Primer:

$$f(x) = f(x_0) + \frac{f^{(1)}(x_0)}{1!}(x-x_0) + \frac{f^{(2)}(x_0)}{2!}(x-x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x-x_0)^3 + \ldots$$

$$f(x+kh) = f(x_0) + \frac{f^{(1)}(x_0)}{1!}(x-x_0+kh) + \frac{f^{(2)}(x_0)}{2!}(x-x_0+kh)^2 + \frac{f^{(3)}(x_0)}{3!}(x-x_0+kh)^3 + \ldots$$

$$f(x_0+kh) = f(x_0) + \frac{f^{(1)}(x_0)}{1!}(kh) + \frac{f^{(2)}(x_0)}{2!}(kh)^2 + \frac{f^{(3)}(x_0)}{3!}(kh)^3 + \ldots$$

Therefore:

$$\frac{f(x_0+h) - f(x_0)}{h} = f^{(1)}(x_0) + O(h)$$

$O(h)$ because divide by h at the end so remaining terms are dominated by h^1 term

$$f(x_0+2h) = f(x_0) + \frac{f^{(1)}(x_0)}{1!}(2h) + \frac{f^{(2)}(x_0)}{2!}(2h)^2 + \frac{f^{(3)}(x_0)}{3!}(2h)^3 + \ldots$$

So to find $f^{(k)}(x_0)$ we need to solve a linear equation for coefficients

Things that make it easier:

(1) The derivative terms all remain the when x is changed so they can be ignored when solving for coefficients

(2) We can only have derivatives up to the $1+$ the number of supplied points (so for 2nd derivative need at least 3 points)

(3) We can have more points than that which will increase the order of the accuracy

Equation form:

$$A\vec{f^{(k)}} = f(x_0 \vec{+} kh)$$

$$A = \begin{bmatrix} k_0^0 & k_0^1 & k_0^2 & \cdots \\ k_1^0 & k_1^1 & k_1^2 & \cdots \\ k_2^0 & k_2^1 & k_2^2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} ; \vec{f^{(k)}} = \begin{bmatrix} f(x_0) \\ f'(x_0)h \\ \frac{f''(x_0)}{2!}h^2 \\ \vdots \end{bmatrix} ; f(x_0 \vec{+} kh) = \begin{bmatrix} f(x_0 + k_0) \\ f(x_0 + k_1) \\ f(x_0 + k_2) \\ \vdots \end{bmatrix}$$

Since A is a square coefficient matrix it's probably invertable lol, so we get:

$$\vec{f^{(k)}} = A^{-1} f(x_0 \vec{+} kh)$$

and with an extra processing step:

$$\vec{f^{(k)}} = A^{-1} f(x_0 \vec{+} kh) \circ [\frac{n!}{h^n}]_{n \geq 0}$$

This resulting form can be truncated to any number of points as long as there are more than the derivative's order.

In [ ]:
```
import numpy as np
from scipy.special import factorial
from scipy.linalg import inv
```

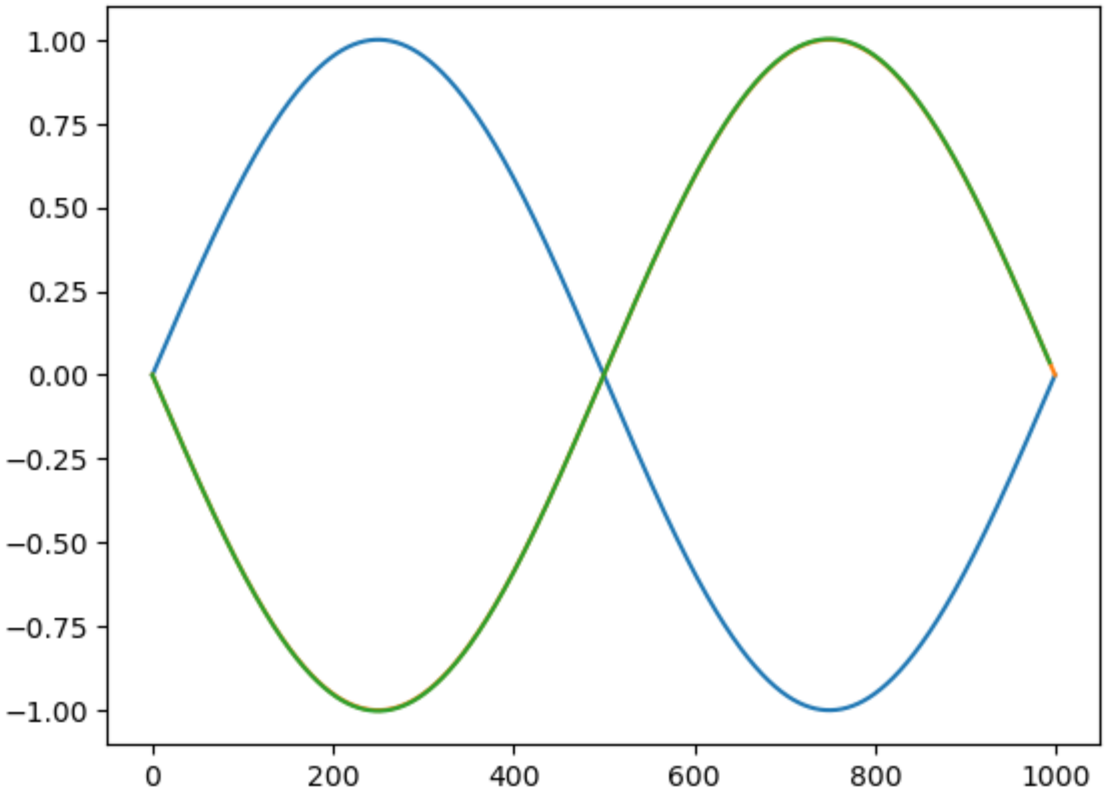
In [ ]:
```
# Each k_offset represents a function evaluation at x_0 + h*k_off; f(x_0 + h*k_off)
# Currently only works with integer offsets because I haven't implemented interpolation to get fractional step predictions
def finite_diff(deriv, k_offsets):
    assert deriv >= 0, "Does not extend to negative derivatives."
    assert deriv < len(k_offsets), "Not enough sampled points for derivative."

    num_k_off = len(k_offsets)
    index_arr = np.linspace(0, num_k_off-1, num_k_off)

    taylor_coeff_m = np.stack([np.power(k_off, index_arr) for k_off in k_offsets])
    finite_diff_coeff_m = inv(taylor_coeff_m)

    def get_diff(fs, h):
        assert len(fs) == num_k_off, f"Incorrect number of supplied points, expected: {num_k_off} at offsets: {k_offsets}"
```

```python
        fdiff_mult_by = factorial(index_arr)/np.power(h, index_arr)
        derivative_results_m = (finite_diff_coeff_m @ fs)*fdiff_mult_by
        return derivative_results_m[deriv]
    return get_diff
```

In [ ]:
```python
x_i, x_f, nx = 0, 2*np.pi, 1000

sintest = np.sin(np.linspace(x_i, x_f, nx))
costest = np.cos(np.linspace(x_i, x_f, nx))
h = (x_f - x_i)/nx

deriv = 2 # change me
k_offsets = np.array([0, 2, 4, 6]) # change me

sin_diff = finite_diff(deriv, k_offsets)
costest_d = [sin_diff(sintest[i+k_offsets], h) for i in range(len(sintest[abs(min(k_offsets)):-max(k_offsets)]))]
```

In [ ]:
```python
import matplotlib.pyplot as plt

plt.plot(sintest)
plt.plot(-sintest)
plt.plot(costest_d)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x271d6d6ff40>]
```



In [ ]: