LagrangeInterpolation

September 4, 2023

1 Lagrange interpolation

Let $(x_i, y_i), i = 0, 1, \cdots, n$ be a list of distinct points on the curve $y = f(x)$. We can find a polynomial $P_n(x)$ passing through all $n + 1$ points.

$$P_n(x) = \sum_{i=0}^{n} y_i \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

To see that this polynomial does indeed pass through all the points, consider the term

$$Q_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

evaluated when $x = x_k$ (where x_k is one of the x-coordinates in the list). With the exception of $Q_k(x_k)$, all $Q_i(x_k) = 0$ since there will a factor of $(x_k - x_k)$ multiplying all the other terms. On the other hand

$$Q_k(x_k) = \prod_{j=0, j\neq k}^{n} \frac{x_k - x_j}{x_k - x_j} = 1$$

The function lagrange_interp below implements this Lagrange polynomial (although not efficiently). Note that the "numba" module and the "@jit" decorator are not needed. They just speed up the calculation.

[38]:
```
import numpy as np
import matplotlib.pyplot as plt
import gmpy2
from numba import jit
WORKING_PRECISION = 256
gmpy2.get_context().precision = WORKING_PRECISION
```

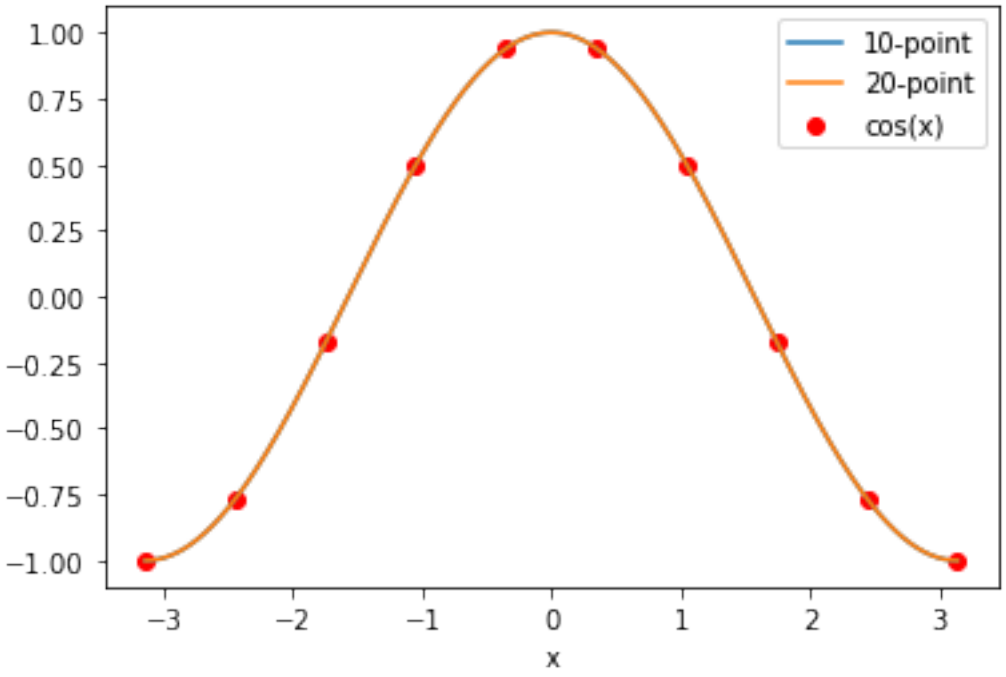
[39]:
```
@jit(nopython=True)
def lagrange_interp(x_val, x_data, y_data):
    assert len(x_data) == len(y_data)

    allresult = []
    for k in range(len(x_val)):
        x = x_val[k]
```

```
        result = 0;
        for i in range(len(y_data)):
            sub_result = y_data[i]
            for j in range(len(x_data)):
                if i == j:
                    continue
                sub_result *= (x - x_data[j])/(x_data[i] - x_data[j])
            result += sub_result
        allresult.append(result)
    return allresult
```

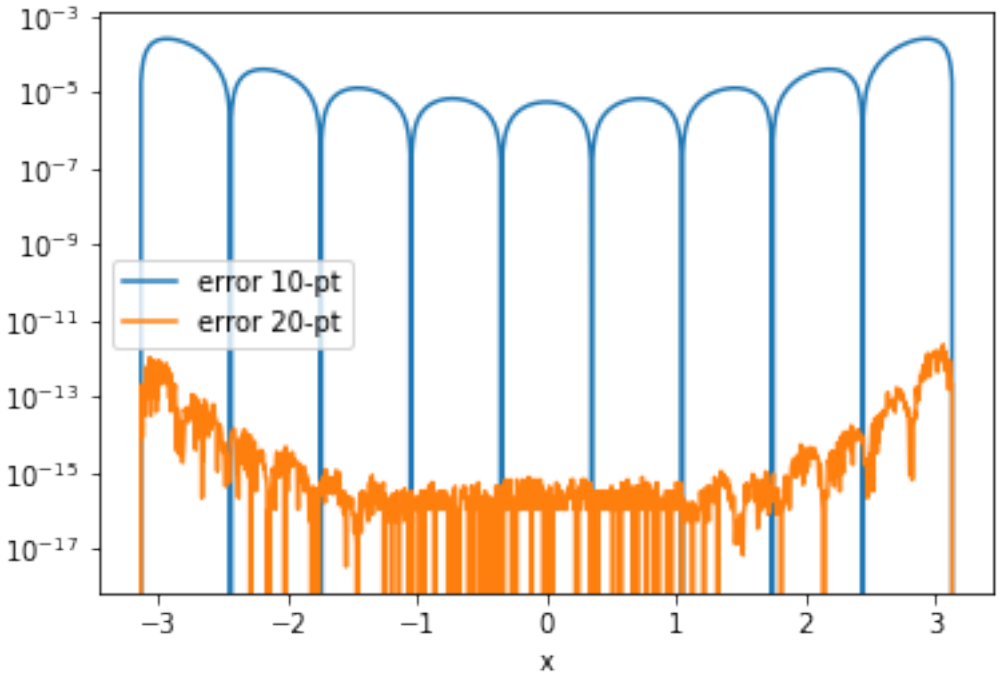
1.1 Example 1.

Interpolate $\cos x$ from an equally spaced grid of from $x = -\pi$ to $x = \pi$. We then evaluate the interplation function between $x = -\pi$ and $x = \pi$ on a much denser grid

[40]:
```
xarray_10 = np.linspace(-np.pi,np.pi, 10)
yarray_10 = np.cos(xarray_10)

xarray_20 = np.linspace(-np.pi,np.pi, 20)
yarray_20 = np.cos(xarray_20)

xvalues = np.linspace(-np.pi, np.pi, 1000)
yvalues_10 = lagrange_interp(xvalues, xarray_10, yarray_10)
yvalues_20 = lagrange_interp(xvalues, xarray_20, yarray_20)

plt.plot(xvalues, yvalues_10, label="10-point")
plt.plot(xvalues, yvalues_20, label="20-point")
plt.scatter(xarray_10, yarray_10, color='red', label="cos(x)")
plt.xlabel("x")
plt.legend();
```

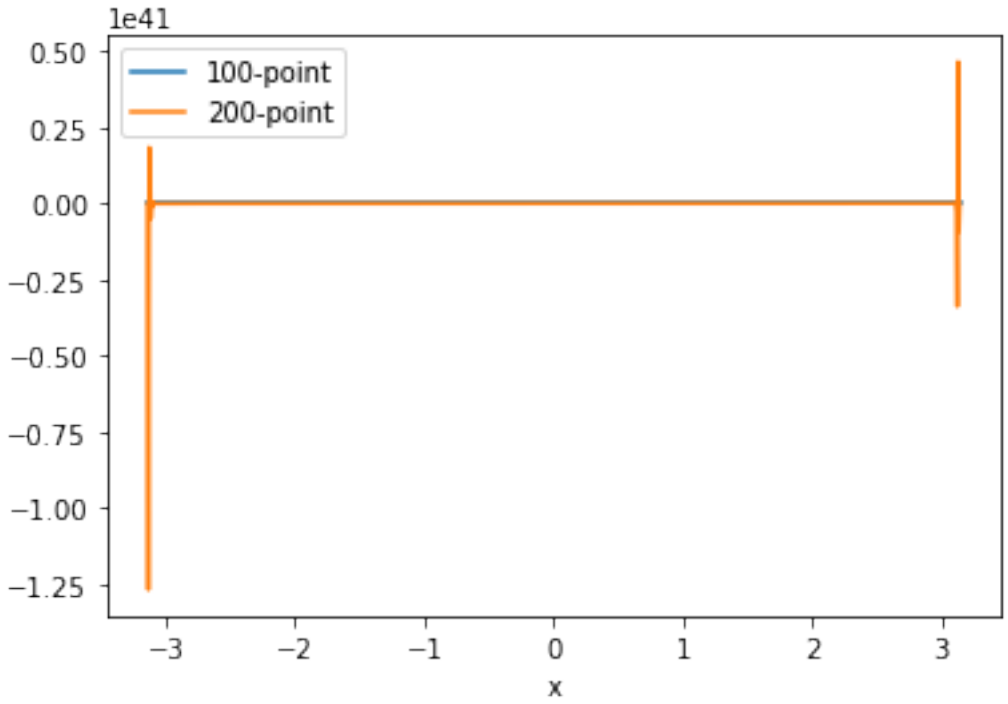


[41]:
```
plt.plot(xvalues, np.abs(yvalues_10 - np.cos(xvalues)), label="error 10-pt")
plt.plot(xvalues, np.abs(yvalues_20 - np.cos(xvalues)), label="error 20-pt")
plt.yscale('log')
plt.legend()
plt.xlabel('x');
```

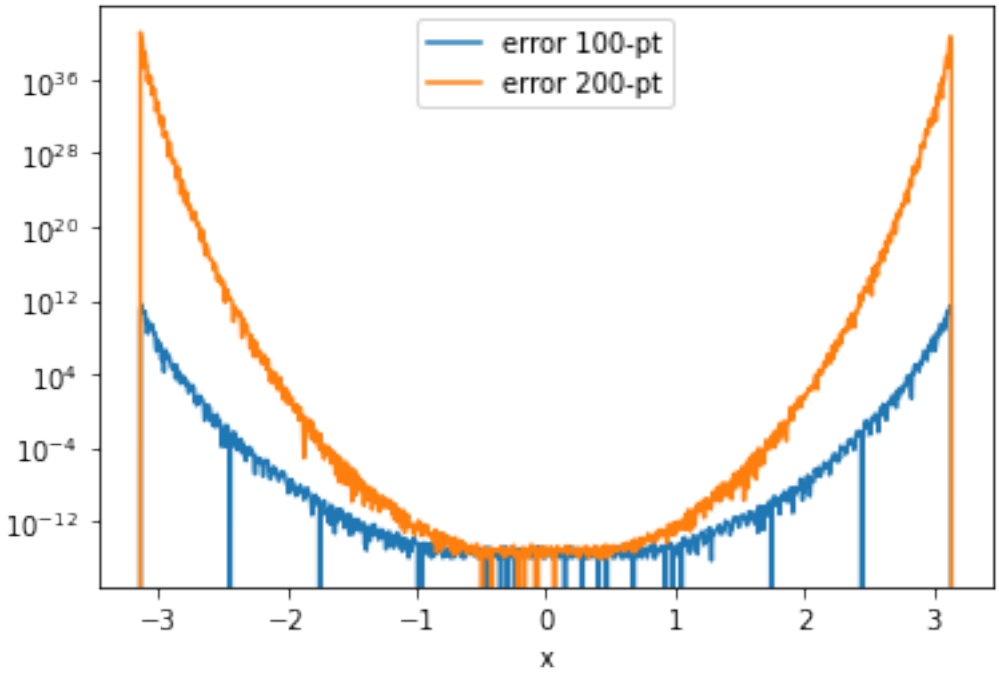


1.2 Example 2. Same function, but this time we use many more points. The instability here is entirely due to extreme roundoff effects.

[42]:
```
xarray_100 = np.linspace(-np.pi,np.pi, 100)
yarray_100 = np.cos(xarray_100)

xarray_200 = np.linspace(-np.pi,np.pi, 200)
yarray_200 = np.cos(xarray_200)

xvalues = np.linspace(-np.pi, np.pi, 1000)
yvalues_100 = lagrange_interp(xvalues, xarray_100, yarray_100)
yvalues_200 = lagrange_interp(xvalues, xarray_200, yarray_200)

plt.plot(xvalues, yvalues_100, label="100-point")
plt.plot(xvalues, yvalues_200, label="200-point")
plt.xlabel("x")
plt.legend();
```



[43]:
```
plt.plot(xvalues, np.abs(yvalues_100 - np.cos(xvalues)), label="error 100-pt")
plt.plot(xvalues, np.abs(yvalues_200 - np.cos(xvalues)), label="error 200-pt")
plt.yscale('log')
plt.legend()
plt.xlabel('x');
```



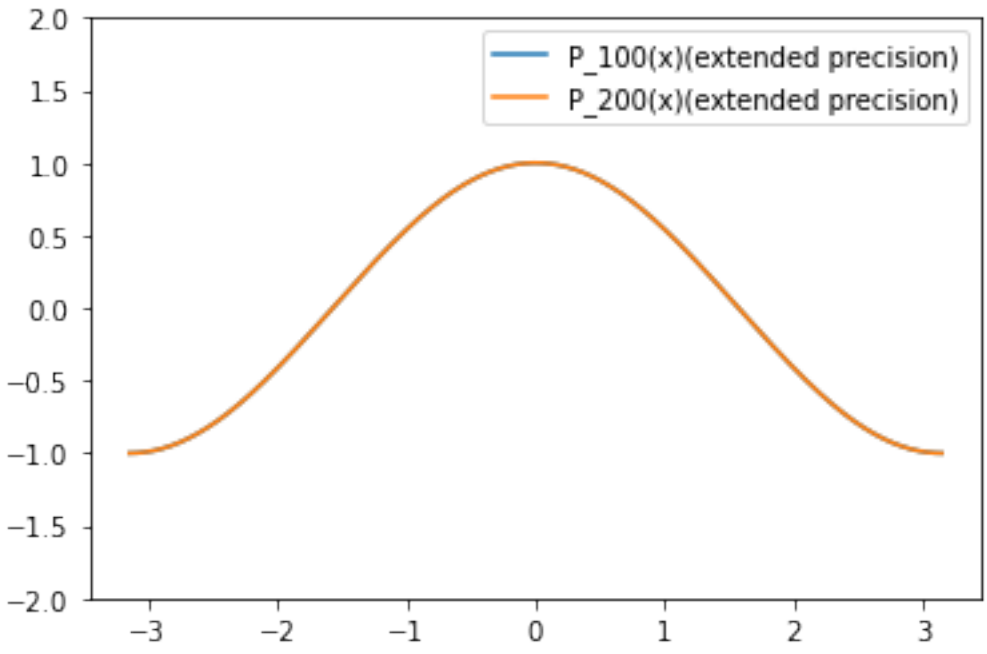
To show that this instability is due to roundoff error, we will recalculate using quadruple precision. We need to redefine the lagrange interpolation because the numba jit is not compatible with extended precision

[44]:
```
def lagrange_interp(x_val, x_data, y_data):
    assert len(x_data) == len(y_data)

    allresult = []
    for k in range(len(x_val)):
        x = x_val[k]

        result = 0;
        for i in range(len(y_data)):
            sub_result = y_data[i]
            for j in range(len(x_data)):
                if i == j:
                    continue
                sub_result *= (x - x_data[j])/(x_data[i] - x_data[j])
            result += sub_result
        allresult.append(result)
    return allresult
```

[45]:
```
pi = gmpy2.const_pi()
xarray_100 = [-pi  + 2*pi * gmpy2.mpz(i)/100 for i in range(100+1)]
yarray_100 = [ gmpy2.cos(x) for x in xarray_100 ]
```

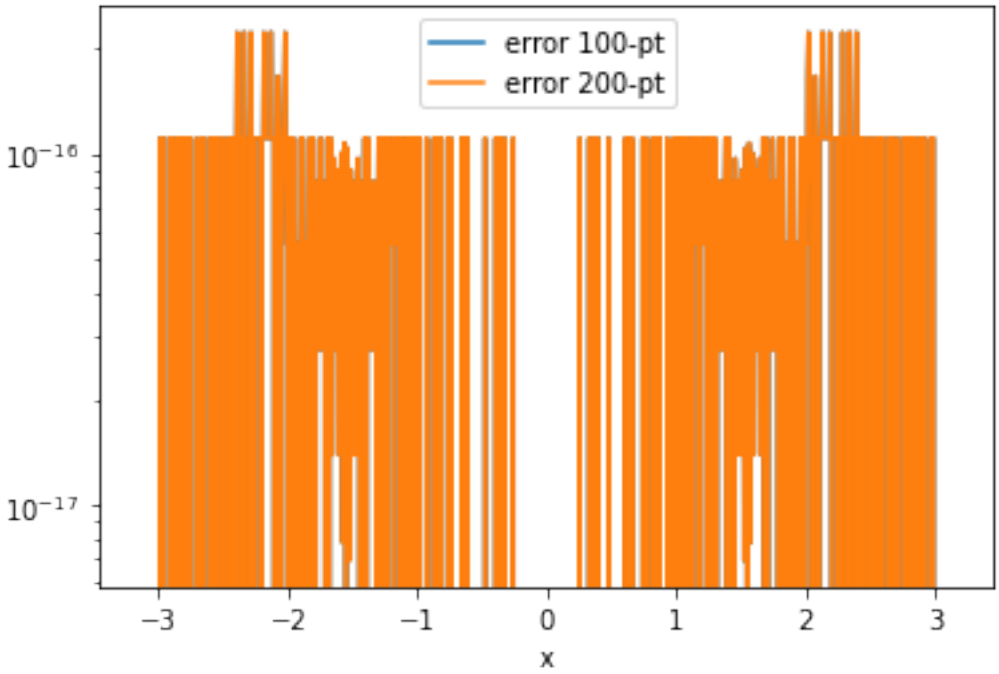
```
xarray_200 = [-pi + 2*pi * i/100 for i in range(200+1)]
yarray_200 = [gmpy2.cos(x) for x in xarray_200 ]

xvalues_1000 = [-pi + 2*pi * gmpy2.mpz(i)/1000 for i in range(1000+1)]

yvalues_100 = lagrange_interp(xvalues_1000, xarray_100, yarray_100)
yvalues_200 = lagrange_interp(xvalues_1000, xarray_200, yarray_200)


plt.plot(xvalues_1000, yvalues_100, label="P_100(x)(extended precision)")
plt.plot(xvalues_1000, yvalues_200, label="P_200(x)(extended precision)")
#plt.plot(xvalues_500, yvalues_500_100, label="P_100(x)")
#plt.plot(xvalues, yvalues - np.cos(xvalues))
plt.legend()

plt.ylim(-2,2)
```

[45]: (-2.0, 2.0)



[46]:
```
xx = np.array(xvalues_1000, dtype=float)
yy_100 = np.array(yvalues_100, dtype=float)
yy_200 = np.array(yvalues_200, dtype=float)

plt.plot(xvalues_1000, np.abs(yy_100 - np.cos(xx)), label="error 100-pt")
```

```
plt.plot(xvalues_1000, np.abs(yy_200 - np.cos(xx)), label="error 200-pt")
plt.yscale('log')
plt.legend()
plt.xlabel('x');
```



1.3 Example 2. Runge's function

Here we consider the interpolation of

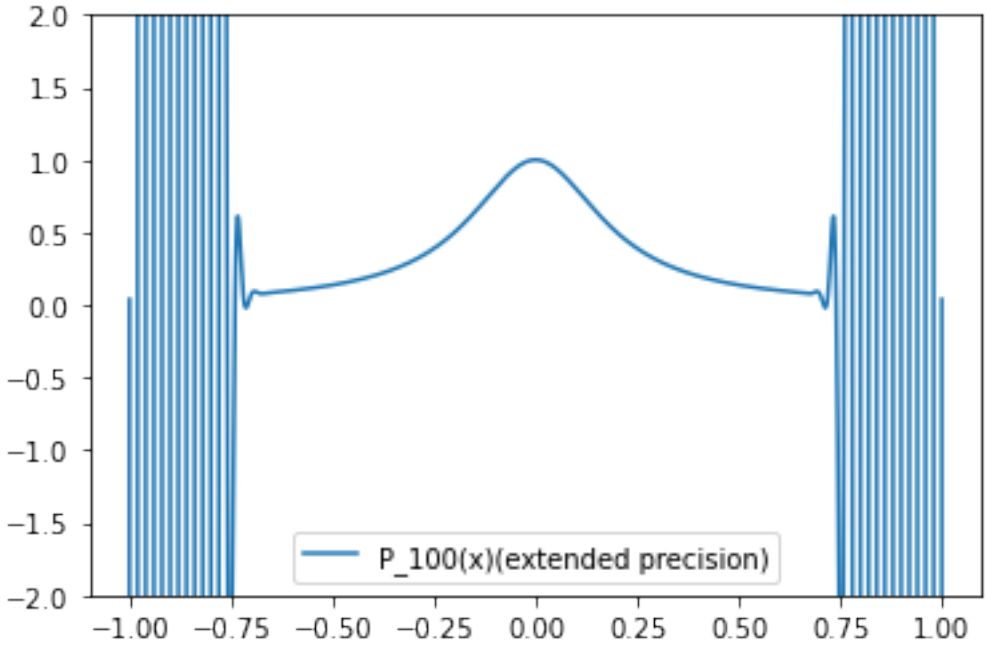
$$f(x) = \frac{1}{1+25x^2}, \quad -1 < x < 1$$

The wild fluctuations are *not* due to roundoff errors. These are errors inherent in the method.

[49]:
```
def Runge(x):
    return 1 / (1 + 25 * x**2)
xarray_100 = [-1  + 2* gmpy2.mpz(i)/100 for i in range(100+1)]
yarray_100 = [ Runge(x) for x in xarray_100 ]


xvalues_1000 = [-1 + 2* gmpy2.mpz(i)/1000 for i in range(1000+1)]

yvalues_100 = lagrange_interp(xvalues_1000, xarray_100, yarray_100)


plt.plot(xvalues_1000, yvalues_100, label="P_100(x)(extended precision)")
```

```
#plt.plot(xvalues_500, yvalues_500_100, label="P_100(x)")
#plt.plot(xvalues, yvalues - np.cos(xvalues))
plt.legend()

plt.ylim(-2,2)
```

[49]: (-2.0, 2.0)



[ ]: