# Phys 222 - HW 1

**Issar Amro**

**Fall 2023**

## 1 Lagrange Interpolation

Given an order n, to construct a curve between every 2 consecutive points, we want to evaluate this:

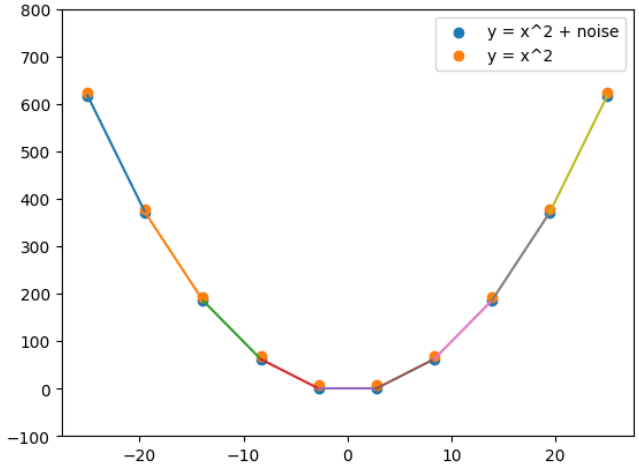
$$y = \sum_{j=0}^{n} f(x_j) \prod_{j=0}^{n} \frac{x - x_m}{x_j - x_m}$$

where $m \neq j$.

### 1.1 First order Lagrange interpolation.

**code:**

```
x = np.linspace(-25,25, 10)
noise = 1.5*np.random.randint(-10, 10)
y = x**2 + noise
y1 = x**2
plt.scatter(x, y, label = "y = x^2 + noise")
plt.scatter(x, y1, label = "y = x^2")
for i in range(0, len(x)-1):
    x1 = x[i]
    x2 = x[i+1]
    y1 = y[i]
    y2 = y[i+1]
    j = 1 # order
    i = np.linspace(x1,x2)
    p = y1 * ((i - x2)/(x1 - x2)) + y2 *((i-x1)/(x2-x1))
    plt.plot(i,p)
plt.ylim(-100, 800)
plt.legend()
plt.show()
```



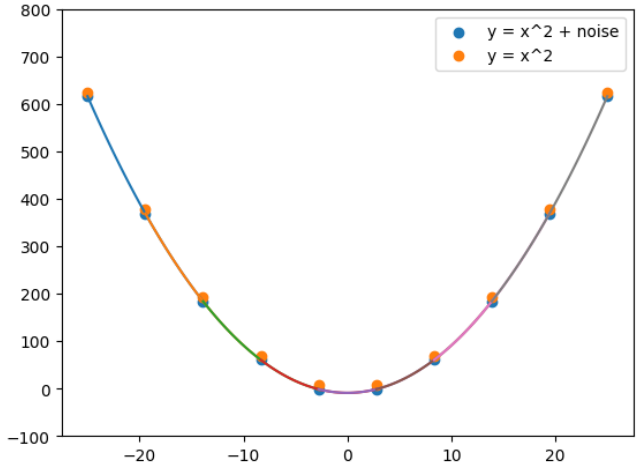

**notes.**

Each line passes by 2 consecutive data points and the collection of the lines forms an approximation, a bad one, of the function.

## 1.2 Second order Lagrange interpolation.

### code:

```
x = np.linspace(-25,25, 10)
noise = 1.5*np.random.randint(-10, 10)
y = x**2 + noise
y1 = x**2
plt.scatter(x, y, label = "y = x^2 + noise")
plt.scatter(x, y1, label = "y = x^2")
for i in range(0, len(x)-2):
    x1 = x[i]
    x2 = x[i+1]
    x3 = x[i+2]
    y1 = y[i]
    y2 = y[i+1]
    y3 = y[i+2]
    j = 2 # order
    i = np.linspace(x1, x3)
    p = y1 * ((i - x2)/(x1 - x2))*((i-x3)/(x1-x3)) +
y2 *((i-x1)/(x2-x1))*((i-x3)/(x2-x3)) +
y3*((i-x2)/(x3-x2))*((i-x1)/(x3-x1))
    plt.plot(i,p)
plt.ylim(-100,800)
plt.legend()
plt.show()
```



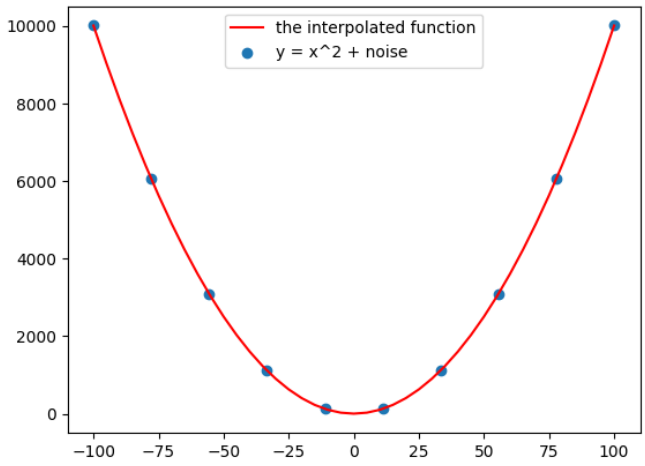

**notes.**

Looping over my data points and using 3 points at a time in the Lagrange polynomials gave me a decent, almost perfect, approximation of the function.

### 1.3 Defining the order as the number of data points - 1.

**code:**

```
x = np.linspace(-100,100, 10)
y = x**2 + 1.5*np.random.randint(-20,20)
y1 = x**2
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1], 5):
    x_given = i
    x_given_int.append(x_given)
    y_we_seek = 0
    for i in range(j+1):
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
plt.scatter(x,y, label = "y = x^2 + noise")
plt.scatter(x,y1, label = "y = x^2")
plt.legend()
plt.show()
```



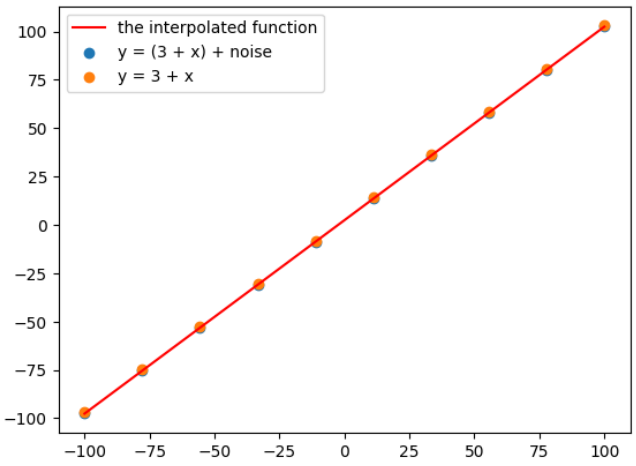

**notes.**

I figured here to go to the highest order which is equal to the number of my data points - 1 and fill the points between each 2 consecutive data points. I stored the "given x" points in an array that represents points between 2 consecutive data points and the corresponding "expected y" points that I got from the interpolation and then plotted the function.

### 1.4 Trying the above on a linear function.

**code:**

```
x = np.linspace(-100,100, 10)
y = 3 + x + 0.2*np.random.randint(-20,20)
y1 = 3 + x
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1], 5):
    x_given = i
    x_given_int.append(x_given)

    y_we_seek = 0
    for i in range(j+1):
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
plt.scatter(x,y, label = "y = (3 + x) + noise")
plt.scatter(x,y1, label = "y = 3 + x")
plt.legend()
plt.show()
```




**notes.**

I wanted to see if higher orders also work on simple linear functions.

# 2 Aitken - Neville Interpolation

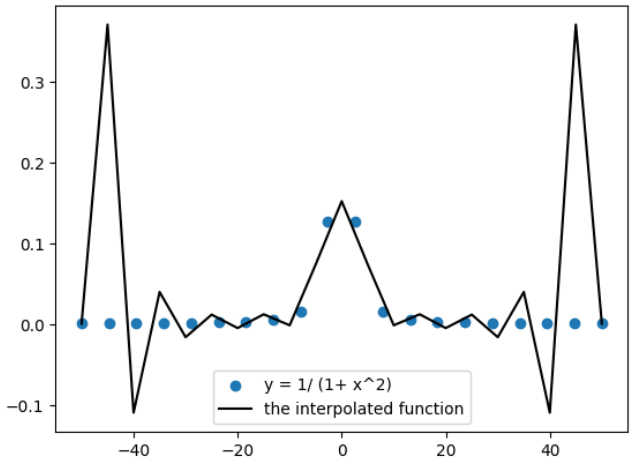
Using the Lagrange polynomials in a recursive way.

## 2.1 For y = $\frac{1}{1+x^2}$

**code:**

```
x = np.linspace(-50,50, 20)
y = 1 / (1 + x**2)
def AN(x, x_given, y, initial, final):
     if initial == final:
         ans = y[final]
     else:
        ans =
        (x_given-x[final])/(x[initial] - x[final]) *
        AN(x,x_given, y,initial, final -1 ) +
        (x_given-x[initial])/(x[final] - x[initial]) *
        AN(x,x_given,y, initial +1, final)
     return ans
x_given = 0
x_given_int = []
y_expected = []
for i in np.arange(x[0], x[len(x) - 1] + 1, 5):
        x_given = i
        x_given_int.append(x_given)
        y_expected.append(AN(x, x_given, y, 0, 19))
plt.scatter(x,y, label = "y = 1/ (1+ x^2)")
plt.plot(x_given_int, y_expected, color = "k", label = "the interpolated function")
plt.legend()
```

```
plt.show()
```




**notes.**

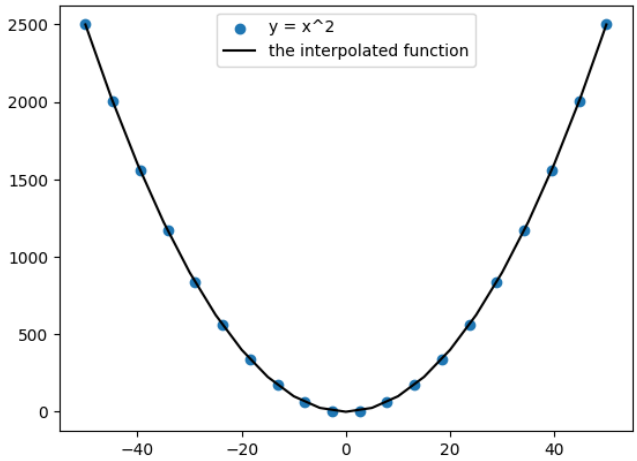
I took the order to be equal to the number of my data points - 1. The interpolated function seems to be a good approximation of the original function but only in a small interval around x = 0, on the edges, we notice unwanted oscillations with big magnitude.

## 2.2 For $y = x^2$

### code:

```
x = np.linspace(-50,50, 20)
y = x**2 + 0.2
def AN(x, x_given, y, initial, final):
     if initial == final:
         ans = y[final]
     else:
        ans =
        (x_given-x[final])/(x[initial] - x[final]) *
        AN(x,x_given, y,initial, final -1 ) +
        (x_given-x[initial])/(x[final] - x[initial]) *
        AN(x,x_given,y, initial +1, final)
     return ans
x_given = 0
x_given_int = []
y_expected = []
for i in np.arange(x[0], x[len(x) - 1] + 1, 5):
        x_given = i
        x_given_int.append(x_given)
        y_expected.append(AN(x, x_given, y, 0, 19))
plt.scatter(x,y, label = "y = x^2")
plt.plot(x_given_int, y_expected, color = "k", label = "the interpolated function")
```

```
plt.legend()
plt.show()
```



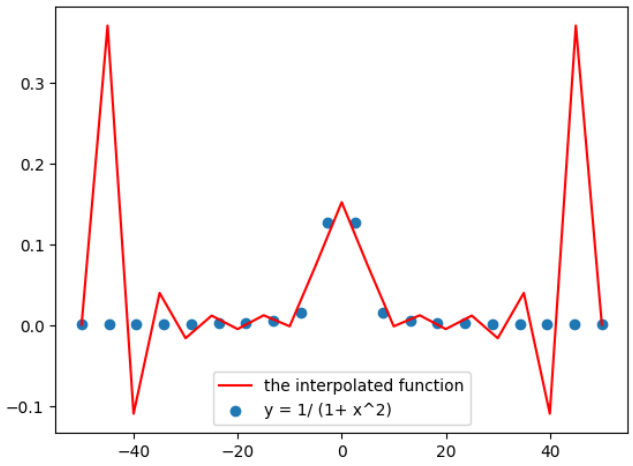

**notes.**

To make sure that the Aitken Neville code works, I used it on the data points that I used Lagrange interpolation on. The interpolated function is a perfect approximation for this function meaning the unwanted oscillations don't always occur (at least not for simple functions) and my code works fine.

## 2.3 Trying Lagrange interpolation on $y = \frac{1}{1+x^2}$

**code:**

```
x = np.linspace(-50,50, 20)
y = 1 / (1+x**2)
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1] +1, 5):
    x_given = i
    x_given_int.append(x_given)
    y_we_seek = 0
    for i in range(j+1):
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
plt.scatter(x,y, label = "y = 1/ (1+ x^2)")
plt.legend()
plt.show()
```




**notes.**

I tried the Lagrange Interpolation on $y = \frac{1}{1+x^2}$ to make sure the oscillations were not an error in the Aitken Neville code. The results were exactly like in the Aitken Neville case. Meaning the higher order polynomials may lead to "over fitting".

# 3 Splines Interpolation

## 3.1 Quadratic splines.

Given n+1 data points, we can derive n equations of the curves between each 2 consecutive points which gives us 2 evaluations, so we end up with 2n equations with 3n unknowns. We impose continuity conditions on these equations at the inner points (we want our fit to be smooth) which gives us n-1 equations. And, finally, we fix the first variable ($a_1 = 0$) such that the first curve would be linear.
We then solve for the 3n-1 equations to find the coefficients of the curves between each 2 consecutive data points:

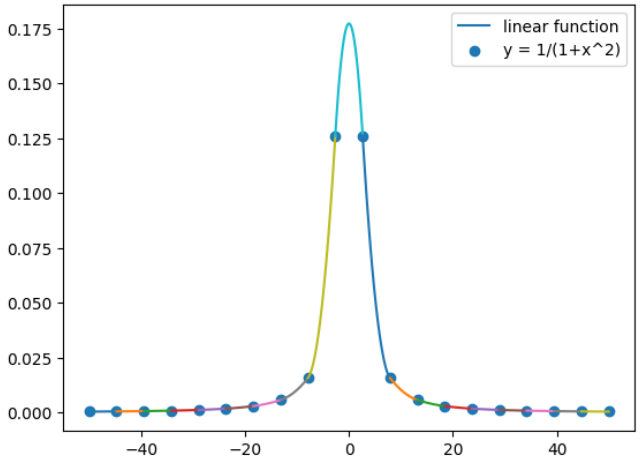
$$a_i x_{i-1}^2 + b_i x_{i_-1} + c_i = y_{i_-1} \quad \forall i \in [1, n]$$
$$a_i x_i^2 + b_i x_i + c_i = y_i \quad \forall i \in [1, n]$$
$$2a_i x_i - 2a_{i+1} xi + b_i - b_{i+1} = 0 \quad \forall i \in [1, n-1]$$

**code:**

```
x = np.linspace(-50,50, 20)
y = 1 / (1+ x**2)
n = len(x) - 1
x_matrix = np.zeros((3*n, 3*n))
y_matrix = np.zeros((3*n - 1, 1))
# fill 2n rows:
for i in range(0, 2*n -1, 2):
    row1 = []
    row2 = []
    for j in range(i - int(i/2)):
```

```
            row1.append(0)
            row1.append(0)
            row1.append(0)
            row2.append(0)
            row2.append(0)
            row2.append(0)
        row1.append(x[i-int(i/2)]**2)
        row1.append(x[i - int(i/2)])
        row1.append(1)
        row2.append(x[i-int(i/2) + 1 ]**2)
        row2.append(x[i-int(i/2) + 1])
        row2.append(1)
        if len(row1) < 3*n:
            for o in range(3*n - len(row1)):
                row1.append(0)
                row2.append(0)
        x_matrix[i] = row1
        x_matrix[i+1] = row2
    # fill n -1 rows:
    for i in range(1, n):
        row = []
        for j in range(1, i):
            row.append(0)
            row.append(0)
            row.append(0)
        row.append(2*x[i])
        row.append(1)
        row.append(0)
        row.append(- 2* x[i])
        row.append(-1)
        if len(row) < 3*n:
            for o in range(3*n - len(row)):
                row.append(0)
        x_matrix[2*n - 1 + i] = row
    # last row of zeros
    # fill y matrix:
    y_matrix[0] = y[0]
    y_matrix[2*n - 1] = y[len(y) -1]
    for i in range(1, 2*n -1, 2):
        y_matrix[i] = y[i + 1 - int((i+1)/2)]
        y_matrix[i+1] = y[i + 1 - int((i+1)/2)]
    x_matrix = np.delete(x_matrix, obj = 0, axis = 1)
    x_matrix = np.delete(x_matrix, obj = 3*n - 1, axis = 0)
    unknown_matrix = np.linalg.solve(x_matrix, y_matrix)
    # a1 = 0
    i = np.linspace(x[0], x[1])
    j = unknown_matrix[0]*i + unknown_matrix[1]
    plt.plot(i,j, label = "linear function")
    for i in range(1, n):
```

```
        o = np.linspace(x[i], x[i+1])
        l = unknown_matrix[3*i - 1] * o**2 +
        unknown_matrix[3*i] * o + unknown_matrix[3*i + 1]
        plt.plot(o,l)
plt.scatter(x,y, label = "y = 1/(1+x^2) + noise")
plt.legend()
plt.show()
```




**notes.**

I constructed a $5 \times 5$ matrix by hand and noticed a pattern that I used in the code to construct any $3n \times 3n$ matrix (the x matrix). then I removed the first column and last row. Did the same for the y matrix then solved for the unknown matrix. I then plotted the linear function relating the first 2 points (I set $a_1 = 0$) and looped over each 2 consecutive points to plot the second order functions. (i want to try cubic and n order spline)

# 4 L2 Optimization using Vandermonde matrix

Given n+1 data points and a polynomial of order m:

$$P_m(x) = \sum_{i=0}^{m} a_i x^i$$

We introduce the sum of the square of the deviations:

$$\chi^2 = \sum_{i=0}^{n} \left[p_m(x_i) - f(x_i)\right]^2$$

We want to find the coefficients that would minimize the deviations of a global fit approximation $p_m(x)$ of the initial function $f(x)$. To do that, we set the derivatives of $\chi^2$ wrt the coefficients to be 0:

$$\frac{\partial \chi^2}{\partial a_i} = 0$$

Alternatively, we set the Valindrome matrix which is a (nb of points $\times$ order) matrix :

$$V = \begin{pmatrix} x_0^n & x_0^{n-1} & ... & x_0 & 1 \\ x_1^n & x_1^{n-1} & ... & x_1 & 1 \\ . & . & . & . & . \\ . & . & . & . & . \end{pmatrix}$$

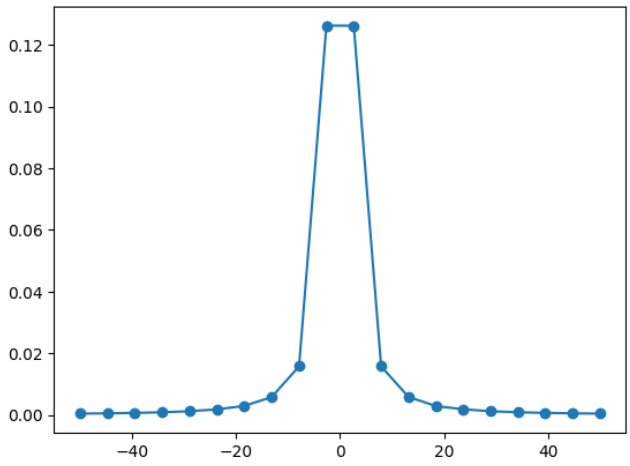
We want to solve this system:

$$\begin{pmatrix} x_0^n & x_0^{n-1} & ... & x_0 & 1 \\ x_1^n & x_1^{n-1} & ... & x_1 & 1 \\ . & . & . & . & . \\ . & . & . & . & . \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ . \end{pmatrix}$$

The solution would be $A = (V^T V)^{-1} V^T y$. And then we plug in the coefficients we found in the global fit.

## code:

```
x = np.linspace(-50,50,20)
y = 1/(1+x**2)
n = len(x)
order = 18
v = np.zeros((n, order+1))
for i in range(n):
    row = []
    for j in range(order+1):
        row.append(x[i])
    v[i] = row
exp = []
for i in range(order, -1, -1):
    exp.append(i)
v = v ** exp
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)
p = 0
for i in range(order+1):
    p += coef_matrix[len(coef_matrix) - (i+1)]* x**i
plt.plot(x, p)
plt.scatter(x,y)
plt.show()
```




**notes.**

The optimal order of the global fit polynomial is dependent on the number of my data points, at least for complex functions like $y = \frac{1}{1+x^2}$. However, for less complex functions like $y = x^2$, order 2 is perfect (which is expected).

# 5 Testing vs Training errors

(still working on it)

# 6 Approximating Derivatives

We start with a Taylor series expansion of our function about a point $x_0$ (the point at which we want to find the derivative):

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + O(h^4)$$

## Approximating the first derivative

**First try:** we cut the series at the first order and we get

$$f'(x_0) = \frac{f(x) - f(x_0)}{h} + O(h)$$

where $h = x - x_0$ represents the distance from $x$ to the nearest point on its left or right. This 1st order approximation requires 1 extra data point ($x$) only.

**Second try:** we cut the series at the second order and we introduce

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x) + O(h^3)$$

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x) + O(h^3)$$

subtracting them, we get

$$f(x_0+h)-f(x_0-h)=2hf'(x_0)+O(h^3)$$

and hence

$$f'(x_0)=\frac{f(x_0+h)-f(x_0-h)}{2h}+O(h^2)$$

For this 2nd order approximation, we need 2 points at an equal distance h from $x_0$ ($x_0+h$ and $x_0-h$).

**Third try:** we cut the series at the third order and we introduce

$$f(x_0+2h)=f(x_0)+2hf'(x_0)+\frac{4h^2}{2}f''(x)+\frac{8h^3}{6}+O(h^4)$$

$$f(x_0-2h)=f(x_0)-2hf'(x_0)+\frac{4h^2}{2}f''(x)-\frac{8h^3}{6}f'''(x_0)+O(h^4)$$

subtracting them we get

$$f(x_0+2h)-f(x_0-2h)=4hf'(x_0)+\frac{8h^3}{3}f'''(x_0)+O(h^5)$$

to find the $f'''(x_0)$ term, we return to

$$f(x_0+h)-f(x_0-h)=2hf'(x_0)+\frac{h^3}{3}f'''(x_0)+O(h^5)$$

hence

$$\frac{h^3}{3}f'''(x_0)=f(x_0+h)-f(x_0-h)-2hf'(x_0)+O(h^5)$$

replacing, we get

$$f(x_0+2h)-f(x_0-2h)=4hf'(x_0)+8f(x_0+h)-8f(x_0-h)-16hf'(x_0)+O(h^5)$$

and finally

$$f'(x_0)=\frac{f(x_0-2h)-f(x_0+2h)-8f(x_0-h)+8f(x_0+h)}{12h}+O(h^4)$$

For this 4th order approximation, we need 4 points at equal distances from $x_0$ ($x_0+h$, $x_0-h$, $x_0+2h$ and $x_0-2h$). Avoiding the derivative at the first 2 and last 2 points, we can find the derivatives at all the inner points by transforming the above formula for the second derivative into matrix form:

$$f(x_i)\ \ \forall i\in(2,n-2)=$$

$$\frac{1}{12h}\begin{pmatrix}1&-8&0&8&-1&0&.&.&.&.\\0&1&-8&0&8&-1&0&.&.&.\\.&&&&&&&&&\\.&&&&&&&&&\\.&.&.&.&0&1&-8&0&8&-1\end{pmatrix}\begin{pmatrix}f(x_0)\\f(x_1)\\f(x_2)\\.\\.\end{pmatrix}$$

### code:

```
x = np.linspace(-50,50,10)
y = 2*x**3
n = len(x)
m = x[2] - x[1]

    def FirstDer4thOrder(n):
    O = np.empty((n-4,n))
    for i in range (n-4):
        for j in range(n):
          row = []
          for o in range(i):
            row.append(0)
          row.append(1)
          row.append(-8)
          row.append(0)
          row.append(8)
          row.append(1)
          for p in range(n - len(row)):
            row.append(0)
        O[i] = row
    return O

O = FirstDer4thOrder(n)
der1_matrix = 1/(12*m) * (O @ np.transpose(y))
```

**comparing analytical values to numerical approximation.**

**Tao: We can, of course, make the accuracy even higher by including more points, but in many cases this is not good practice. For real problems, the derivatives at points close to the boundaries are important and need to be calculated accurately. The errors in the derivatives of the boundary points will accumulate in other points when the scheme is used to integrate an equation. The more points involved in the expressions of the derivatives, the more difficulties we encounter in obtaining accurate derivatives at the boundaries. Another way to increase the accuracy is by decreasing the interval h.**

## Approximating the second derivative

### 1D Grid:

We use $f(x_0 + h) + f(x_0 - h) = 2f(x_0) = h^2 f''(x_0) + O(h^4)$, and we find

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + O(h^2)$$

We need 2 points besides $x_0$, one right before and one right after $x_0$. Avoiding the derivative at the first and last points, we can find the derivatives at all the inner points by transforming the

above formula for the second derivative into matrix form:

$$\nabla^2 = \frac{1}{h^2}\begin{pmatrix} 1 & -2 & 1 & 0 & 0 & . & . & . \\ 0 & 1 & -2 & 1 & 0 & . & . & . \\ 0 & 0 & 1 & -2 & 1 & 0 & . & . \\ . & . & . & . & 0 & 1 & -2 & 1 \end{pmatrix}\begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ . \\ . \end{pmatrix} = -D + A$$

where $D$ is a diagonal matrix called the degree matrix. And $A$ denotes the equally spaced neighbor points.

## code:

```
x = np.linspace(-50,50,10)
y = 2*x**3
n = len(x)
m = x[2] - x[1]

    def LaplacianMatrix(n):
    D = np.empty((n-2,n))
    for i in range (n-2):
        for j in range(n):
          row = []
          for o in range(i):
            row.append(0)
          row.append(1)
          row.append(-2)
          row.append(1)
          for p in range(n - len(row)):
            row.append(0)
        D[i] = row
    return D

D = LaplacianMatrix(n)
der2_matrix = 1/(m**2)* (D @ np.transpose(y))
```

**comparing analytical values to numerical approximation.**

For our purposes, the Laplacian is used in the diffusion equation to describe how information diffuses in a network:

$$\frac{\partial C}{\partial t} = D\nabla^2 C$$

To use this logic, we have to abandon the notion of the Euclidean distance and define a new meaning for "distance" in each context. For instance, if we want to study a Facebook network that includes 3 elements/users {R,S,G}, we can use a matrix notation to deduce the "distance" between each 2:

$$\begin{matrix} & R & S & G \\ R \\ S \\ G \end{matrix}\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

where 1 represents a connection on Facebook (they follow each other). The "distance" between R and G is of magnitude 1 since they are directly connected. And the "distance" between R and S is of magnitude 2 since R is connected to G and G is connected to S.
Back to the diffusion equation, it is a linear heat equation (linear evolution equation) whose solution is dependent on the eigenvalues of the Laplacian matrix and is of the form $\mathbf{C}(t) = e^{\lambda t}\mathbf{v}$.

**Solution to the diffusion equtaion**

I will be discussing this problem (for fun).
To solve a linear heat equation of the form $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$, we start with the elementary scalar ODE $\frac{\partial u}{\partial t} = \lambda u$ where $\lambda$ is a scalar. We know the solution is of the form $u(t) = ce^{\lambda t}$.
Applying this logic in the case when we have $\frac{\partial \mathbf{u}}{\partial t} = A\mathbf{u}$ where $A$ is an $n \times n$ matrix (the Laplacian matrix), we suggest (by analogy with the previous case) the ansatz $\mathbf{u}(t) = e^{\lambda t}\mathbf{v}$. Differentiating wrt time, we get the condition $A\mathbf{v} = \lambda\mathbf{v}$ which is an eigenvalue problem that has a nonzero solution $\mathbf{v} \neq 0$ if and only if $\lambda$ is an eigenvalue of $A$ and $\mathbf{v}$ a corresponding eigenvector.
Note that if $\nabla^2$ has 2 eigenvalues, the physical meaning of the largest one is the speed of diffusion and the smallest one is the relaxation time (time needed for the information to reach the whole network).

**2D Grid:**

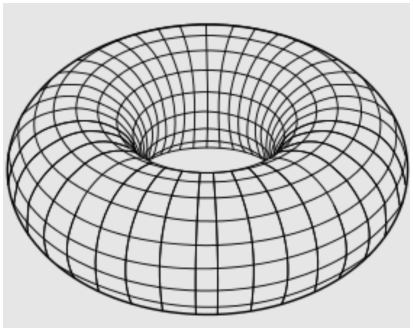
For higher dimensions, specifically working with 2 dimensions here, the formula for the second derivative becomes

$$f''(x_0, y_0) = \frac{f(x_0 + h, y_0) - 4f(x_0, y_0) + f(x_0 - h, y_0) + f(x_0, y_0 + h) + f(x_0, y_0 - h)}{h^2}$$

which now requires 4 neighboring points (2 to the left and right of $x_0$ and 2 above and below $x_0$). In general, the number of the required neighboring points for the second derivative is equal to $2^d$ where $d$ is the number of dimensions.

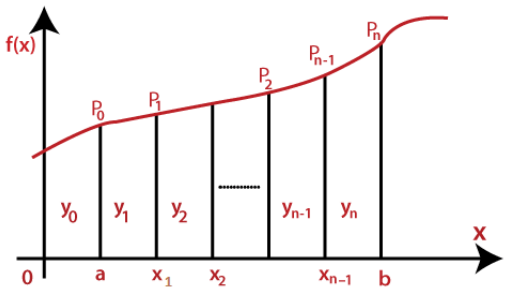
**Boundaries Problem**

How to evaluate the derivatives at the boundary point?
For the first derivative, we can only use the first order approximation and use the point right before/after the boundary point. But if we want more precision or we want to calculate the second derivative (we need 2 neighboring points minimum), we manipulate the topology of the grid. we could fold the 1D grid (line) into a circle and we could fold the 2D grid in a way that it forms a torus.



# 7 Approximating Integrals

## 1- Trapezoidal rule



Integration by the trapezoidal rule

For "curvy" functions, approximating the integral over an interval by summing over the area of trapezoids is more effective than the Riemann sum. We can approximate the area under the curve by:

$$\sum_{i=1}^{i=n-1} y_i \Delta x + (y_0 + y_n)\frac{\Delta x}{2}$$

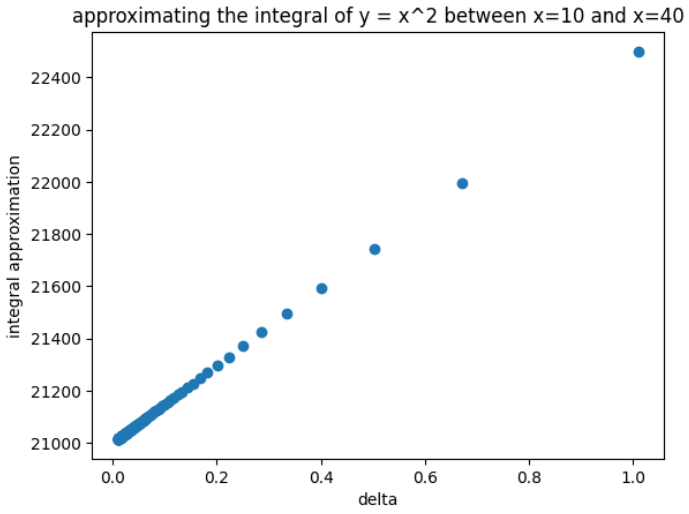
Where $n$ is the number of data points with a given interval $[x_0, x_n]$.

• **First try:** as a first try to code the trapezoidal rule, I generated a data set from the function $y = x^2$ and interpolated over them using L2 optimization method. I fixed my interval using the coefficients I got from the Vandermonde matrix. Then, I started varying the number of data points generated, thus changing the width of the trapezoids, and plotted the outputted values of the approximated integral as a function of the decreasing width of the trapezoids.

**code.**

```
def TrapRule1(x, x_0, x_n, p_0, p_n, delta):
    approx_int = (p_0 + p_n) * (delta/2)
    for i in range(n):
        if x[i] > x_0 and x[i] < x_n:
            approx_int += y[i] * delta
    return approx_int
approx_int = []
delta_change = []

x_0 = 10
x_n = 40

for i in range(100, 10000, 50):
   x = np.linspace(-50,50,i)
   delta = x[1]-x[0]
   order = 2
```

```
    n = len(x)
    y = x**2
    v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
    v_new = np.transpose(v) @ v
    y_matrix = np.transpose(v) @ y
    coef_matrix = np.linalg.solve(v_new, y_matrix)
    p_0 = 0
    p_n = 0
    for j in range(order+1):
      p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
      p_n += coef_matrix[len(coef_matrix) - (j+1)]* x_n**j
    approx_int.append(TrapRule1(x, x_0, x_n, p_0, p_n, delta))
    delta_change.append(delta)

plt.scatter(delta_change, approx_int1)
plt.ylabel("integral approximation")
plt.xlabel("delta")
plt.title("approximating the integral of y = x^2 between x=10 and x=40")
plt.show()
```



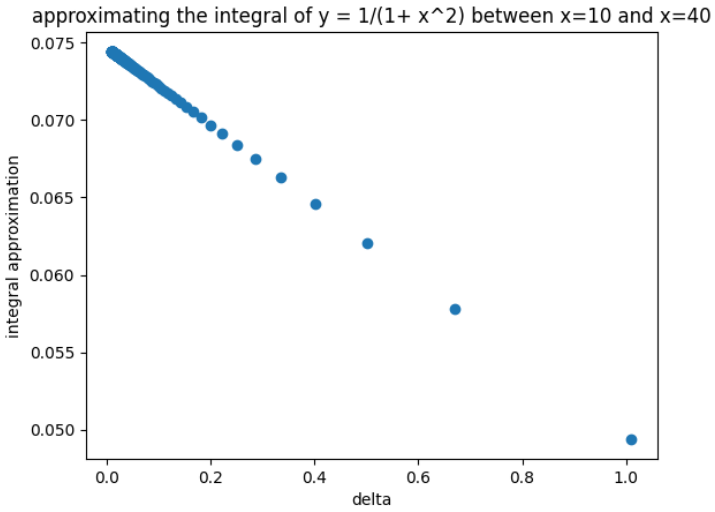

**notes.**

The integral is

$$\int_{10}^{40} x^2 dx = \frac{40^3 - 10^3}{3} = 21000$$

In the above graph, we see how the approximated value of the integral converges to 21000 as the width decreases (the convergence gets faster as I decrease the width), which is something expected.

Using the above code on the function $y = \frac{1}{1+x^2}$ (with order $= 18$), we get the below graph



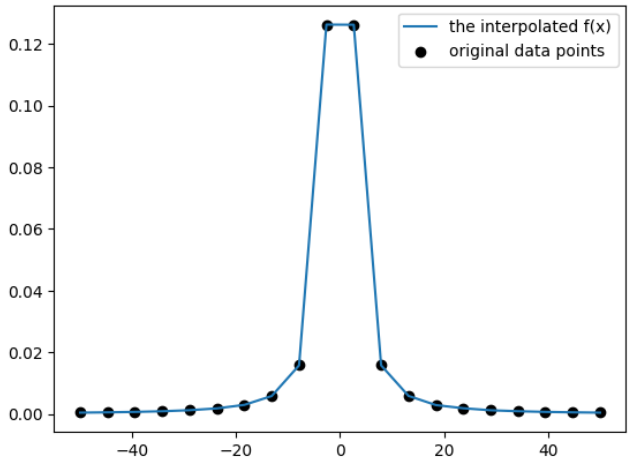

**notes.**

The integral is

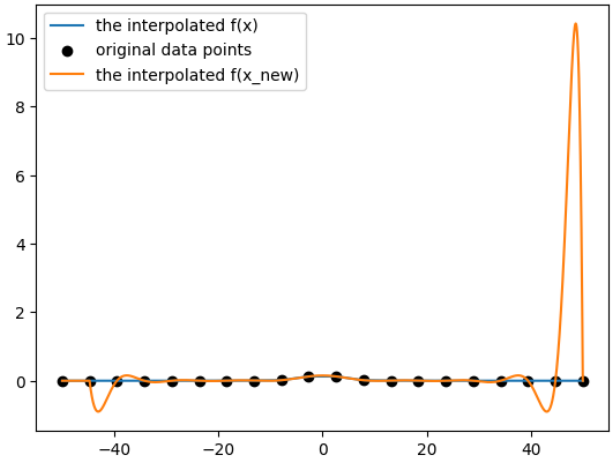
$$\int_{10}^{40} \frac{1}{1+x^2} dx = \arctan(40) - \arctan(10) = 0.07467385887$$

And we see in the above graph the convergence to this number as function of the decreasing width.

• **Second try:** in the first try, I used the generating function to increase the number of my data points. If I do not know the generating function, I cannot generate new data points (which is the case generally). So instead, I tried using the L2 optimization on the data points and globally fit them. Then, I fixed my interval using the coefficients. I then looped over every 2 data points and found the middle point between them then added it to the data set. I did this a couple of times, each time doubling the number of my points and decreasing the spacing between them by half.
I generated a small number of data points initially (to avoid noisy behavior) and I interpolated the original set using L2 optimization (which is very good, no noise):



Then I added to the graph the interpolation of the new set (original data points + the repeatedly found middle points) also using L2 optimization:



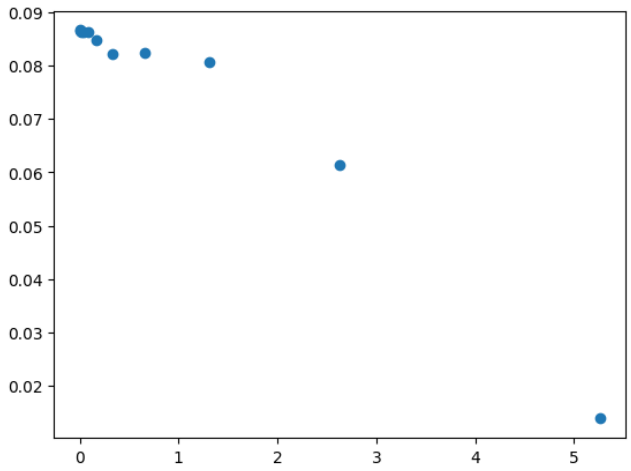

**notes.**

In the case of the interpolation of the new set, the testing set (the points added to the original set) is way larger than the training set (the original points), hence noise is expected. We see how the new set interpolation diverges sometimes from the original interpolation.
Now, I applied the trapezoidal rule on the new set each time I half the spacing, and plotted the outputted values of the integral as function of the spacing.

## code.

```
def TrapRule(x, y,x_0, x_n, p_0, p_n, delta):
    l = len(x)
    approx_int = (p_0 + p_n) * (delta/2)
    for i in range(l):
        if x[i] > x_0 and x[i] < x_n:
            approx_int += y[i] * delta
    return approx_int
x = np.linspace(-50,50,20)
y = 1/(1+x**2)
delta = x[1] - x[0]
order = 18
n = len(x)
approx_int = []
delta_change = []
v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)

p = 0
for i in range(order+1):
```

```
    p += coef_matrix[len(coef_matrix) - (i+1)]* x**i

x_0 = 10
x_n = 30
p_0 = 0
p_n= 0
for j in range(order+1):
     p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
     p_n += coef_matrix[len(coef_matrix) - (j+1)]* x_n**j
approx_int.append(TrapRule(x, y,x_0, x_n, p_0, p_n, delta))
delta_change.append(delta)
x_new = x

for o in range(10):
   y_new = 0
   for i in range(1,len(x_new)-1):
     x_t = (x_new[i] + x_new[i+1])/2
     x_new = np.append(x_new, x_t)
   x_new.sort()
   for j in range(order+1):
     y_new += coef_matrix[len(coef_matrix) - (j+1)]* x_new**j
   delta = delta/2
   approx_int.append(TrapRule(x_new, y_new,x_0, x_n, p_0, p_n, delta))
   delta_change.append(delta)

plt.scatter(delta_change, approx_int)
plt.show()
plt.plot(x, p, label = "the interpolated f(x)")
plt.scatter(x,y, label = "original data points", color = "k")
plt.plot(x_new, y_new, label = "the interpolated f(x_new)")
plt.legend()
plt.show()
```



**notes.**

The integral is

$$\int_{10}^{30} \frac{1}{1+x^2} dx = \arctan(30) - \arctan(10) = 0.06634765661$$

We see in the graph, the approximated value of the integral started very far from the analytical value then it started oscillating about it as I decreased the spacing, but still the final outputted values were larger than the analytical value. Noting that the outputted values seem to stop growing when the spacing got so small (0,00..) hence the final approximated value of the integral was $\approx 0.0865$.
The accuracy of the approximations of this method is affected by the noise/ error due to interpolation and the testing error.

## 2- Simpson's rule

Given an integral $\int_{x_0}^{x_2} f(x)dx$ such that $|x_2 - x_0| = 2h$, we can expand it about the middle point $(x_1)$:

$$\int_{x_0}^{x_2} f(x_1) + f'(x_1)(x - x_1) + f''(x)\frac{(x-x_1)^2}{2!} + f'''(x_1)\frac{(x-x_1)^3}{3!} + \cdots dx$$

$$= f(x_1)(x_2 - x_0) + \left[ f'(x)\frac{(x-x_1)^2}{2!} + f''(x_1)\frac{(x-x_1)^3}{3!} + \cdots \right]_{x_0}^{x_2}$$

only the odd powers survive, so we end up with

$$2hf(x_1) + \frac{2h^3}{3!}f''(x_1) + O(h^5) = 2hf(x_1) + \frac{h}{3}f(x_0) - \frac{2h}{3}f(x_1) + \frac{h}{3}f(x_2) + O(h^5)$$

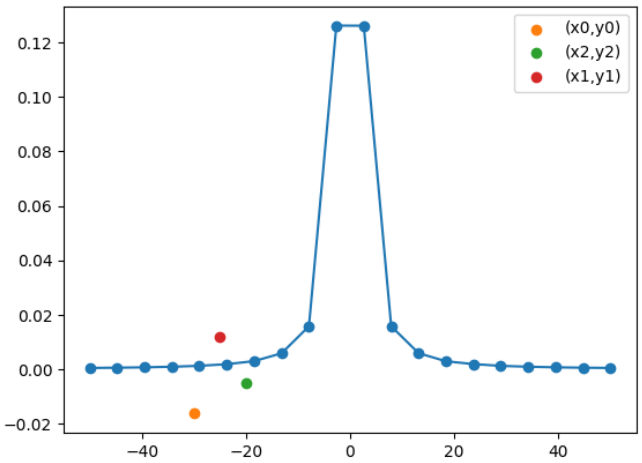
As $h$ gets smaller, the more accurate the result becomes.

• To check how the value of h affects the accuracy of the approximation, I wrote a code to approximate the integrals over some intervals. I used L2 optimization to fix my interval and find the middle point. Then, I checked 2 cases where h was small in one and large in the other:

**Small h:** for the function $y = \frac{1}{1+x^2}$, I want to approximate

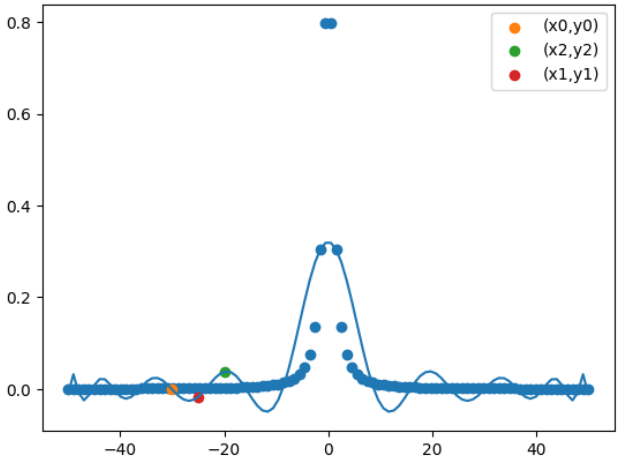
$$\int_{-30}^{-20} \frac{1}{1+x^2} dx = 0.01663739984$$

I used L2 optimization to fix this interval $(-20, -30)$ and fixed the middle point $-25$ such that $h = 5$. I plotted the interpolated function and these 3 points




**notes.**

I used the coefficients from L2 optimization to find these 3 points, and it is obvious that they lie in the noisy area of the interpolation. I used a small number of data points (20 points), so the interpolation does not appear noisy but as I increase the number of initially generated data points, the interpolation gets noisy (something we witnessed while coding the L2 optimization method). For 100 points, I got this graph:




**notes**

In this case, the output value for the integral I am getting is negative so this is bad. I will stick to fewer number of data points.

**code.**

```
def SimpRule(p_0, p_1, p_2, h):
   approx_int = 0
   approx_int += ((4*h)/3)*p_1
```

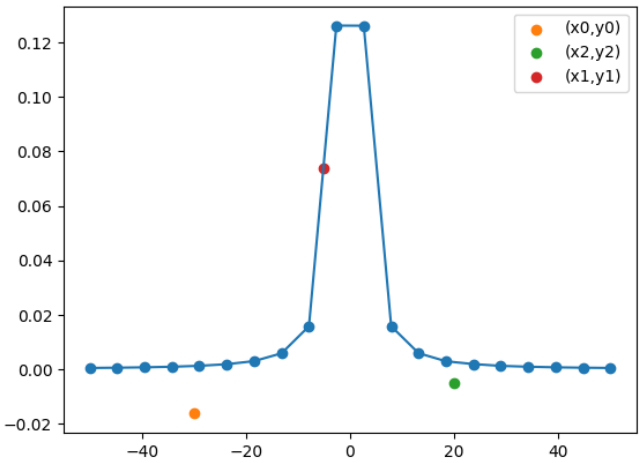
```
   approx_int += (h/3) * p_0
   approx_int += (h/3)*p_2
   return approx_int

x = np.linspace(-50,50,20)
y = 1/(1+x**2)
delta = x[1] - x[0]
order = 18
n = len(x)

y = 1/(1+x**2)
v = (np.reshape(x, (n, 1)) * np.ones(order + 1)) ** [i for i in range(order, -1, -1)]
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)

p=0
x_0 = -30
x_2 = -20
x_1= (x_0 + x_2)/2
h = abs(x_2- x_1)
p_0 = 0
p_2= 0
p_1 = 0
for j in range(order+1):
     p_0 += coef_matrix[len(coef_matrix) - (j+1)]* x_0**j
     p_2 += coef_matrix[len(coef_matrix) - (j+1)]* x_2**j
     p_1 += coef_matrix[len(coef_matrix) - (j+1)]* x_1**j
     p += coef_matrix[len(coef_matrix) - (j+1)]* x**j


approx_int = SimpRule(p_0, p_1, p_2, h)
plt.plot(x,p)
plt.scatter(x,y)
plt.scatter(x_0,p_0, label = "(x0,y0)")
plt.scatter(x_2,p_2, label = "(x2,y2)")
plt.scatter(x_1,p_1, label = "(x1,y1)")
plt.legend()
plt.show()
```

Which outputted this value for the approximated integral: **0.04206470822323488.** So it is obvious that the noise affected the accuracy of the method.

**Big h:** here, I wanted to see what happens if the $h$ was big, so I tried to approximate

$$\int_{-30}^{20} \frac{1}{1+x^2} dx = 3.058313262$$

I used L2 optimization to fix this interval $(-30, 20)$ and fixed the middle point $-5$ such that $h = 25$:




Using the above code, I got the value **2.2775349317548614**.

**Comapring errors:** the error/deviation due to the noise with the small $h$:

$$0.0420647082232348 - 0.01663739984 = 0.02542730838$$

and the error due to the noise with the big $h$:

$$3.058313262 - 2.2775349317548614 = 0.7807783302$$

Concluding that the choice of $h$ does affect the approximation with the smaller the $h$ the less the deviation from the analytical value.

# 8 Differential Equations

## 1- Euler method for 1st order ODEs

$\frac{dy}{dt} = y' = f(y,t) = \frac{y(t+h)-y(t)}{h}$ for $t = 0, h, 2h, 3h...$ We can "predict" the next step relying on the previous step using First order Taylor series approximation:

$$y(t+h) = y(t) + hf(y(t),t) + \mathcal{O}(h^2)$$

starting from the initial condition $(t_0, y_0) = (0, y(0))$ such that $y(h) = y(0) + hf(y(0))$. For second-order equations ODEs of the form $y'' = g(t,y)$, we can reduce the problem to 2 first order ODEs:

$$\begin{cases} y' = v = f(y,t) \\ v' = g(y,t) \end{cases}$$

with solutions

$$\begin{cases} y(t+h) = y(t) + hf(y(t),t) \\ v(t+h) = v(t) + hg(y(t),t) \end{cases}$$

Using this method, we can calculate the deviation of the solution from the analytical value from

$$\frac{1}{N}\sum_{n=0}^{n=N}(y_{analytical} - y_{numerical})^2$$

**Siff equations**

A stiff equation is a differential equation for which certain numerical methods, like Euler method, are numerically unstable, unless the step size is taken to be extremely small. Stiff differential equations are characterized as those whose exact solution has a term of the form $e^{-ct}$, where c is a large positive constant. Euler's method is expected to be stable for the test equation only if the step size $h < \frac{2}{c}$.

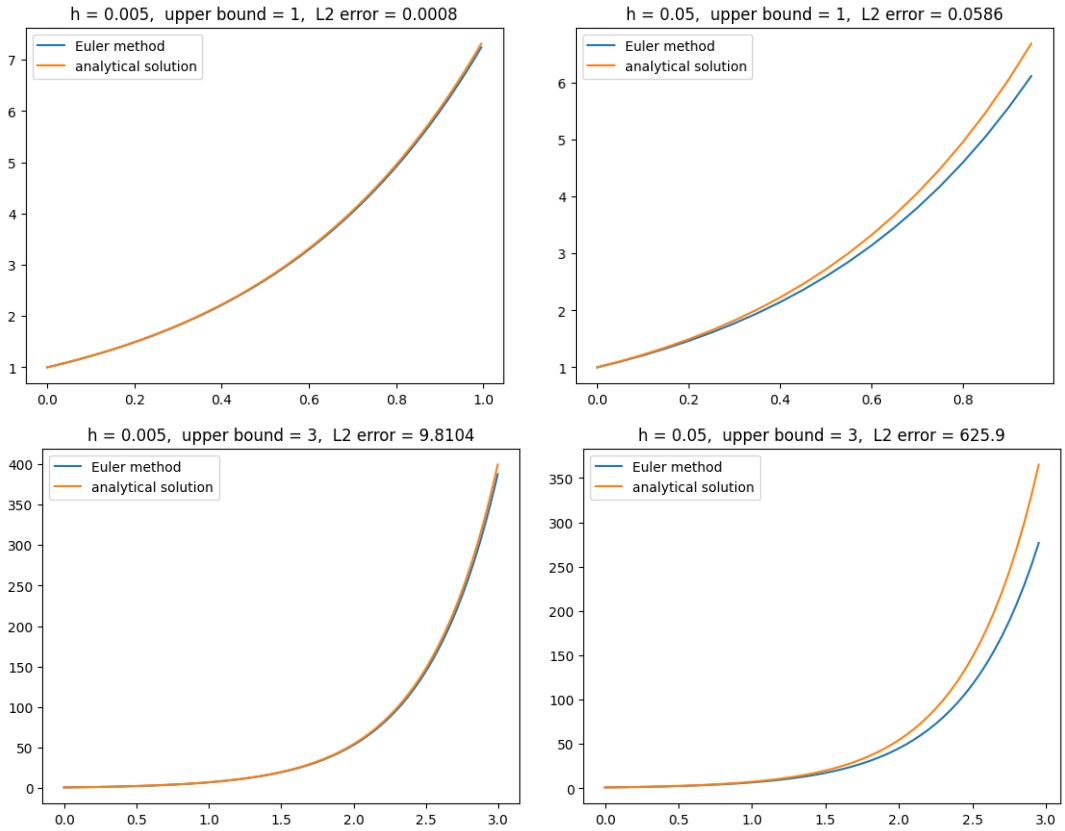
## Try 1:

I will apply the Euler method on the function $y' = 2y$ with the initial condition $y(0) = 1$.

**code.**

I will be using this code while modifying the values of $h$ and the upper bound to formulate a judgment:

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
y = np.zeros(len(t))
y_0 = 1
y[0] = y_0
for i in range(0, int((upper_bound)/h)-1):
    y [i + 1] = y[i] + 2*h*y[i]
y_a = np.exp(2*t)
plt.plot(t,y, label = "Euler method")
plt.plot(t,y_analytical, label = "analytical solution")
plt.legend()
plt.show()
l2_error = np.sum((y_analytical - y)**2)/len(y)
```




**notes.**

Euler method works best on small time intervals with small separations ($h$). As these 2 parameters increase, the accuracy of Euler method becomes more and more questionable. To use Euler method accurately on a large interval, hence, is a matter of dividing these intervals into smaller ones and applying Euler method on each small interval.

## Try 2

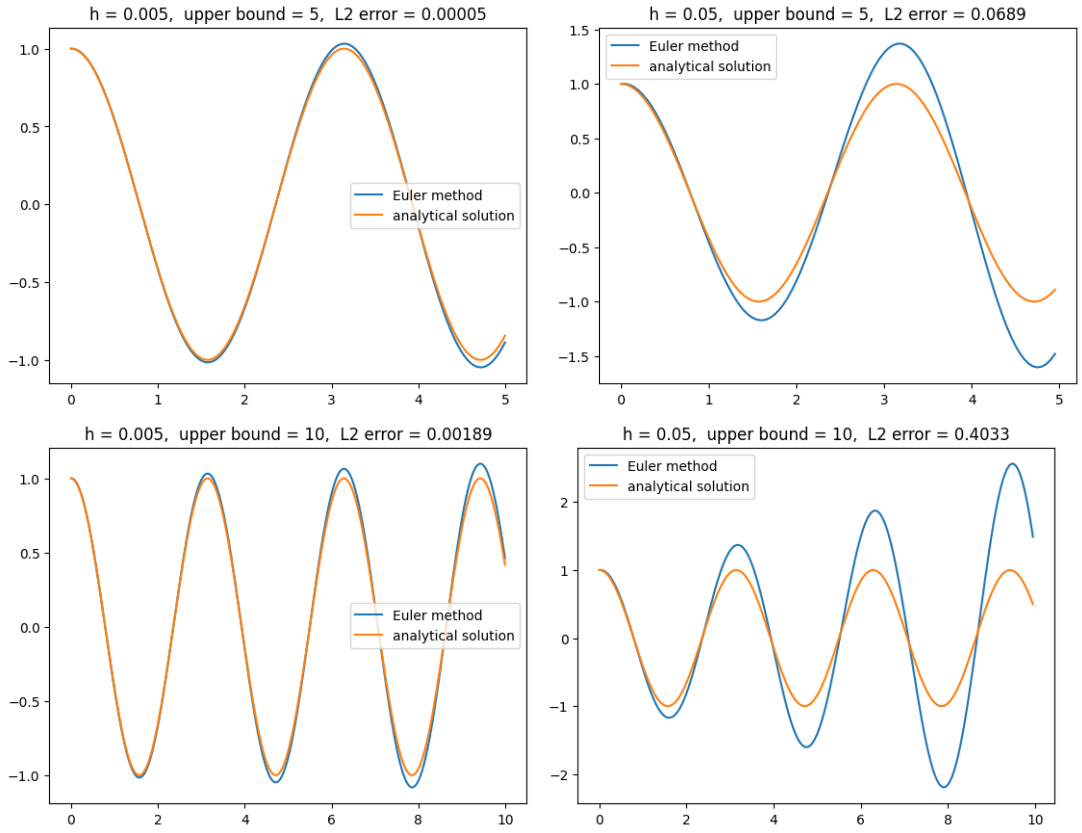
I will apply the Euler method on the equation of a damped harmonic oscillator $x'' + w^2 x = 0$ with the initial conditions $(x(0), x'(0)) = (1, 0)$ and frequency $w = 2$.

**code.**

I will be using this code while modifying the values of $h$ and the upper bound to formulate a judgment:

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x = np.zeros(len(t))
v = np.zeros(len(t))
w_2 = 4
```

```
x_0 = 1
v_0 = 0
x[0] = x_0
v[0] = v_0
for i in range(0, int((upper_bound)/h)-1):
     x [i + 1] = x[i] + h*v[i]
     v [i + 1] = v[i] - h*w_2*x[i]
x_a = np.cos(2*t)
plt.plot(t,x, label = "Euler method")
plt.plot(t,x_a, label = "analytical solution")
plt.legend()
plt.show()
l2_error = np.sum((x_a - x)**2)/len(x)
```




**notes.**

The same observations regarding the spacing and the interval. The solution here is a cosine function so this is not a stiff equation, we notice how the errors in this case are way less than the errors in the first try where the equation had an exponential solution.

## 2- RK2 method for nth order ODEs

If we have a second order differential equation $y'' = g(y)$, we can reduce to a system of 2 1st order differential equations:

$$\begin{cases} y' = v = f(y,t) \\ v' = g(y,t) \end{cases}$$

$$y'(t) = f(t,y)$$

$$y''(t) = \frac{df(y,t)}{dt} = f_t(y,t) + f_y(y,t)y'(t)$$

We obtain the form of the solution by expanding $y(t+h)$ about $t$ up to the second order:

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2}y''(t) + \mathcal{O}(h^3)$$

$$= y(t) + hf(y,t) + \frac{h^2}{2}\Big(f_t(y,t) + f_y(y,t)f(y,t)\Big) + \mathcal{O}(h^3)$$

$\frac{h^2}{2}\Big(f_t(y,t) + f_y(y,t)f(y,t)\Big) = h\Big(\frac{h}{2}f_t(y,t) + \frac{hf(y,t)}{2}f_y(y,t)\Big)$ defining $k = hf(y,t)$ we get

$$h\Big(\frac{h}{2}f_t(y,t) + \frac{k}{2}f_y(y,t)\Big)$$

which is 2 dimensional expansion of $f(y+\frac{k}{2}, t+\frac{h}{2})$ around $(y,t)$. Replacing in the solution we get

$$y(t+h) = y(t) + hf\Big(y + \frac{k}{2}, t + \frac{h}{2}\Big) + \mathcal{O}(h^3)$$

We can now construct the solutions to the system of the 2 1st order ODEs we got:

$$\begin{cases} y(t+h) = y(t) + hf\big(y + \frac{k_1}{2}, t + \frac{h}{2}\big) + \mathcal{O}(h^3) \\ v(t+h) = v(t) + hg\big(y + \frac{k_2}{2}, t + \frac{h}{2}\big) + \mathcal{O}(h^3) \end{cases}$$

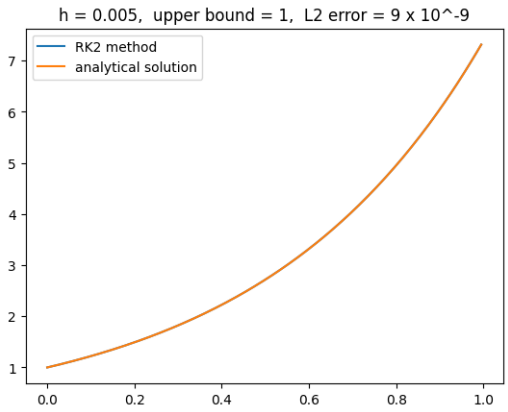
where $k_1 = hf(y,t)$ and $k_2 = hf(y,t)$
RK2 method is more accurate than the Euler method since to predict the next step, it accounts for the neighboring point to avoid overshooting.
I will use RK2 method on the functions I used Euler method on and compare the errors between the 2 methods.

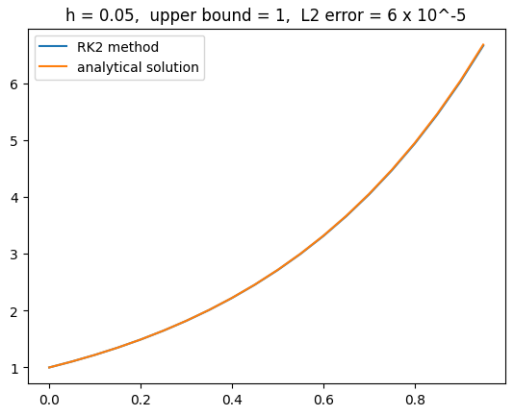
**Using RK2 on $y' = 2y$:**

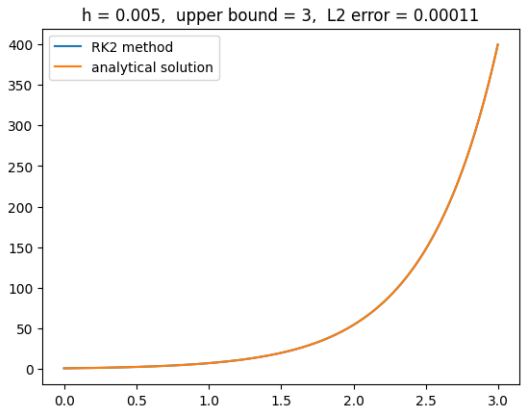
I will use this code while varying the values of h and the upper bound and find the l2 error:

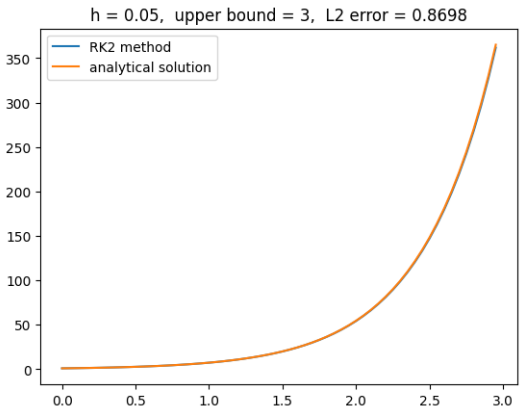
```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x_0 = 1
x = np.zeros(len(t))
x[0] = x_0
for i in range(0, int(upper_bound/h)-1):
    k = h*2*x[i]
```

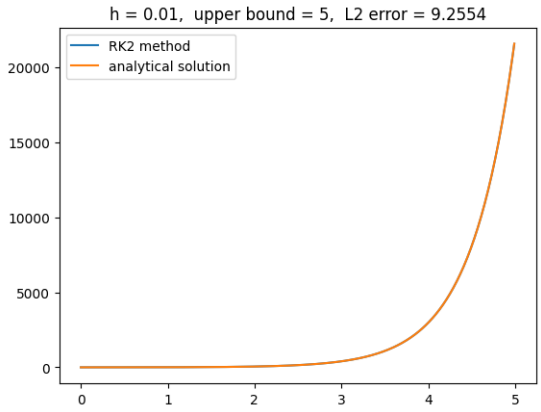
```
    x[i+1] = x[i] + h*2*(x[i] + k/2)
x_a = np.exp(2*t)
plt.plot(t,x, label = "RK2 method")
plt.plot(t,x_a, label = "analytical solution")
plt.legend()
plt.show()
l2_error = np.sum((x_a - x)**2)/len(x)
```

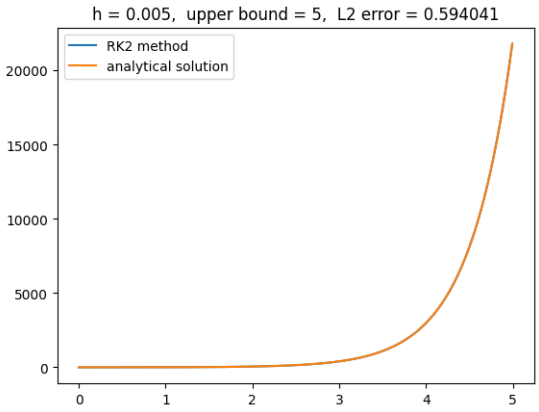


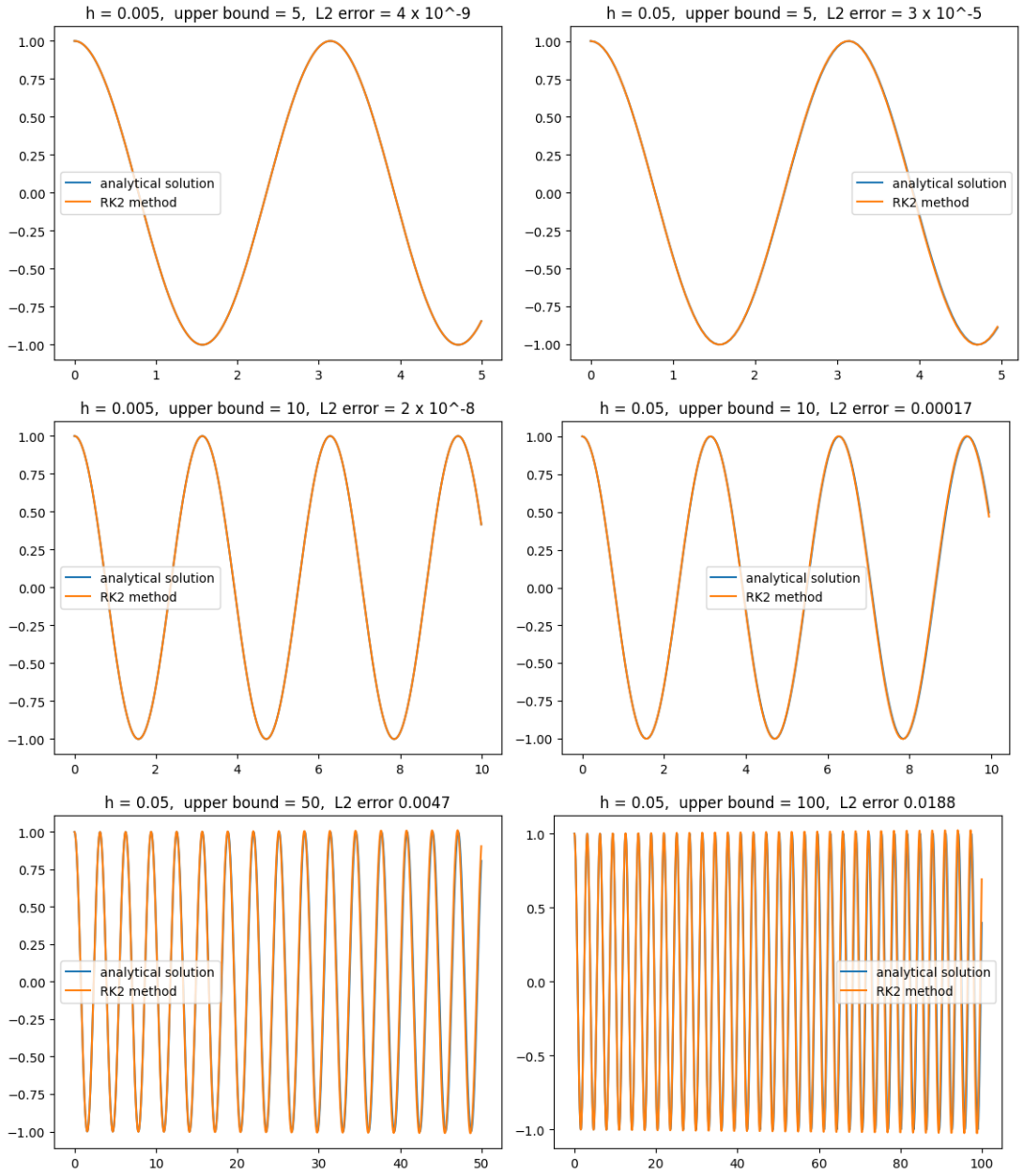
**notes.**

Comparing these results with the ones I got using the Euler method on the equation, we see a huge difference in the errors. RK2 solutions do not diverge drastically for relatively big values of t and they seem to overlap with the analytical solution.

**Using RK2 on the damped harmonic oscillator:**

I will use RK2 on the equation $x'' + w^2x = 0$ with the initial conditions $(x(0), x'(0)) = (1, 0)$ and frequency $w = 2$, then I will compare the resulting errors with the Euler method. I will be using this code while varying the values of h and upper bound:

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x = np.zeros(len(t))
v = np.zeros(len(t))
x_0 = 1
v_0 = 0
w_2 = 4
x[0] = x_0
v[0] = v_0
for i in range(0, int(upper_bound/h)-1):
    k1 = h * v[i]
    k2 = -h * w_2 * x[i]
    x[i+1] = x[i] + h * (v[i] + k2/2)
    v[i+1] = v[i] -h * w_2 * (x[i] + k1/2)SS
x_a = np.cos(2*t)
plt.plot(t, x_a, label="analytical solution")
plt.plot(t, x, label="RK2 method")
plt.legend()
plt.show()
l2_error = np.sum((x_a - x)**2)/len(x)
```




**notes.**

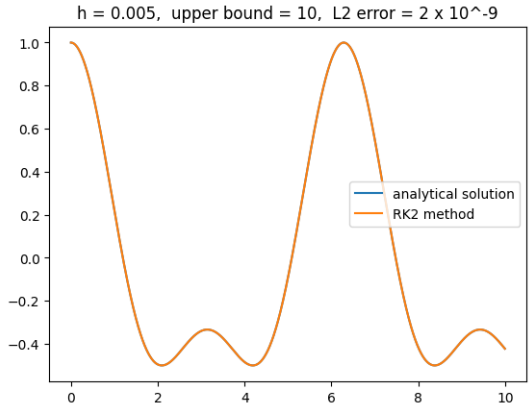
Comparing to the solutions I got using the Euler method, these solutions are better approximations of the analytical solution with very small least square errors.

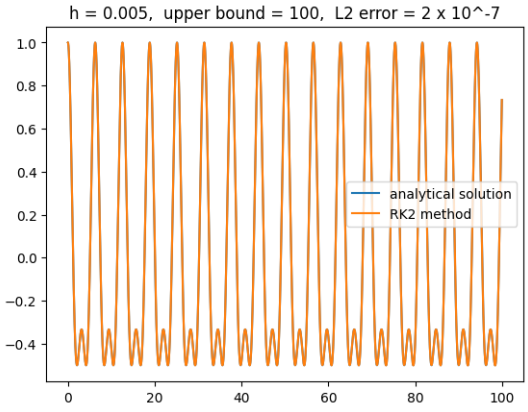
### Using RK2 on the damped forced harmonic oscillator

I will use RK2 to solve equations of the form $x'' + w_d^2 x = F\cos(w_f t)$. Here, we have explicit time dependency in the $g(y,t)$ term. For the initial conditions $(x(0), x'(0)) = (1, 0)$ and the values

$w_d = 2, w_f = 1, F = 1$: the anlaytical solution is $x = \frac{1}{3}\cos(2t) + \frac{2}{3}\cos(t)$. I will be using this code while varying the values of h and the upper bound:

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x = np.zeros(len(t))
v = np.zeros(len(t))
x_0 = 1
v_0 = 0
x[0] = x_0
v[0] = v_0
wd = 2
wf = 1
F = 2
for i in range(0, int(upper_bound/h)-1):
    k1 = h*v[i]
    k2 = h*F*np.cos(wf*t[i]) - h*wd**2*x[i]
    x[i+1] = x[i] + h * (v[i] + k2/2)
    v[i+1] = v[i] + h*F*np.cos(wf*(t[i]+h/2)) - h*wd**2 * (x[i] + k1/2)
x_a = (1/3) * np.cos(2*t) + (2/3) * np.cos(t)
plt.plot(t, x_a, label="analytical solution")
plt.plot(t, x, label="RK2 method")
plt.legend()
plt.show()
l2_error = np.sum((x_a - x)**2)/len(x)
```




**notes.**

The results are impressive.

## 3- RK4 method for nth order ODEs

We expand $y(t+h)$ this time up to the 4th order and we get

$$y(t+h) = y(t) + h\left(f(y,t) + \frac{h}{2}f'(y,t) + \frac{h^2}{3!}f''(y,t) + \frac{h^3}{4!}f'''(y,t)\right) + \mathcal{O}(h^5)$$

$$= y(t) + \left(a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4\right) + \mathcal{O}(h^5)$$

where $\{a\}$ are the weights, plugging their numerical values, we get hence, the system of 1st order ODEs becomes

$$\begin{cases} y(t+h) = y(t) + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) + \mathcal{O}(h^5) \\ v(t+h) = v(t) + \frac{1}{6}\left(k_1' + 2k_2' + 2k_3' + k_4'\right) + \mathcal{O}(h^5) \end{cases}$$

where

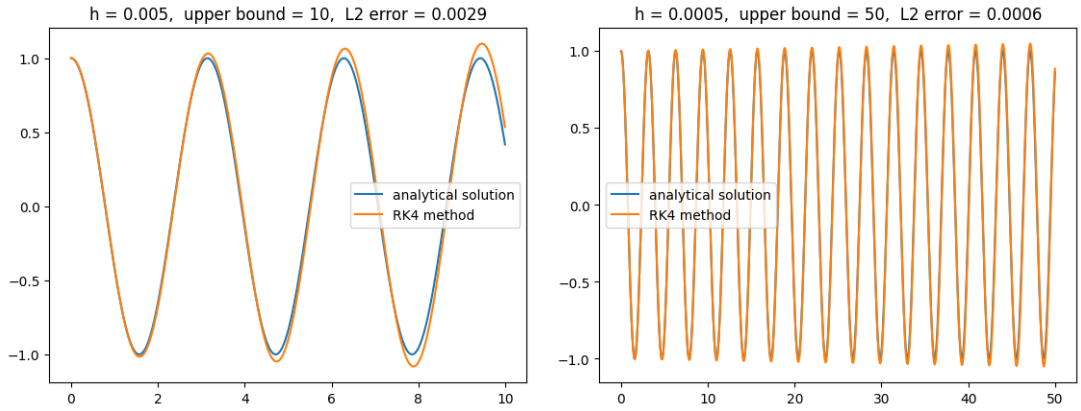
$$k_1 = hf\left(t, y\right)$$
$$k_2 = hf\left(t + \frac{h}{2}, y + \frac{k_1}{2}\right)$$
$$k_3 = hf\left(t + \frac{h}{2}, y + \frac{k_2}{2}\right)$$
$$k_4 = hf\left(t + h, y + k_3\right)$$
$$k_1' = hg\left(t, y\right)$$
$$k_2' = hg\left(t + \frac{h}{2}, y + \frac{k_1'}{2}\right)$$
$$k_3' = hg\left(t + \frac{h}{2}, y + \frac{k_2'}{2}\right)$$
$$k_4' = hg\left(t + h, y + k_3'\right)$$

RK4 method scans the environment before predicting the next step.

**Using RK4 on the damped harmonic oscillator**

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x = np.zeros(len(t))
v = np.zeros(len(t))
x_0 = 1
v_0 = 0
w = 2
x[0] = x_0
v[0] = v_0
for i in range(0, int(upper_bound/h)-1):
    k1 = h*v[i]
    k2 = h*(v[i] + k1/2)
    k3 = h*(v[i] + k2/2)
    k4 = h*(v[i] + k3)
    k1_ = -h* w**2 * x[i]
    k2_ = -h* w**2 * (x[i] + k1_/2)
    k3_ = -h* w**2 * (x[i] + k2_/2)
    k4_ = -h* w**2 * (x[i] + k3_)
    x[i+1] = x[i] + (1/6) * (k1 + 2*k2 + 2*k3 + k4)
    v[i+1] = v[i] + (1/6) * (k1_ + 2*k2_ + 2*k3_ + k4_)
x_a = np.cos(2*t)
plt.plot(t, x_a, label="analytical solution")
```

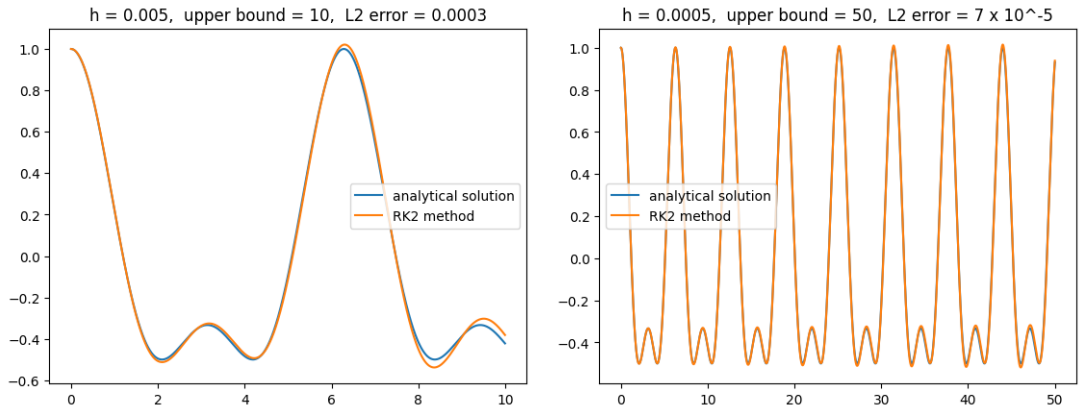
```
plt.plot(t, x, label="RK4 method")
plt.legend()
plt.show()
l2_error = np.sum((x_a - x)**2)/len(x)
```




**Using RK4 on the forced damped harmonic oscillator**

```
h =
upper_bound =
t = np.arange(0, upper_bound, h)
x = np.zeros(len(t))
v = np.zeros(len(t))
x_0 = 1
v_0 = 0
x[0] = x_0
v[0] = v_0
wd = 2
wf = 1
F = 2
for i in range(0, int(upper_bound/h)-1):
    k1 = h*v[i]
    k2 = h*(v[i] + k1/2)
    k3 = h*(v[i] + k2/2)
    k4 = h*(v[i] + k3)
    k1_ =  h*F*np.cos(wf*(t[i]+h/2)) -h* w**2 * x[i]
    k2_ =  h*F*np.cos(wf*(t[i]+h/2)) -h* w**2 * (x[i] + k1_/2)
    k3_ =  h*F*np.cos(wf*(t[i]+h/2)) -h* w**2 * (x[i] + k2_/2)
    k4_ =  h*F*np.cos(wf*(t[i]+h/2)) -h* w**2 * (x[i] + k3_)
    x[i+1] = x[i] + (1/6) * (k1 + 2*k2 + 2*k3 + k4)
    v[i+1] = v[i] + (1/6) * (k1_ + 2*k2_ + 2*k3_ + k4_)
x_a = (1/3) * np.cos(2*t) + (2/3) * np.cos(t)
plt.plot(t, x_a, label="analytical solution")
plt.plot(t, x, label="RK2 method")
plt.legend()
plt.show()
```

```
l2_error = np.sum((x_a - x)**2)/len(x)
```




**notes.**

RK4 seems to give more accurate results when the spacing $h$ is very small, unlike RK2 that worked fine for relatively smaller $h$.

## 3- Perturbative approach

Given an equation of the form $\dot{x} = f(x,t)$, we can solve it by expanding the solution (to be found) in terms of the variable $t$ about $t_0$:

$$x = x[0] + x[1](t-t_0) + x[2](t-t_0)^2 + ... + x[k](t-t_0)^k$$

where $x[0] = x(t_0)$. We also expand $f(x,t)$ about $t_0$:

$$f(x,t) = f[0] + f[1](t-t_0) + ... + f[k](t-t_0)^k$$

where $f[k] = \frac{1}{k!}\frac{\partial^k}{\partial t^k} f(t_0)$. Now, differentiating the expansion of x, we get

$$\dot{x} = x[1] + 2x[2](t-t_0) + 3x[3](t-t_0)^2 + ... + kx[k](t-t_0)^{k-1}$$

comparing with the expansion of $f(x,t)$, we get $x[1] = f[0] = f(t_0)$, $x[2] = \frac{1}{2}f[1] = \frac{1}{2}\frac{\partial}{\partial t}f(t_0)$... Noticing the pattern we conclude

$$x[k+1] = \frac{1}{k+1}f[k] = \frac{1}{(k+1)!}\frac{\partial^k}{\partial t^k}f(t_0)$$

**note.**

The number of terms in these expansions depends on the order of the polynomial $f(x,t)$.
**Example.** for the equation $\dot{x} = x^2$, we try the expansion about 0 $x(t) = x_0 + x[1]t + \mathcal{O}(t^2)$, squaring we get

$$x^2(t) = x_0^2 + x[1]^2t^2 + 2x_0x[1]t + \mathcal{O}(h^4)$$

and we know

$$f(x,t) = f[0] + f[1]t + f[2]t^2$$

comparing, we get these coefficients:

$$x_0 = x(t_0)$$
$$2x_0x[1] = f[0] = x_0^2$$
$$x[2] = \frac{1}{2}f[1] = \frac{1}{2}2x_0x[1] = x_0^3$$
$$x[3] = \frac{1}{3}f[2] = \frac{1}{3}(x[1]^2 + 2x_0x[2]) = x_0^4$$

Hence $x(t) = x_0 + x_0^2t = x_0^3t^2 + ... = \frac{x_0}{1-x_0t}$.

To implement this numerically, a stopping condition is set to be $h < \left(\frac{\epsilon}{x_k}\right)^{\frac{1}{k}}$ where $\epsilon$ is such that $x[k]h^k < \epsilon$. This condition is such that we get convergence.