!

Lagrange Interpolation

In [3]:
```
def lagrange_polynomial_value(data, x):
    res = 0
    for i in range(len(data)):
        res += data[i, 1]*L(i, x, data)
    return res

def L(n, x, data):
    numerator = reduce(lambda a, b: a*b, [x - data[i,0] for i in range(len(data)) if i is not n])
    denominator = reduce(lambda a, b: a*b, [data[n, 0] - data[i, 0] for i in range(len(data)) if i is not n])
    return numerator/denominator
```

Lagrange Interpolation





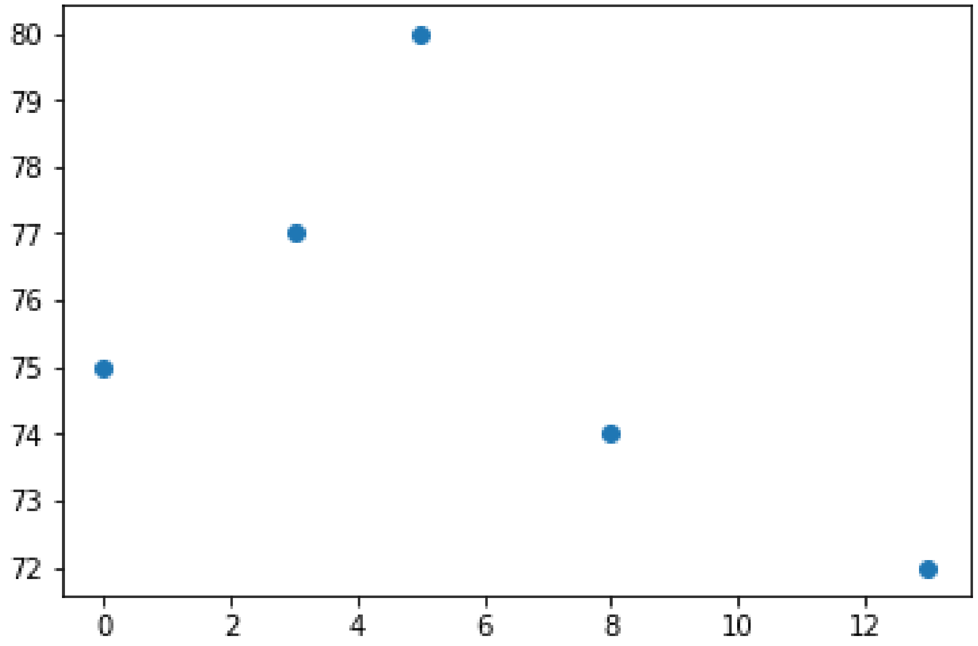
Divided Differences (Newton)

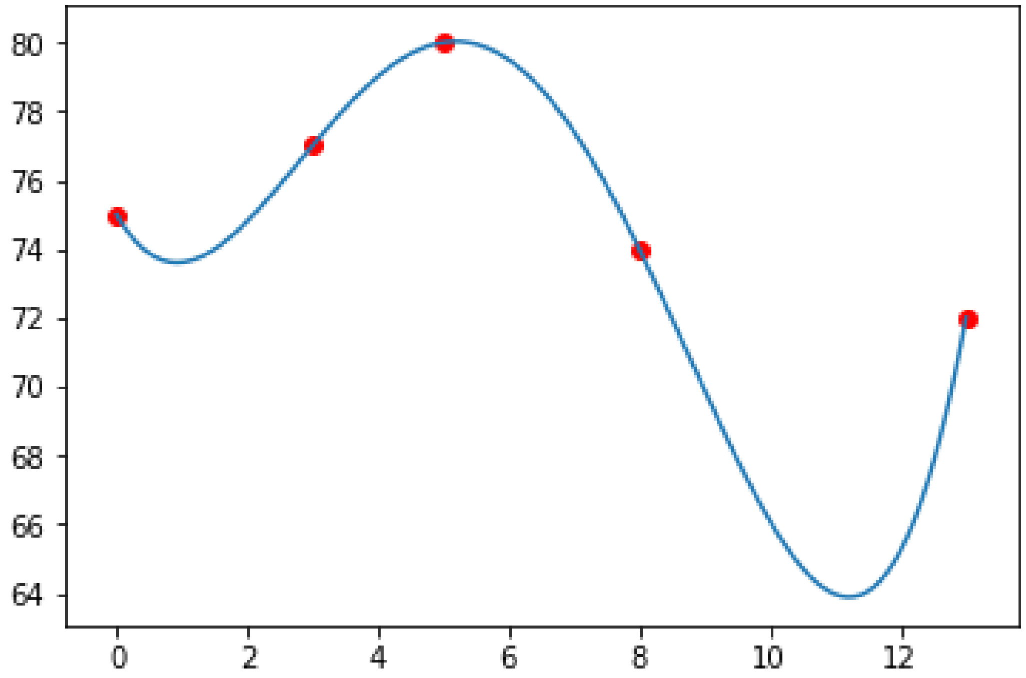
- Successively generated Polynomial
  - $P_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, \dots, x_k](x - x_0) \dots (x - x_k)$
  - With $f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$ and $f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$
- “Divided Differences“ as coefficients

Divided Differences

In [7]:
```
def newton_divided_differences(data, x):
    F = divided_differences(data)
    res = F[0]
    for i in range(1, len(F)):
        mult = reduce(lambda a, b: a*b, [x - data[j, 0] for j in range(i)])
        res += F[i]*mult
    return res

def divided_differences(data):
    coefficients = np.empty((len(data), len(data)))
    coefficients[:, 0] = data[:, 1]
    for i in range(1, len(data)):
        for j in range(1, i+1):
            coefficients[i, j] = (coefficients[i, j-1] - coefficients[i-1, j-1])/(data[i, 0] - data[i-j, 0])
    return np.diag(coefficients)
```

Divided Differences





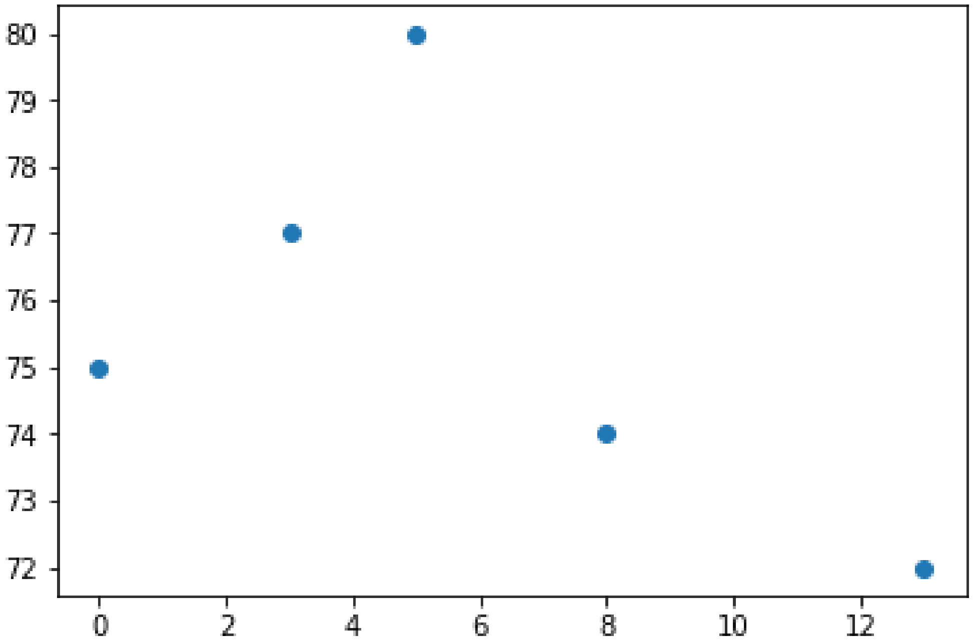
Neville's Method

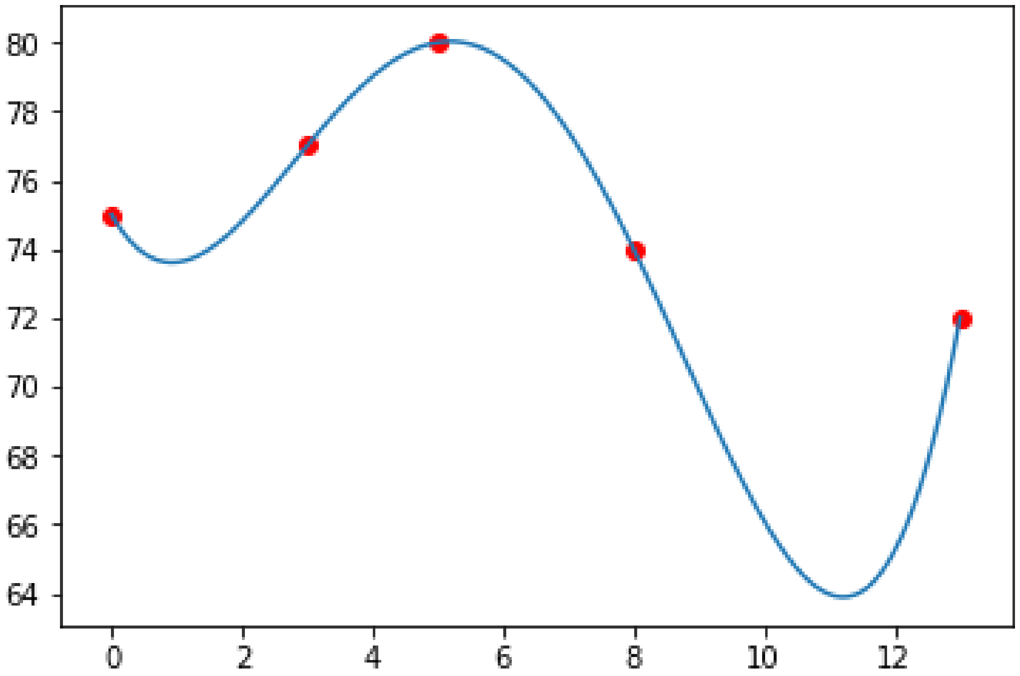
- Iterative Generation of Interpolation Polynomial
  - $P(x) = \frac{(x - x_j)P_{0,1,\ldots,j-1,j+1,\ldots k}(x) - (x - x_i)P_{0.1.\ldots.i-1,i+1,\ldots,k}(x)}{x_i - x_j}$
- Easiest to implement

Neville‘s Method

In [9]:
```
def neville_interpolation(data, x):
    Q = np.empty((len(data), len(data)))
    Q[:, 0] = data[:, 1]
    for i in range(1, len(data)):
        for j in range(1, i+1):
            Q[i, j] = ((x - data[i-j, 0])*Q[i, j-1] - (x - data[i, 0])*Q[i-1, j-1])/(data[i, 0] - data[i-j, 0])
    return Q[-1, -1]
```

Neville’s Method



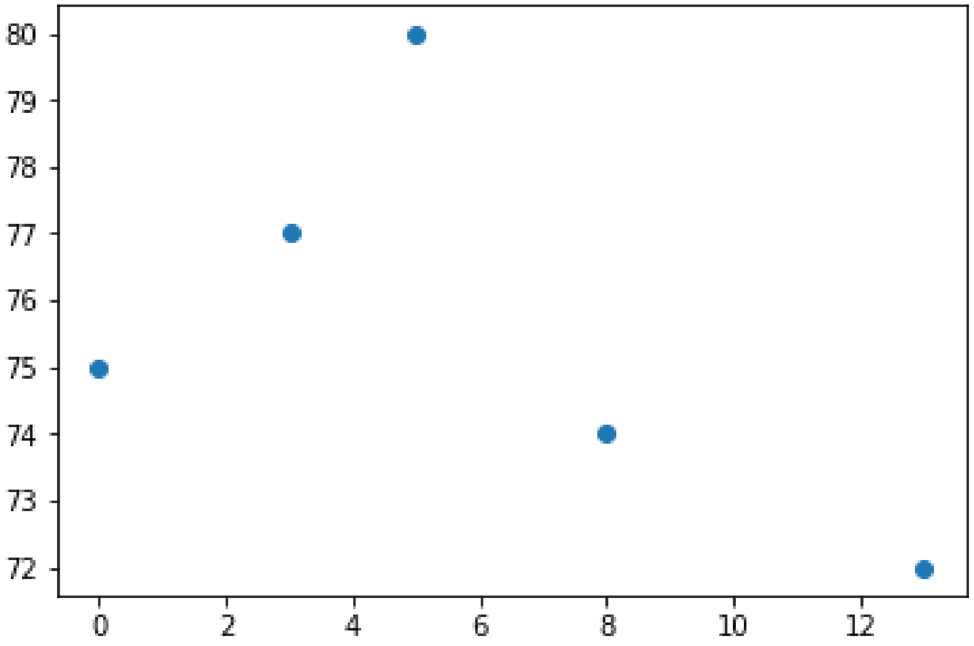


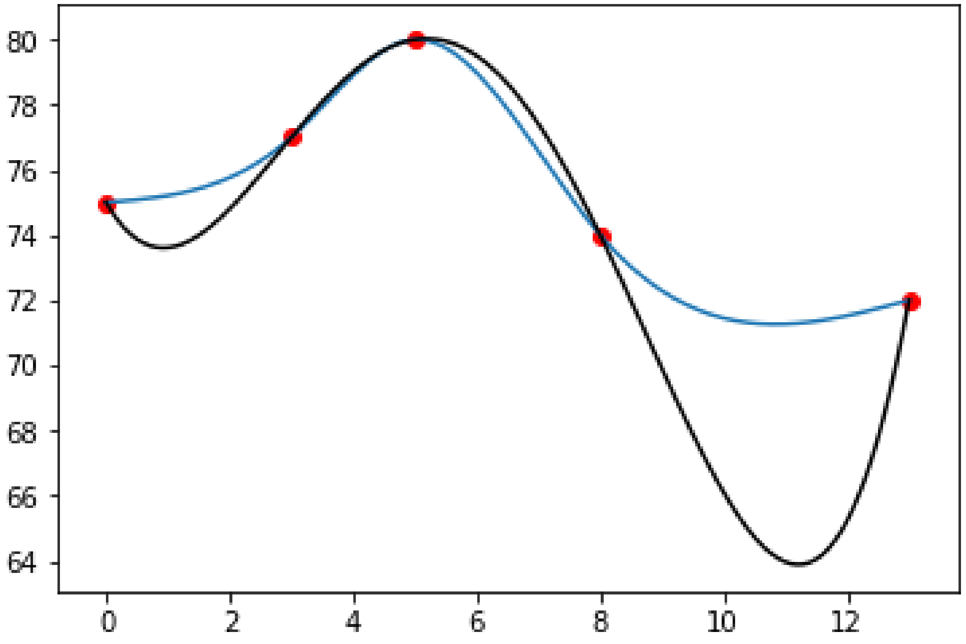
Cubic Splines

- Piecewise defined => non algbebraic
- Cubic Function for each interval $[x_i, x_{i+1}]$
- Numerically Stable
- Most complex implementation

```python
def find_index_spline(data, x):
    # Find the interval which x is in => the piece of the piecewise defined cubic to use
    index1 = np.where(data[:, 0] <= x)[0][-1]
    index2 = np.where(data[:, 0] >= x)[0][0]
    return int((index1+index2)/2)

def cubic_spline_value(data, x):
    a, b, c, d = cubic_spline(data)
    index = find_index_spline(data, x)
    return a[index] + b[index]*(x - data[index, 0]) + c[index]*(x - data[index, 0])**2 + d[index]*(x - data[index, 0])**3

def cubic_spline(data):
    a = np.empty(len(data))
    b = np.empty(len(data))
    c = np.empty(len(data))
    d = np.empty(len(data))
    a = data[:, 1]
    h = np.empty(len(data)-1)
    for i in range(len(h)):
        h[i] = data[i+1, 0] - data[i, 0]
    alpha = np.empty(len(data)-1)
    for i in range(1, len(data)-1):
        alpha[i] = 3/h[i] * (a[i+1] - a[i]) - 3/h[i-1] * (a[i] - a[i-1])

    l = np.empty(len(data)+1)
    mu = np.empty(len(data)+1)
    z = np.empty(len(data)+1)
    l[0] = 1
    mu[0] = 0
    z[0] = 0

    for i in range(1, len(data)-1):
        l[i] = 2*(data[i+1, 0] - data[i-1, 0]) - h[i-1]*mu[i-1]
        mu[i] = h[i]/l[i]
        z[i] = (alpha[i] - h[i-1]*z[i-1])/l[i]

    l[-1] = 1
    z[-1] = 0
    c[-1] = 0

    for j in range(len(data)-2, -1, -1):
        c[j] = z[j] - mu[j]*c[j+1]
        b[j] = (a[j+1] - a[j])/h[j] - h[j]*(c[j+1] + 2*c[j])/3
        d[j] = (c[j+1] - c[j])/(3*h[j])

    return a, b, c, d
```

Cubic Splines



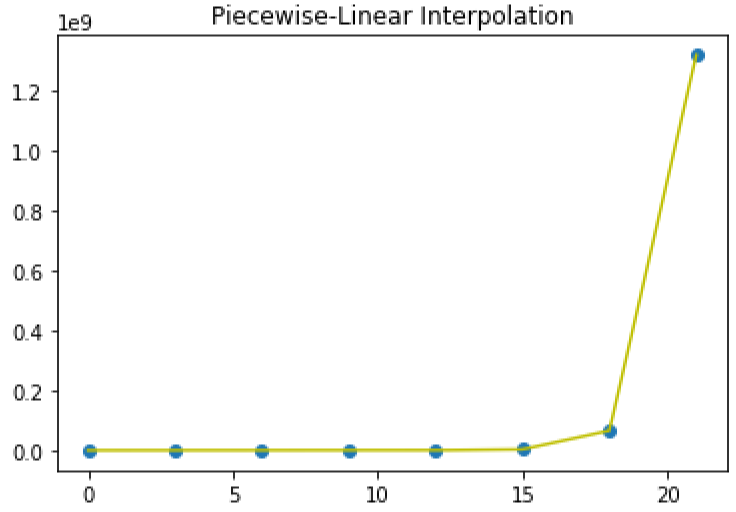


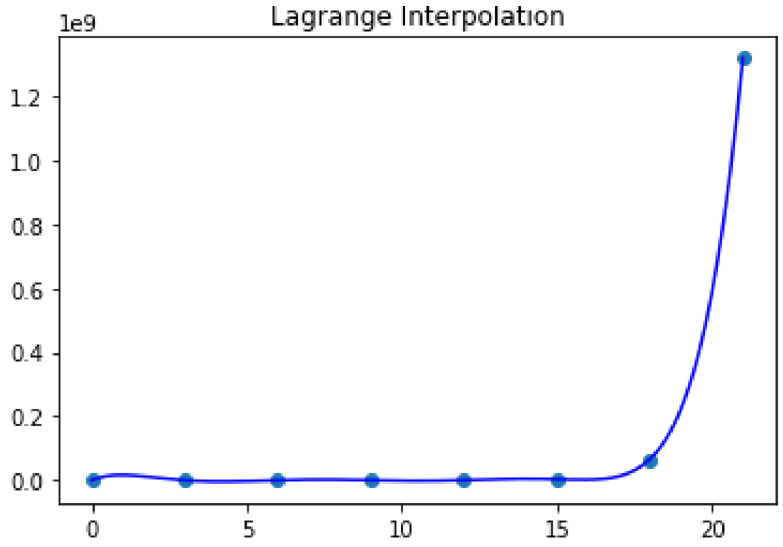
Interpolating Known Functions

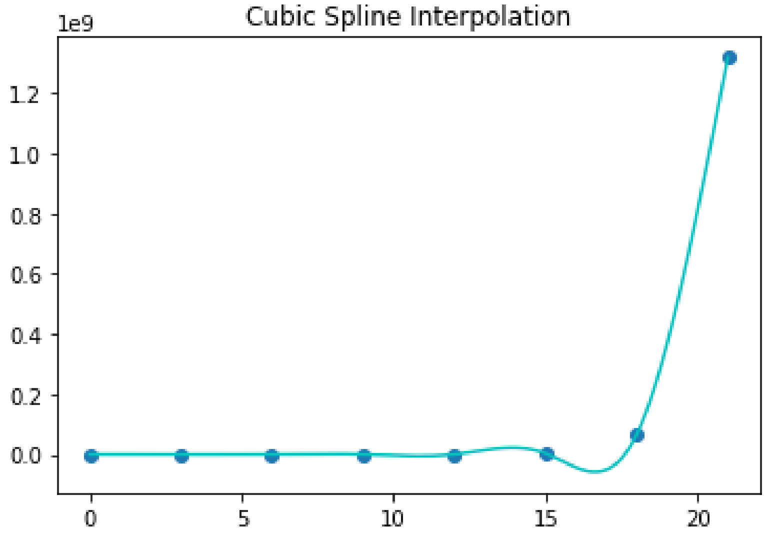
- Examining Interpolation Methods on
  - $f(x) = e^x$
  - $f(x) = \sin(x)$
  - $f(x) = x$
  - $f(x) = x^4$

Interpolating Known Functions

- $f(x) = e^x$

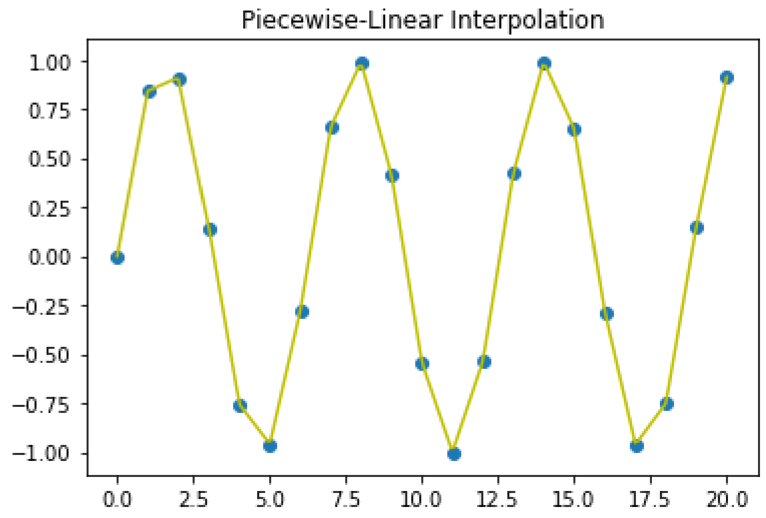


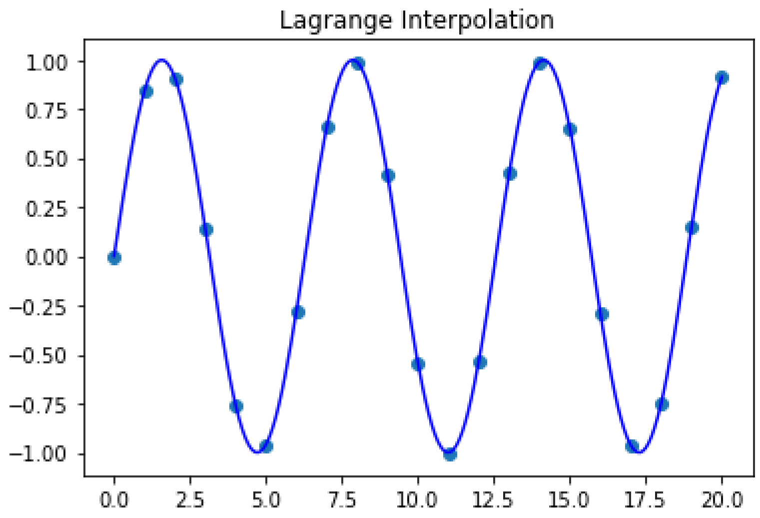


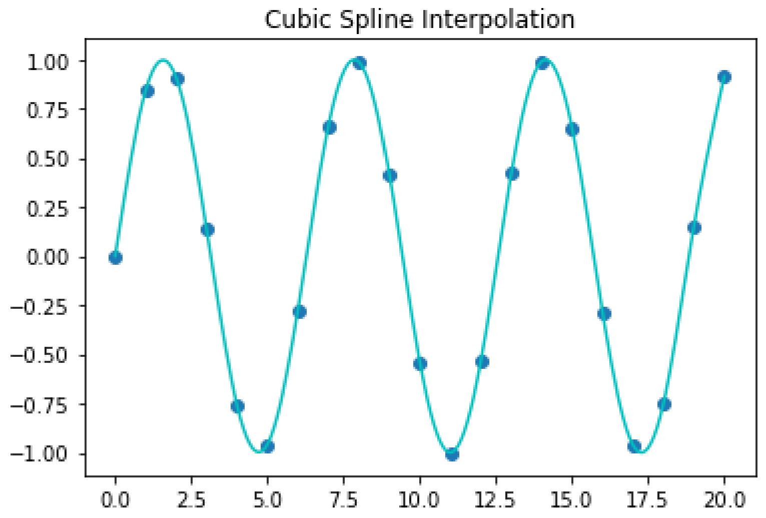


Interpolating Known Functions

- $f(x) = \sin(x)$

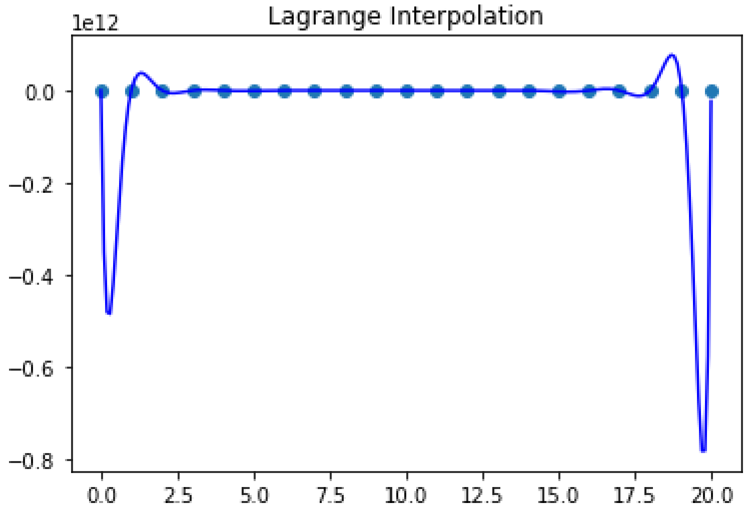


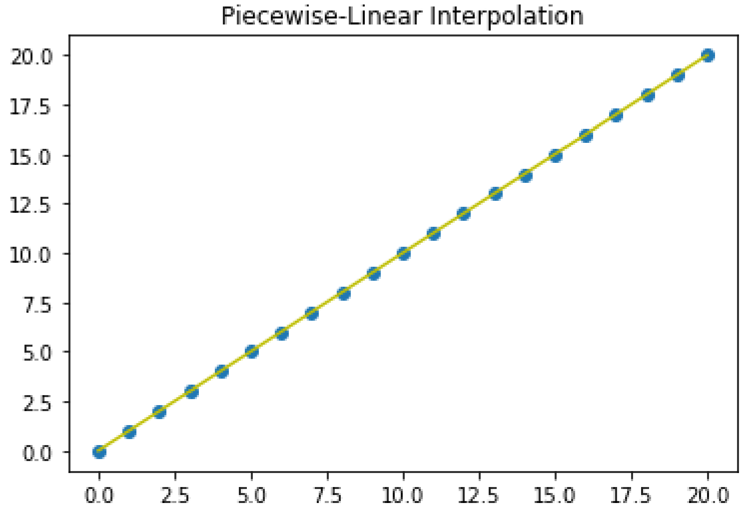


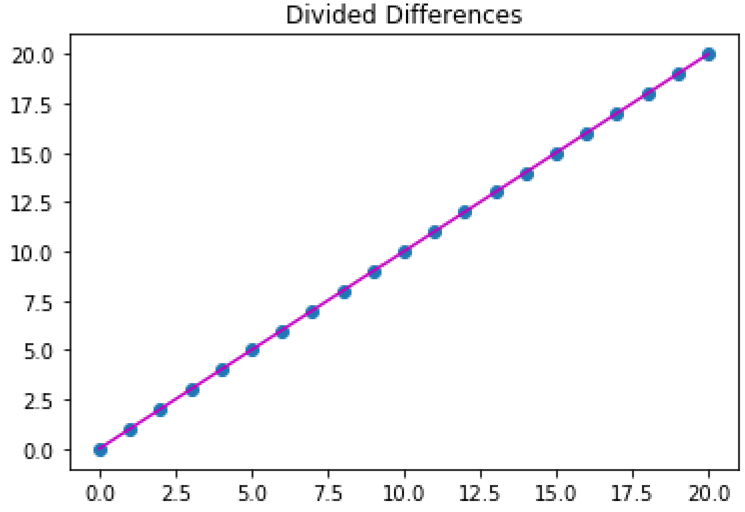


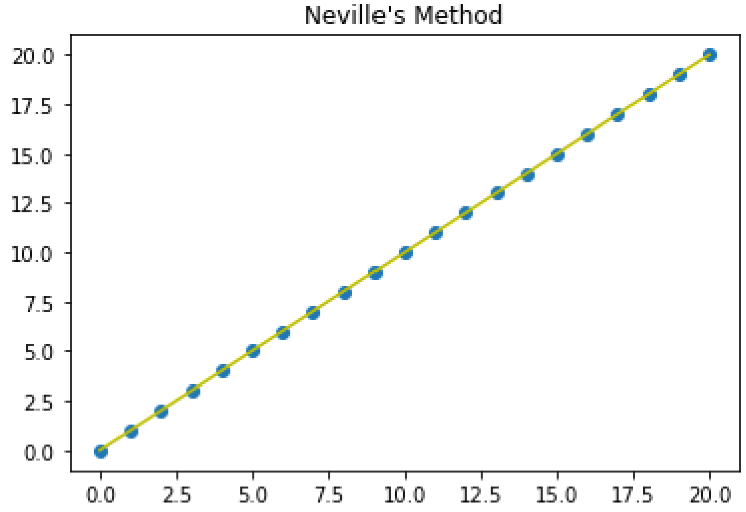
Interpolating Known Functions

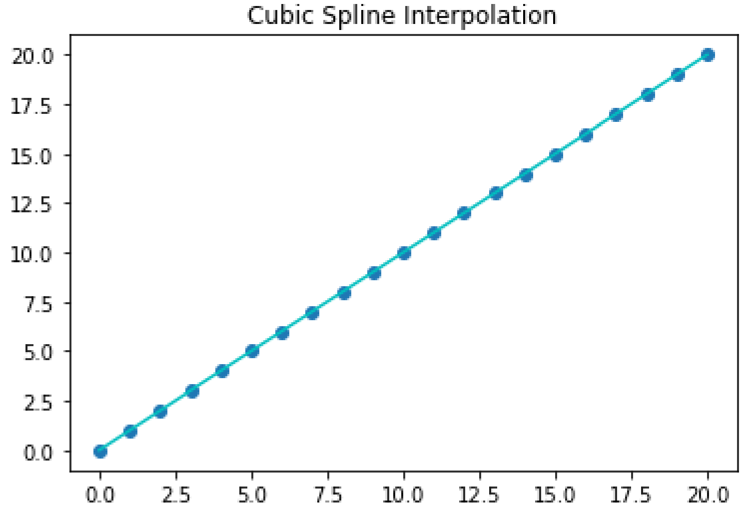
- $f(x) = \mathrm{x}$





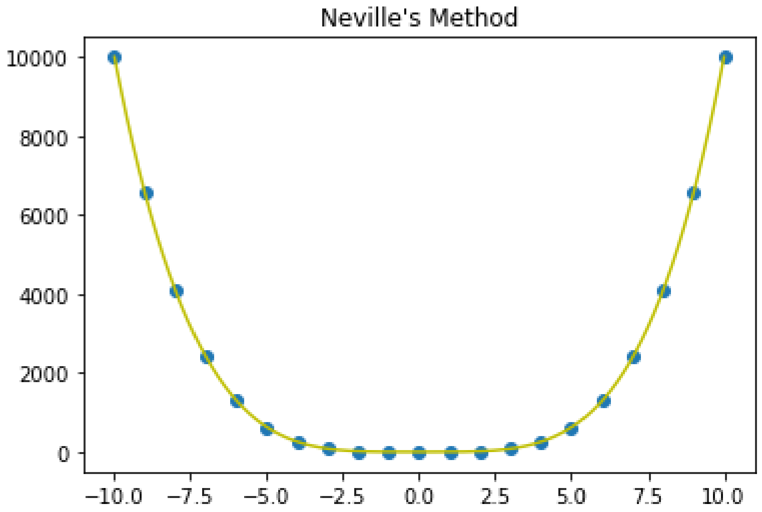


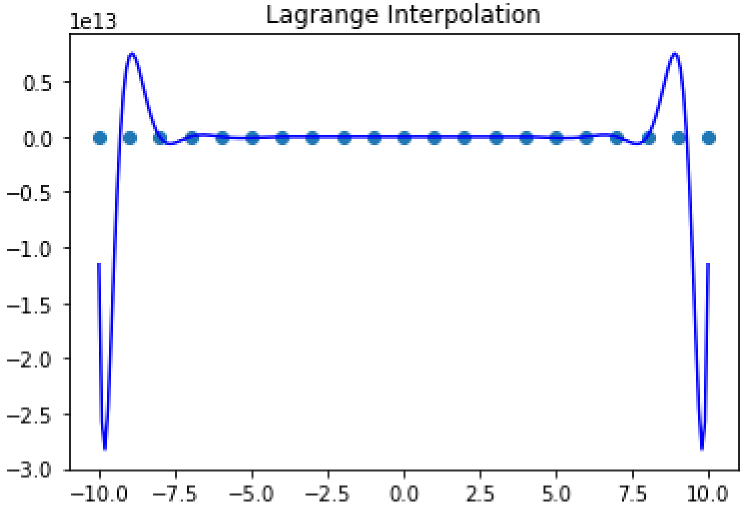


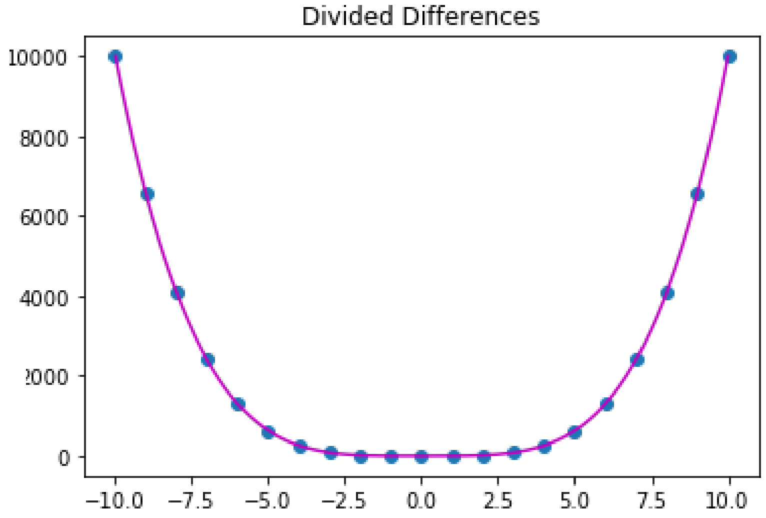


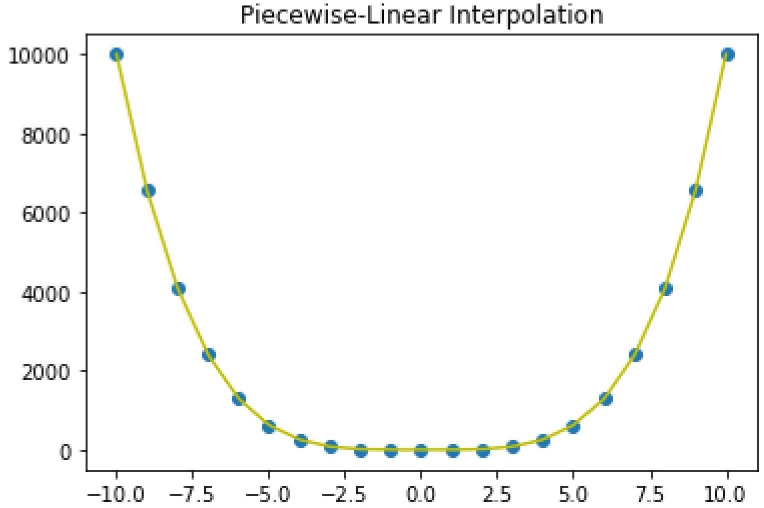
Interpolating Known Functions

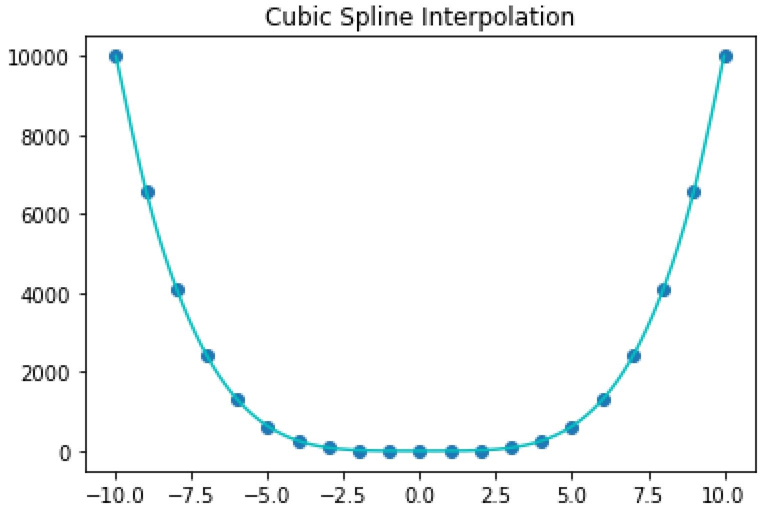
- $f(x) = x^4$



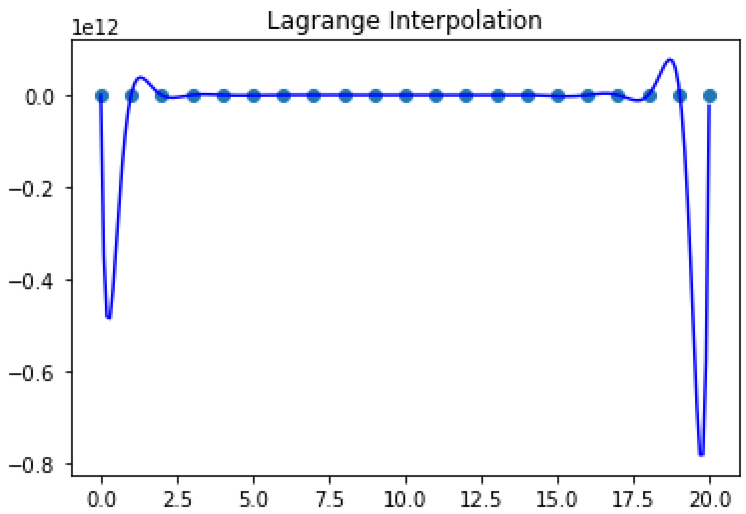




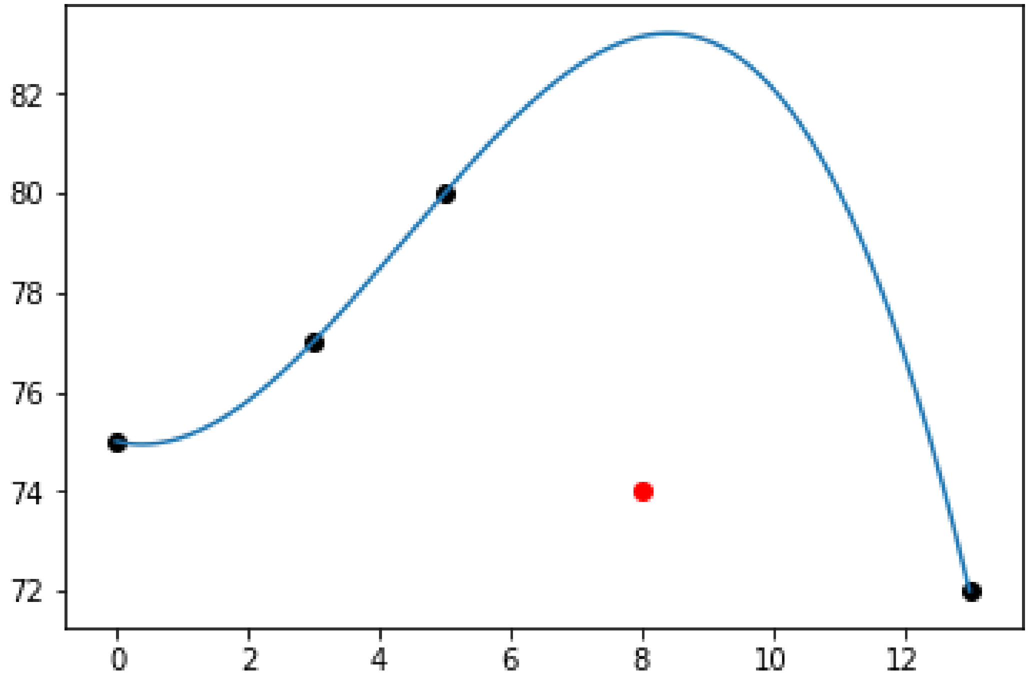




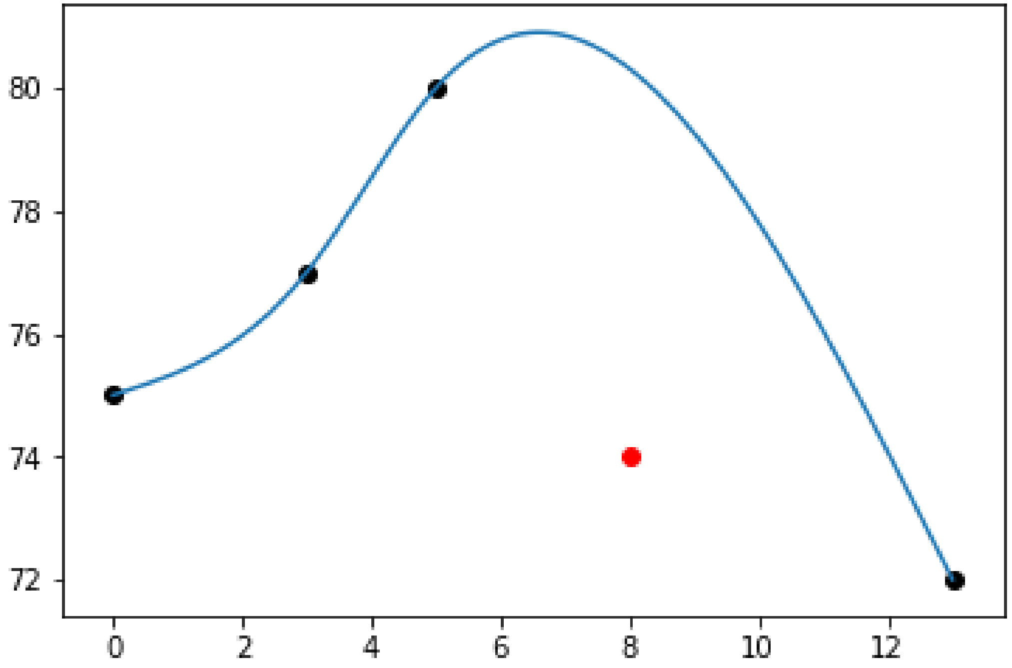
Comparing Errors



- Large roundoff error => Instability
- For unknown f: Cross Validation



Lagrange Interpolation without $(8, 74)$



Cubic Spline Interpolation without $(8, 74)$

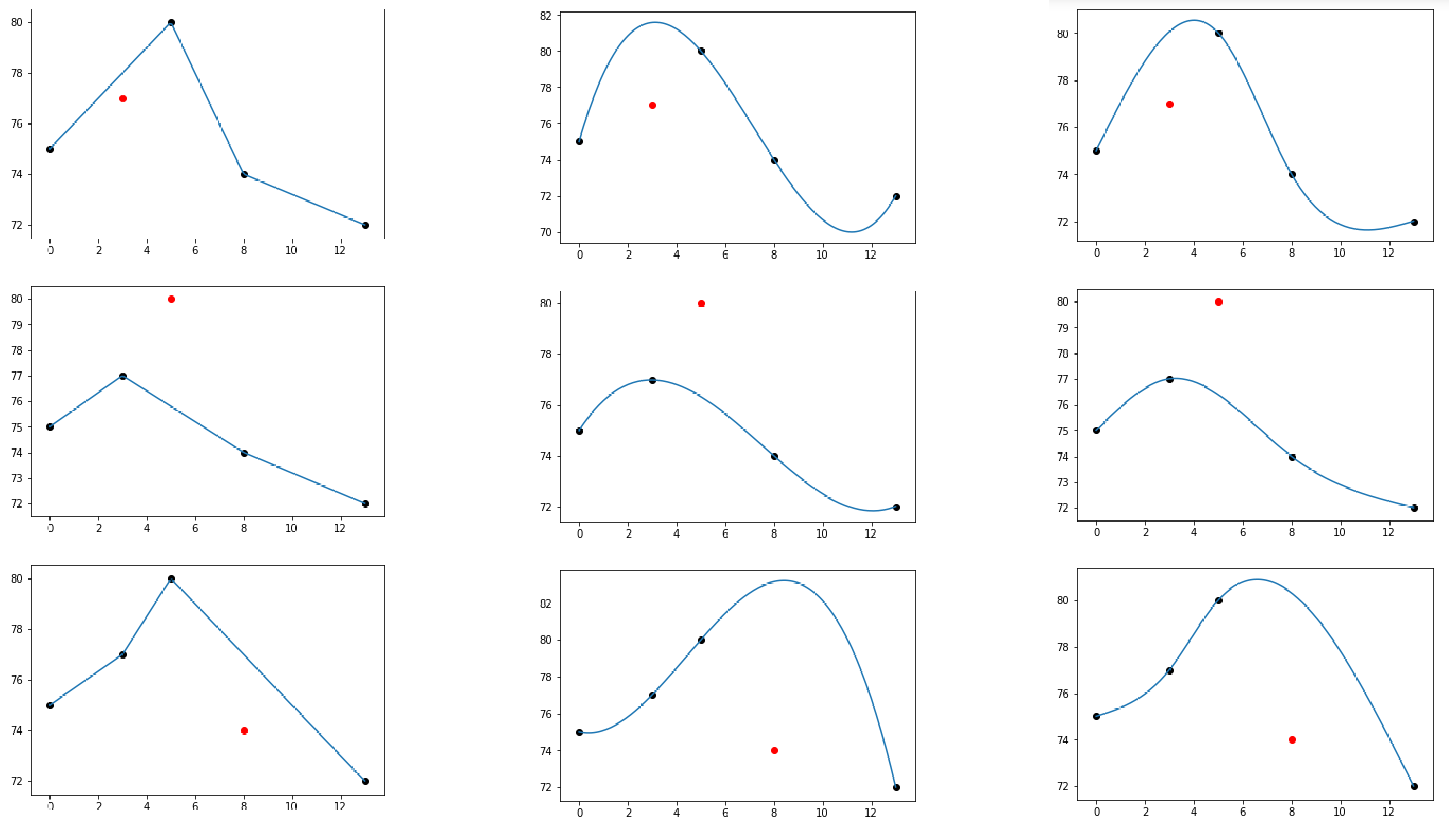
Comparing Errors

Method	Total Error
Linear (PW)	8.2
Lagrange	17.39
Newton	17.39
Neville	17.39
Spline	12.98

- Cross Validation Results

- Piecewise Linear Interpolation works best in this example

- But: not always!


80 78 76 74 72
0 2 4 6 8 10 12
82 80 78 76 74 72 70
0 2 4 6 8 10 12
80 78 76 74 72
0 2 4 6 8 10 12
80 79 78 77 76 75 74 73 72
0 2 4 6 8 10 12
80 78 76 74 72
0 2 4 6 8 10 12
80 79 78 77 76 75 74 73 72
0 2 4 6 8 10 12
80 78 76 74 72
0 2 4 6 8 10 12
82 80 78 76 74 72
0 2 4 6 8 10 12
80 78 76 74 72
0 2 4 6 8 10 12

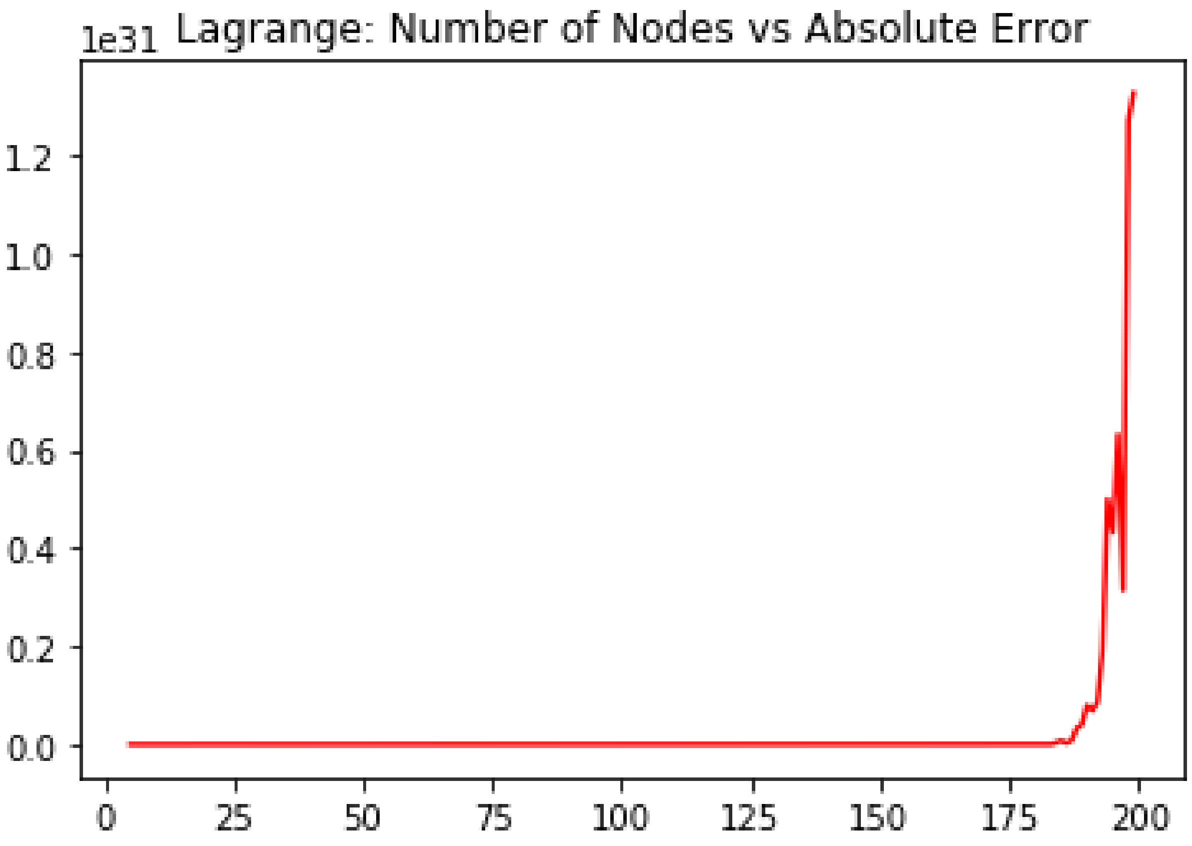
Errors for Known Functions

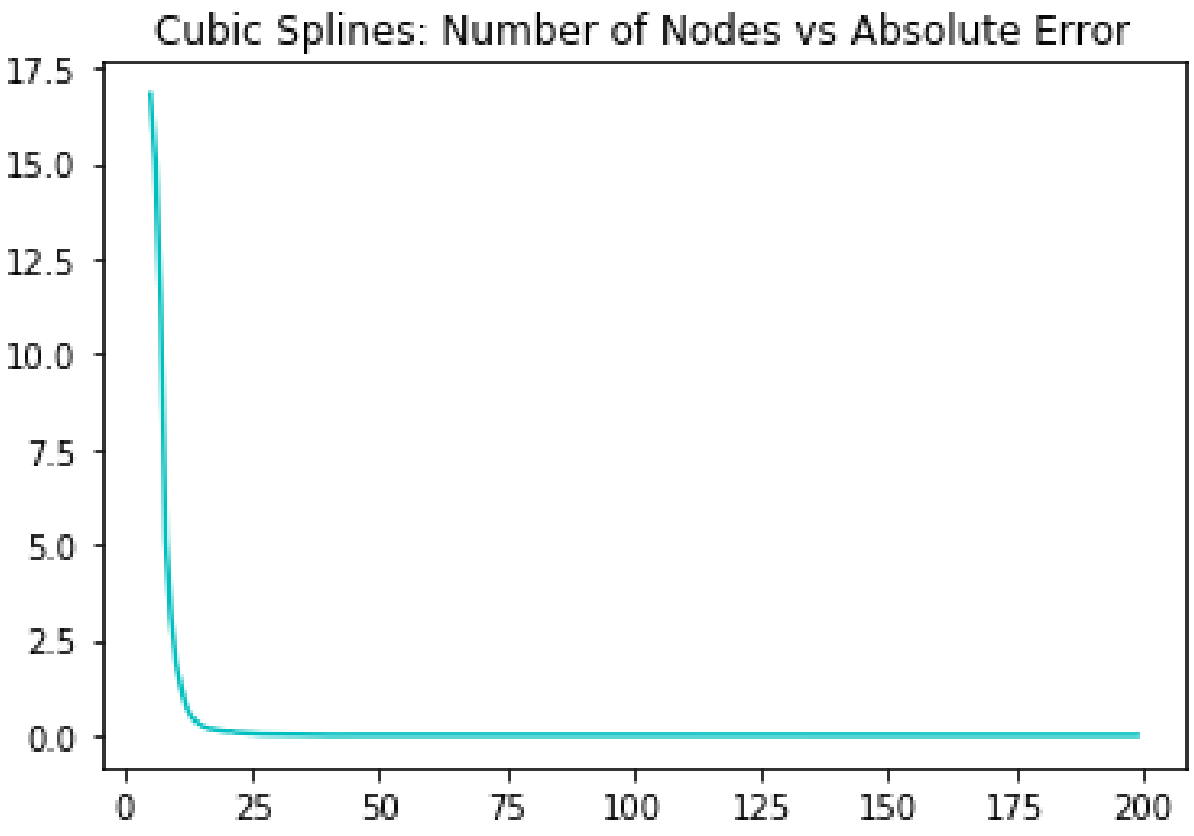
- Comparing interpolated value to actual value
- Example: $f(x) = \sin(x)$
  - Lagrange/Newton/Neville has lowest error

Method	Error
Linear (PW)	1.5349
Lagrange	0.0021
Newton	0.0021
Neville	0.0021
Spline	0.0969

Stability

- Lagrange instable → so what? Increase number of nodes to see!





Conclusion

- No Method is always the best
- Lagrange unstable for large n