Scientific Python Intro and Theory

This series of notebooks is based on Python along with the following scientific libraries/packages:

- NumPy (for working with arrays)
- Matplotlib (for plotting and visualizing data)
- SymPy (for symbolic calculations)
- SciPy (for scientific tools such as special functions, root finding, numerical integration, and numerical solutions to ODE's)

twd- i forget where this snippet comes from... (numpy.linalg) comment from mysterious source...

The NumPy linear algebra functions rely on BLAS and LAPACK to provide efficient low level implementations of standard linear algebra algorithms. Those libraries may be provided by NumPy itself using C versions of a subset of their reference implementations but, when possible, highly optimized libraries that take advantage of specialized processor functionality are preferred. Examples of such libraries are OpenBLAS, MKL (TM), and ATLAS. Because those libraries are multithreaded and processor dependent, environmental variables and external packages such as threadpoolctl may be needed to control the number of threads or specify the processor architecture.

The SciPy library also contains a linalg submodule, and there is overlap in the functionality provided by the SciPy and NumPy submodules. SciPy contains functions not found in numpy.linalg, such as functions related to LU decomposition and the Schur decomposition, multiple ways of calculating the pseudoinverse, and matrix transcendentals such as the matrix logarithm. Some functions that exist in both have augmented functionality in scipy.linalg. For example, scipy.linalg.eig can take a second matrix argument for solving generalized eigenvalue problems. Some functions in NumPy, however, have more flexible broadcasting options. For example, numpy.linalg.solve can handle “stacked” arrays, while scipy.linalg.solve accepts only a single square array as its first argument.

NOTE: The term matrix as it is used on this page indicates a 2d numpy.array object, and not a numpy.matrix object. The latter is no longer recommended, even for linear algebra. See the matrix object documentation for more information.

1. Python Lists Primer

Lists are ubiquitous data structures in almost every langauge and programming platform. Here is universal python list information. The following sections the additional libraries (Numpy, Sympy, etc) will also address lists in their own specific ways.

Creating Lists

There are multiple ways of creating lists.

In [31]:
```
[1, 2, 3, 4]
```

Out[31]:
```
[1, 2, 3, 4]
```

In [32]:
```
list((2, 4, 6, 8))
```

Out[32]:
```
[2, 4, 6, 8]
```

Lists with repeated elements can be created by mutliplying a singlet-list containing the element to be repeated.

In [33]:
```
[1]*2
```

Out[33]:
```
[1, 1]
```

In [34]:
```
[0]*20
```

Out[34]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Lists may contain strings as well as numbers.

In [35]:
```
animals = ["cat", "dog", "squirrel", "rabbit"]
animals
```

Out[35]:
```
['cat', 'dog', 'squirrel', 'rabbit']
```

Objects of different data types may occur in the same list.

In [36]:
```
mixed_list = [1, 2, 3.0, 4.0, "cat", "dog", True, False, ['a', 'b']]
mixed_list
```

Out[36]:
```
[1, 2, 3.0, 4.0, 'cat', 'dog', True, False, ['a', 'b']]
```

List comprehension

Lists can be created quickly using list "comprehensions".

In [37]:
```
[x for x in range(10)]
```

Out[37]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Suppose we want to flatten a list of lists. We could do the following.

In [38]:
```
verts = [[1,2], [3,4]]
coords = []                  #starts out blank
for sublist in verts:        #for each sublist in verts
    for item in sublist:     #for each item in sublist
        coords.append(item)  #add item to coords
coords
```

Out[38]:
```
[1, 2, 3, 4]
```

Equivalently, we could use a list comprehension as follows.

In [39]:
```
coords = [xi for sublist in verts for xi in sublist]
coords
```

Out[39]:
```
[1, 2, 3, 4]
```

Accessing list elements

In [40]:
```
mylist = ["A", "B", "C", "D", "E", "F", "G", "H"]
print(mylist[0])
print(mylist[1])
print(mylist[2])
print(mylist[3])
print(mylist[4])
print(mylist[5])
print(mylist[6])
print(mylist[7])
```

```
A
B
C
D
E
F
G
H
```

In [41]:
```
print(mylist[-1])
print(mylist[-2])
print(mylist[-3])
print(mylist[-4])
```

```
H
G
F
E
```

In [42]:
```
print(mylist[:])
print(mylist[:3])
print(mylist[3:])
print(mylist[1:3])
print(mylist[::2])
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
['A', 'B', 'C']
['D', 'E', 'F', 'G', 'H']
['B', 'C']
['A', 'C', 'E', 'G']
```

In [43]:
```
for x in mylist:  #prints all the elements in the list
    print(x)
```

```
A
B
C
D
E
F
G
H
```

3. Inserting and removing elements

In [44]:
```
mylist = ["cat", "dog", "bird", "snake"]
mylist.append("fish")
mylist
```

Out[44]:
```
['cat', 'dog', 'bird', 'snake', 'fish']
```

In [45]:
```
mylist.insert(2, "pig")
mylist
```

Out[45]:
```
['cat', 'dog', 'pig', 'bird', 'snake', 'fish']
```

In [46]:
```
mylist.remove("fish") #you can remove by name rather than index location
mylist
```

Out[46]:
```
['cat', 'dog', 'pig', 'bird', 'snake']
```

In [47]:
```
del mylist[1]
mylist
```

Out[47]:
```
['cat', 'pig', 'bird', 'snake']
```

In [48]:
```
mylist.pop(2) #removes the element at index 2
mylist
```

Out[48]:
```
['cat', 'pig', 'snake']
```

In [49]:
```
mylist.pop() #removes the last element
mylist
```

Out[49]:
```
['cat', 'pig']
```

In [50]:
```
mylist.clear() #removes all elements
mylist
```

Out[50]:
```
[]
```

Note: Copying a list

You cannot copy a list simply by assignment (list1 = list2). This will mererely reference list2 to list1. Changes made in list1 will automatically also be made in list2

In [51]:
```
list1 = [1, 2, 3, 4]
list2 = list1
list2.pop() #removes the last element
print(list1)
print(list2) #list2 has also changed
```

```
[1, 2, 3]
[1, 2, 3]
```

To make a copy, you can use the copy() method.

In [52]:
```
list1 = [1, 2, 3, 4]
list2 = list1.copy()
list2.pop()
print(list1)
print(list2)
```

```
[1, 2, 3, 4]
[1, 2, 3]
```

Alternatively, use the list() method on the list you want to copy.

In [53]:
```
list1 = [1, 2, 3, 4]
list2 = list(list1)
list2.pop()
print(list1)
print(list2)
```

```
[1, 2, 3, 4]
[1, 2, 3]
```

5. Other list methods

In [54]:
```
list3 = ["cat", "dog", "bird", "cat", "cat"]
list3.count("cat") #counts the number of times "cat" appears in the list
```

Out[54]:
```
3
```

In [55]:
```
list3.index("bird")
```

Out[55]:
```
2
```

In [56]:
```
list3.reverse() #reveresed order applied to list 3
list3
```

Out[56]:
```
['cat', 'cat', 'bird', 'dog', 'cat']
```

In [57]:
```
list3.sort() #sorts the list alphabetically
list3
```

Out[57]:
```
['bird', 'cat', 'cat', 'cat', 'dog']
```

2. Data handling DataFrames Pandas

Importing Data with Pandas

In [58]:
```
import pandas as pd
```

In [59]:
```
#Importing Data
pd.read_csv(filename)     # From a CSV file
pd.read_table(filename)   # From a delimited text file (like TSV)
```

```
pd.read_excel(filename)   # From an Excel file
pd.read_sql(query, connection_object) # Reads from a SQL table/database
pd.read_json(json_string) # Reads from a JSON formatted string, URL or file.
pd.read_html(url)
# Parses an html URL, string or file and extracts tables to a list of dataframes
pd.read_clipboard()
# Takes the contents of your clipboard and passes it to read_table()
pd.DataFrame(dict)
# From a dict, keys for columns names, values for data as lists
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-59-cfbb6692bbab> in <module>
      1 #Importing Data
----> 2 pd.read_csv(filename)     # From a CSV file
      3 pd.read_table(filename)   # From a delimited text file (like TSV)
      4 pd.read_excel(filename)   # From an Excel file
      5 pd.read_sql(query, connection_object) # Reads from a SQL table/database

NameError: name 'filename' is not defined
```

Exploring Data Frames DF

In [ ]:
```
#Exploring Data
df.shape()       # Prints number of rows and columns in dataframe
df.head(n)       # Prints first n rows of the DataFrame
df.tail(n)       # Prints last n rows of the DataFrame
df.info()        # Index, Datatype and Memory information
df.describe()    # Summary statistics for numerical columns
s.value_counts(dropna=False) # Views unique values and counts
df.apply(pd.Series.value_counts) # Unique values and counts for all columns
df.describe()    # Summary statistics for numerical columns
df.mean()        # Returns the mean of all columns
df.corr()        # Returns the correlation between columns in a DataFrame
df.count()     # Returns the number of non-null values in each DataFrame column
df.max()         # Returns the highest value in each column
df.min()         # Returns the lowest value in each column
df.median()      # Returns the median of each column
df.std()         # Returns the standard deviation of each column
```

Selecting Data

In [ ]:
```
#selecting Data
df[col]            # Returns column with label col as Series
df[[col1, col2]]   # Returns Columns as a new DataFrame
s.iloc[0]          # Selection by position (selects first element)
s.loc[0]           # Selection by index (selects element at index 0)
df.iloc[0,:]       # First row
df.iloc[0,0]       # First element of first column
```

Data Cleaning

In [ ]:
```
#Data Cleaning
df.columns = ['a','b','c'] # Renames columns
pd.isnull()                # Checks for null Values, Returns Boolean Array
pd.notnull()               # Opposite of s.isnull()
df.dropna()                # Drops all rows that contain null values
df.dropna(axis=1)          # Drops all columns that contain null values
df.dropna(axis=1,thresh=n) # Drops all rows have have less than n non nullvalue
df.fillna(x)               # Replaces all null values with x
s.fillna(s.mean()) # Replaces all null values with the mean (mean can be
# replaced with almost any function from the statistics section)
s.astype(float)            # Converts the datatype of the series to float
s.replace(1,'one')         # Replaces all values equal to 1 with 'one'
s.replace([1,3],['one','three'])  # Replaces all 1 with 'one' and 3 with 'three'
df.rename(columns=lambda x: x + 1) # Mass renaming of columns
df.rename(columns={'old_name': 'new_ name'}) # Selective renaming
df.set_index('column_one')         # Changes the index
df.rename(index=lambda x: x + 1)   # Mass renaming of index
```

Filter, Sort and Group By

In [ ]:
```
#Filter, Sort and Group By
df[df[col] > 0.5]       # Rows where the col column is greater than 0.5
df[(df[col] > 0.5) & (df[col] < 0.7)] # Rows where 0.5 < col < 0.7
df.sort_values(col1)    # Sorts values by col1 in ascending order
df.sort_values(col2,ascending=False) # Sorts values by col2 in descending order
df.sort_values([col1,col2], ascending=[True,False])
# Sorts values by col1 in ascending order then col2 in descending order
df.groupby(col) # Returns a groupby object for values from one column
df.groupby([col1,col2])  # Returns a groupby object values from multiple columns
df.groupby(col1)[col2].mean()
# Returns the mean of the values in col2, grouped by the values in col1
# (mean can be replaced with almost any function from the statistics section)
df.pivot_table(index=col1, values= col2,col3], aggfunc=mean)
# Creates a pivot table that groups by col1 and calculates the mean
# of col2 and col3
df.groupby(col1).agg(np.mean)
# Finds the average across all columns for every unique column 1 group
```

```
df.apply(np.mean)        # Applies a function across each column
df.apply(np.max, axis=1) # Applies a function across each row
```

Joining and Combining

In [ ]:
```
#Joining and Combining
df1.append(df2)
# Adds the rows in df1 to the end of df2 (columns should be identical)
pd.concat([df1, df2],axis=1)
# Adds the columns in df1 to the end of df2 (rows should be identical)
df1.join(df2,on=col1,how='inner')
# SQL-style joins the columns in df1 with the columns on df2 where the rows
# for col have identical values. how can be one of 'left', 'right', 'outer',
# 'inner'
```

Writing Data

In [ ]:
```
#Writing Data
df.to_csv(filename)    # Writes to a CSV file
df.to_excel(filename)  # Writes to an Excel file
df.to_sql(table_name, connection_object) # Writes to a SQL table
df.to_json(filename)   # Writes to a file in JSON format
df.to_html(filename)   # Saves as an HTML table
df.to_clipboard()      # Writes to the clipboard
```

3. NumPy

NumPy allows for efficient vectorized calculations. NumPy has its own vectorized versions of many mathematical functions. This can lead to confusion for the beginner, since there is another Python library called Math which also has many mathematical functions. However, Math only allows for scalar functions. In much of scientific computing, it is faster and more efficient to use vectorized code when ever possible. For that reason we will always use NumPy rather than Math for numerical calculations. **HUH???????????????**

In [60]:
```
# import package
import numpy as np
```

To learn more about NumPy, see the official NumPy quick start guide or my NumPy primer github.com/ejwest2/.../numpy-primer.ipynb

3.1 Creating and slicing arrays

You can create arrays in a number of ways. Here are four common examples.

In [61]:
```
# create an array manually
np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0])
```

Out[61]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [62]:
```
# create an array of evenly spaced elements from 0 to 10, with 11 total elements
np.linspace(0, 10, 11)
```

Out[62]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [63]:
```
np.linspace(0, 10, 10) #because when you start with 0, there are 11
```

Out[63]:
```
array([ 0.        ,  1.11111111,  2.22222222,  3.33333333,  4.44444444,
        5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.        ])
```

In [64]:
```
# create an array of evenly spaced elements from 0 to 10, in increments of 1
np.arange(0.0, 11.0, 1)
```

Out[64]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [65]:
```
# create an empty array of zeros
np.zeros(11)
```

Out[65]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [66]:
```
# create a (3x3) array manually
np.array([[1, 2, 3], [4, 5, 6], [6, 7, 8]])
```

Out[66]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [6, 7, 8]])
```

In [67]:
```
# create a higher-dimensional array using one of the many built-in functions.
# here we use zeros() but there is more:
# ones() diag() eye()
# zeros_like()  ones_like()
np.zeros((2,3))
```

Out[67]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Use arrays to construct expressions and carry out numerical operations.

In [68]:
```
# create a sample array
x = np.array([1, 3, 7, 10])
# calculate some elementary functions
f1 = np.cos(x)
f2 = np.sin(x)
f3 = np.exp(x)
f4 = np.log(1+ x)
f5 = np.sqrt(x)
f6 = x + 3*x*x - 4*x**5
f1, f2, f3, f4, f5, f6  # prints all these functions onto seperate lines
# note that you can only have one output print event per code block...
# for example, you can't ask it to print f3 after the f3 definition because the
# final line of code is the output
```

Out[68]:
```
(array([ 0.54030231, -0.9899925 ,  0.75390225, -0.83907153]),
 array([ 0.84147098,  0.14112001,  0.6569866 , -0.54402111]),
 array([2.71828183e+00, 2.00855369e+01, 1.09663316e+03, 2.20264658e+04]),
 array([0.69314718, 1.38629436, 2.07944154, 2.39789527]),
 array([1.        , 1.73205081, 2.64575131, 3.16227766]),
 array([     0,    -942,  -67074, -399690]))
```

Slicing

It is often important to be able to select elements of an array, either individually, or in entire rows, or columns, or other more complicated chunks. This topic is called **SLICING**. Here we illustrate a few common ways to "slice" NumPy arrays.

In [69]:
```
### some ways to slice one-dimensional arrays ###
# create a 1D array
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# select the first element
b = a[0]
# select the last element
c = a[-1]
# select elements 1 to 5
d = a[1:5]
# select the first four elements
e = a[:4]
# select elements the last four elements
f = a[-4:]
# select every other element
g = a[::2]
a, b, c, d, e, f, g
```

Out[69]:
```
(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
 1,
 10,
 array([2, 3, 4, 5]),
 array([1, 2, 3, 4]),
 array([ 7,  8,  9, 10]),
 array([1, 3, 5, 7, 9]))
```

In [70]:
```
### some ways to slice two-dimensional arrays ###
# create a 2D array, in this case a (2x11) array
a = np.array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
              [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]])
b = a[0,4]       # select element in the 1st row, 5th column
c = a[0, :]      # select the entire 1st row
d = a[:, 2]      # select the entire 3rd column
e = a[1, :4]     # select elements in the 2nd row and first four columns
f = a[1, -4:]    # select elements in the 2nd row and last four columns
g = a[1, 2:6]    # select elements in the 2nd row and columns 3-7
a, b, c, d, e, f, g
```

Out[70]:
```
(array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
        [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]]),
 5,
 array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
 array([ 3, 13]),
 array([11, 12, 13, 14]),
 array([17, 18, 19, 20]),
 array([13, 14, 15, 16]))
```

3.2 Addition / Subtraction / Multiplication of matrix

`np.add()`

In [71]:
```
#create first matrix
A = np.array([[1,2,1],[3,4,1]])
print("Matrix A :")
print(A)

#create second matrix
B = np.array([[5,6,1],[7,8,1]])
print("Matrix B :")
print(B)
print(" ")

# adding two matrix
print('Matrix A + Matrix B')
```

```
C = np.add(A,B)
print(C)
```

```
Matrix A :
[[1 2 1]
 [3 4 1]]
Matrix B :
[[5 6 1]
 [7 8 1]]

Matrix A + Matrix B
[[ 6  8  2]
 [10 12  2]]
```

`np.subtract()`

In [72]:
```
#create first matrix
A = np.array([[5,6,1],[7,8,1]])

print("Matrix A :")
print(A)

#create second matrix
B = np.array([[1,2,1],[3,4,1]])
print("Matrix B :")
print(B)
print(" ")

# subtracting two matrix
print('Matrix A - Matrix B = ')
C = np.subtract(A,B)
print(C)
```

```
Matrix A :
[[5 6 1]
 [7 8 1]]
Matrix B :
[[1 2 1]
 [3 4 1]]

Matrix A - Matrix B =
[[4 4 0]
 [4 4 0]]
```

[3x3] martix multiplied by [3x4]

In [73]:
```
# take a 3x3 matrix
X = [[12,7,3],
     [4,5,6],
     [7,8,9]]

# take a 3x4 matrix
Y = [[5,8,1,2],
     [6,7,3,0],
     [4,5,9,1]]

result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]

# iterating by rows of A
for i in range(len(A)):

    # iterating by columns of B
    for j in range(len(B[0])):

        # iterating by rows of B
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]


print("Matrix X :")
print(X)

print("Matrix Y :")
print(Y)
print(" ")

print('Multiplication of matrix X and matrix Y : ')
for r in result:
    print(r)
```

```
Matrix X :
[[12, 7, 3], [4, 5, 6], [7, 8, 9]]
Matrix Y :
[[5, 8, 1, 2], [6, 7, 3, 0], [4, 5, 9, 1]]

Multiplication of matrix X and matrix Y :
[23, 34, 11, 0]
[31, 46, 15, 0]
[0, 0, 0, 0]
```

Multiply DOT PRODUCT

In [74]:
```
# DOT PRODUCT
# take a 3x3 matrix
X = [[12,7,3],
     [4,5,6],
     [7,8,9]]

# take a 3x4 matrix
Y = [[5,8,1,2],
     [6,7,3,0],
     [4,5,9,1]]

# this will be 3x4 matrix
result = np.dot(X,Y)

print("Matrix X :")
print(X)

print("Matrix Y :")
print(Y)
print(" ")

print('Multiplication of matrix X and matrix Y : ')
for r in result:
    print(r)
```

```
Matrix X :
[[12, 7, 3], [4, 5, 6], [7, 8, 9]]
Matrix Y :
[[5, 8, 1, 2], [6, 7, 3, 0], [4, 5, 9, 1]]

Multiplication of matrix X and matrix Y :
[114 160  60  27]
[74 97 73 14]
[119 157 112  23]
```

DOT PRODUCT 2D vector rotation `np.dot`

In [75]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def plot_vect(x, b, xlim, ylim):     # function to plot two vectors,
    # x - the original vector        # b - the transformed vector
    # xlim - the limit for x         # ylim - the limit for y
    plt.figure(figsize = (10, 6))
    # creates arrow from origin to X0,X1  inputs to this function
    plt.quiver(0,0,x[0],x[1],\
        color='k',angles='xy',\
        scale_units='xy',scale=1,\
        label='Original vector')
    plt.quiver(0,0,b[0],b[1],\
        color='g',angles='xy',\
        scale_units='xy',scale=1,\
        label ='Transformed vector')
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.grid(True)
    plt.gca().set_aspect('equal')
    plt.legend()
    plt.show()

A = np.array([[2, 0],[0, 1]]) # A is the transformer   #INPUT
x = np.array([[1],[1]])       # Original Vector        #INPUT
b = np.dot(A, x)
plot_vect(x,b,(0,3),(0,3))    #also set the limits     #INPUT
```

TRANSPOSE using `numpy.matrix.T`

---

printing the transpose of an input matrix by applying the `.T` attribute of the NumPy matrix of the numpy Module

In [76]:
```
# importing NumPy module
import numpy as np

# input matrix
inputMatrix = np.matrix([[6, 1, 5], [2, 0, 8], [1, 4, 3]])
# printing the input matrix
print("Input Matrix:\n", inputMatrix)
print("Transpose of an input matrix\n", inputMatrix.T)
#https://www.tutorialspoint.com/matrix-and-linear-algebra-calculations-in-python
```

```
Input Matrix:
 [[6 1 5]
 [2 0 8]
 [1 4 3]]
Transpose of an input matrix
 [[6 2 1]
 [1 0 4]
 [5 8 3]]
```

INVERSE using `numpy.matrix.I`

In [77]:
```
import numpy as np
inputMatrix = np.matrix([[6, 1, 5],[2, 0, 8],[1, 4, 3]])
print("Input Matrix:\n", inputMatrix)
print("Inverse of an input matrix:\n", inputMatrix.I)
```

```
Input Matrix:
 [[6 1 5]
 [2 0 8]
 [1 4 3]]
Inverse of an input matrix:
 [[ 0.21333333 -0.11333333 -0.05333333]
 [-0.01333333 -0.08666667  0.25333333]
 [-0.05333333  0.15333333  0.01333333]]
```

INVERSE using `numpy.matrix.I`

In [78]:
```
inputMatrix = np.matrix([[6, 1, 5],[2, 0, 8],[1, 4, 3]])
print("Input Matrix:\n", inputMatrix)
print("Inverse of an input matrix:\n", inputMatrix.I)
#Inputmatrix.I is the inverse
```

```
Input Matrix:
 [[6 1 5]
 [2 0 8]
 [1 4 3]]
Inverse of an input matrix:
 [[ 0.21333333 -0.11333333 -0.05333333]
 [-0.01333333 -0.08666667  0.25333333]
 [-0.05333333  0.15333333  0.01333333]]
```

3.3 Solving Systems of Linear Equations $AX = B$

$$40A + 15B = 100$$

$$25B = 50 + 50A$$

As you know, the above system of equations can be written as an Augmented Matrix in the following form:

$$\left[\begin{array}{cc|c} 40 & 15 & 100 \\ -50 & 25 & 50 \end{array}\right]$$

`IPython` and `display, Math` can output answers in nice format, but it requires code that isn't the prettiest...

In [79]:
```
import numpy as np
from IPython.display import display, Math

# Define the matrix A and column vector B
A = np.array([[40, 15], [25, 50]])
B = np.array([100, 50])

# Solve for the column vector X
X = np.linalg.solve(A, B)

# Format and display the results using LaTeX
display(Math(r'A = \begin{{bmatrix}} {} & {} \\ {} & {}\end{{bmatrix}}'.format(*A.flatten())))
display(Math(r'B = \begin{{bmatrix}} {} \\ {}\end{{bmatrix}}'.format(*B)))
display(Math(r'X = \begin{{bmatrix}} {:.2f} \\ {:.2f}\end{{bmatrix}}'.format(*X)))
```

$$A = \begin{bmatrix} 40 & 15 \\ 25 & 50 \end{bmatrix}$$

$$B = \begin{bmatrix} 100 \\ 50 \end{bmatrix}$$

$$X = \begin{bmatrix} 2.62 \\ -0.31 \end{bmatrix}$$

In [80]:
```
import numpy as np
from IPython.display import display, Math

# Define the matrix A and column vector B
A = np.array([[1, 0, 0], [0, 0.6, 0], [0, 0, 1]])
B = np.array([0, 90, 0])

# Solve for the column vector X
X = np.linalg.solve(A, B)

# Format and display the results using LaTeX
display(Math(r'A = \begin{{bmatrix}} {} & {} & {} \\ {} & {} & {} \\ {} & {} & {}\end{{bmatrix}}'.format(*A.flatten())))
display(Math(r'B = \begin{{bmatrix}} {} \\ {} \\ {}\end{{bmatrix}}'.format(*B)))
display(Math(r'X = \begin{{bmatrix}} {:.2f} \\ {:.2f} \\ {:.2f}\end{{bmatrix}}'.format(*X)))
```

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 0.6 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ 90 \\ 0 \end{bmatrix}$$

$$X = \begin{bmatrix} 0.00 \\ 150.00 \\ 0.00 \end{bmatrix}$$

In [81]:
```
import numpy as np
from IPython.display import display, Math

# Define the matrix A and column vector B
A = np.array([[1, 0, 3, 0], [0, 0.6, 0, 0], [0, -0.8, 0.5, 0], [0, 0, 0, 1]])
B = np.array([0, 90, 0, 1])

# Solve for the column vector X
X = np.linalg.solve(A, B)

# Format and display the results using LaTeX
display(Math(r'A = \begin{{bmatrix}} {} & {} & {} & {} \\ {} & {} & {} & {} \\ {} & {} & {} & {} \\ {} & {} & {} & {}\end{{bmatrix
display(Math(r'B = \begin{{bmatrix}} {} \\ {} \\ {} \\ {}\end{{bmatrix}}'.format(*B)))
display(Math(r'X = \begin{{bmatrix}} {:.2f} \\ {:.2f} \\ {:.2f} \\ {:.2f}\end{{bmatrix}}'.format(*X)))
```

$$A = \begin{bmatrix} 1.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.6 & 0.0 & 0.0 \\ 0.0 & -0.8 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ 90 \\ 0 \\ 1 \end{bmatrix}$$

$$X = \begin{bmatrix} -720.00 \\ 150.00 \\ 240.00 \\ 1.00 \end{bmatrix}$$

3.4 Underdetermined Systems

More unknowns than equations - These types of systems can have infinite solutions. i.e., we can not find an unique x such that $Ax = b$. In this case, we can find a set of equations that represent all of the solutions that solve the problem by using Gauss Jordan and the Reduced Row Echelon form. Lets consider the following example:

$$\begin{bmatrix} 5 & -2 & 2 & 1 \\ 4 & -3 & 4 & 2 \\ 4 & -6 & 7 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

What is the Reduced Row Echelon form for A?

Notice how the above RREF form of matrix A is different from what we have seen in the past. In this case not all of our values for x are unique. When we write down a solution to this problem by defining the variables by one or more of the undefined variables. for example, here we can see that x_4 is undefined. So we say $x_4 = x_4$, i.e. x_4 can be any number we want. Then we can define x_3 in terms of x_4. In this case $x_3 = \frac{11}{15} - \frac{4}{15}x_4$. The entire solution can be written out as follows:

$$x_1 = \frac{1}{15} + \frac{1}{15}x_4$$
$$x_2 = \frac{2}{5} + \frac{2}{5}x_4$$
$$x_3 = \frac{11}{15} - \frac{4}{15}x_4$$
$$x_4 = x_4$$

Sometimes, in an effort to make the solution more clear, we introduce new variables (typically, r, s, t) and substitute them in for our undefined variables so the solution would look like the following:

$$x_1 = \frac{1}{15} + \frac{1}{15}r$$
$$x_2 = \frac{2}{5} + \frac{2}{5}r$$
$$x_3 = \frac{11}{15} - \frac{4}{15}r$$
$$x_4 = r$$

We can find a particular solution to the above problem by inputing any number for r. For example, set r equal to zero and create a vector for all of the x values.

$$x_1 = \frac{1}{15}$$
$$x_2 = \frac{2}{5}$$
$$x_3 = \frac{11}{15}$$
$$x_4 = 0$$

In [82]:
```
##here is the same basic math in python

r = 50        #start at r=0 for learning exercise
x = np.matrix([1/15+1/15*r, 2/5+2/5*r, 11/15-4/15*r, r]).T
x
```

Out[82]:
```
matrix([[  3.4],
        [ 20.4],
        [-12.6],
        [ 50. ]])
```

In [83]:
```
# Define the matrix A and column vector B
A = np.array([[5, -2, 2, 1], [4, -3, 4, 2], [4, -6, 4, 2], [4, -6, 7, 4]])
b = np.array([1, 2, 3])
```

Define two more matrixes A, b representing the above system of equations $Ax = b$:

Now let us check our answer by multipying matrix A by our solution x and see if we get b

In [84]:
```
np.allclose(A*x,b)
```

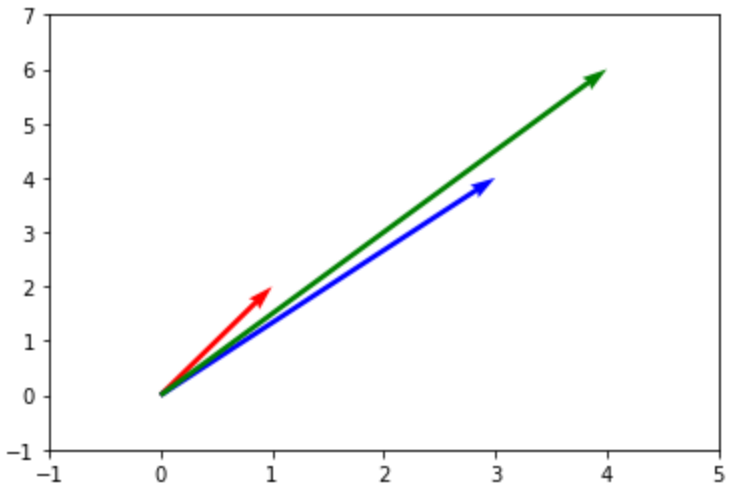
Out[84]:
```
False
```

Now go back and pick a different value for r and see that it also produces a valid solution for $Ax = b$.

Add 2 vectors defined by magnitude angle, plot

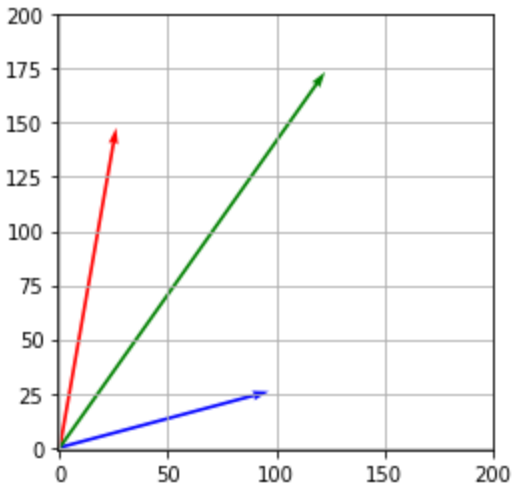
In [85]:
```
import matplotlib.pyplot as plt
import numpy as np

# Define the two vectors
v1 = np.array([1, 2])
v2 = np.array([3, 4])

# Add the two vectors to create the resultant vector
resultant_vector = v1 + v2

# Plot the vectors
origin = [0], [0]
plt.quiver(*origin, v1[0], v1[1], color='r', angles='xy', scale_units='xy', scale=1)
plt.quiver(*origin, v2[0], v2[1], color='b', angles='xy', scale_units='xy', scale=1)
plt.quiver(*origin, resultant_vector[0], resultant_vector[1], color='g', angles='xy', scale_units='xy', scale=1)

# Set the x-limits and y-limits of the plot
plt.xlim(-1, 5)
plt.ylim(-1, 7)
```

```
# Show the plot
plt.show()
```



from polar coordinate input to cartesian, add two vectors

In [86]:
```
import matplotlib.pyplot as plt
import numpy as np

# Define the length and angle (in degrees) of the two vectors
length1, angle1 = 150, 80
length2, angle2 = 100, 15

# Convert the polar coordinates to Cartesian coordinates
v1 = np.array([length1 * np.cos(np.deg2rad(angle1)), length1 * np.sin(np.deg2rad(angle1))])
v2 = np.array([length2 * np.cos(np.deg2rad(angle2)), length2 * np.sin(np.deg2rad(angle2))])

# Add the two vectors to create the resultant vector
resultant_vector = v1 + v2

# Plot the vectors
origin = [0], [0]
plt.quiver(*origin, v1[0], v1[1], color='r', angles='xy', scale_units='xy', scale=1)
plt.quiver(*origin, v2[0], v2[1], color='b', angles='xy', scale_units='xy', scale=1)
plt.quiver(*origin, resultant_vector[0], resultant_vector[1], color='g', angles='xy', scale_units='xy', scale=1)

# Set the x-limits and y-limits of the plot
plt.xlim(-1, 200)
plt.ylim(-1, 200)

# Display a grid
plt.grid()

# Set the aspect ratio of the plot to 'equal' to ensure square proportions
plt.gca().set_aspect('equal')

# Show the plot
plt.show()
```



3.5 Eigenvectors connected to Null Space, Nullity of a Matrix

$$Ax = \lambda x$$

A =

$$\begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix}$$

In [87]:
```
import numpy as np
from numpy.linalg import eig
a = np.array([[0, 2],
              [2, 3]])
w,v=eig(a)
print('E-value:', w)
print('E-vector', v)
#https://pythonnumericalmethods.berkeley.edu/notebooks/chapter15.01-Eigenvalues-and-Eigenvectors-Problem-Statement.html
```

```
E-value: [-1.  4.]
E-vector [[-0.89442719 -0.4472136 ]
 [ 0.4472136  -0.89442719]]
```

A =

$$\begin{bmatrix} 2 & 2 & 4 \\ 1 & 3 & 5 \\ 2 & 3 & 4 \end{bmatrix}$$

In [88]:
```
a = np.array([[2, 2, 4],
              [1, 3, 5],
              [2, 3, 4]])
w,v=eig(a)
print('E-value:', w)
print('E-vector', v)
```

```
E-value: [ 8.80916362  0.92620912 -0.73537273]
E-vector [[-0.52799324 -0.77557092 -0.36272811]
 [-0.604391    0.62277013 -0.7103262 ]
 [-0.59660259 -0.10318482  0.60321224]]
```

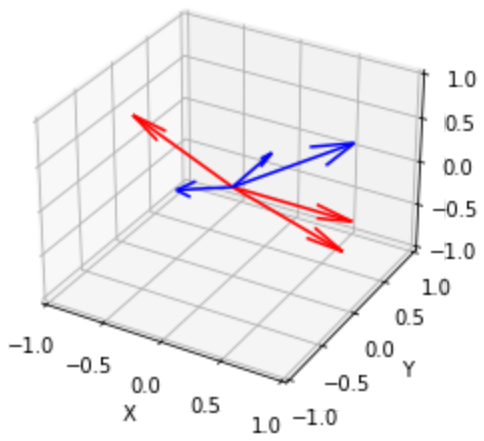
In [89]:
```
from scipy import linalg
import numpy as np
my_arr = np.array([[5,7],[11,3]])   #INPUT
eg_val, eg_vect = linalg.eig(my_arr)
print("The Eigenvalues are :")
print(eg_val)
print("The Eigenvectors are :")
print(eg_vect)
#https://www.tutorialspoint.com/how-can-scipy-be-used-to-calculate-the-eigen-values-and-eigen-vectors-of-a-matrix-in-python
```

```
The Eigenvalues are :
[12.83176087+0.j -4.83176087+0.j]
The Eigenvectors are :
[[ 0.66640536 -0.57999285]
 [ 0.74558963  0.81462157]]
```

In [90]:
```
import numpy as np
from IPython.display import display, Math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the stress tensor
#stress_tensor = np.array([[5, 0, 0], [0, 0, 10], [0, 10,10]])
stress_tensor = np.array([[1, 0, .8660254], [0, 0.6, 0], [0, -0.8, 0.5]])

# Calculate the eigenvalues and eigenvectors of the stress tensor
eigenvalues, eigenvectors = np.linalg.eig(stress_tensor)

# Format and display the eigenvalues and eigenvectors using LaTeX
display(Math(r'\text{{Eigenvalues:}}\quad\lambda = \begin{{bmatrix}} {:.2f} & {:.2f} & {:.2f}\end{{bmatrix}}'.format(*eigenvalues)
display(Math(r'\text{{Eigenvectors:}}\quad V = \begin{{bmatrix}} {:.2f} & {:.2f} & {:.2f} \\ {:.2f} & {:.2f} & {:.2f} \\ {:.2f} &

# Create a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the stress tensor and its eigenvectors
for i in range(3):
    ax.quiver(0, 0, 0, stress_tensor[i][0], stress_tensor[i][1], stress_tensor[i][2], color='b')
    ax.quiver(0, 0, 0, eigenvectors[0][i], eigenvectors[1][i], eigenvectors[2][i], color='r')

# Find the maximum value among the stress tensor and its eigenvectors
max_value = max(np.max(np.abs(stress_tensor)), np.max(np.abs(eigenvectors)))
ax.set_xlim([-max_value, max_value])
ax.set_ylim([-max_value, max_value])
ax.set_zlim([-max_value, max_value])

# Set the axis labels
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Show the plot
plt.show()
```

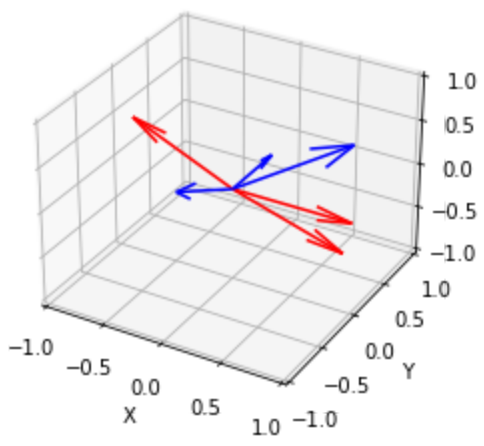
Eigenvalues: $\lambda = \begin{bmatrix} 1.00 & 0.50 & 0.60 \end{bmatrix}$

Eigenvectors: $V = \begin{bmatrix} 1.00 & -0.87 & 0.91 \\ 0.00 & 0.00 & 0.05 \\ 0.00 & 0.50 & -0.42 \end{bmatrix}$



In [91]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the stress tensor
stress_tensor = np.array([[1, 0, .8660254], [0, 0.6, 0], [0, -0.8, 0.5]])
#stress_tensor = np.array([[5, 0, 0], [0, 0, 10], [0, 10,10]])

# Calculate the eigenvalues and eigenvectors of the stress tensor
eigenvalues, eigenvectors = np.linalg.eig(stress_tensor)
print ("eigenvalues:")
print (eigenvalues)
print ("eigenvectors:")
print (eigenvectors)

# Create a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the stress tensor and its eigenvectors
for i in range(3):
    ax.quiver(0, 0, 0, stress_tensor[i][0], stress_tensor[i][1], stress_tensor[i][2], color='b')
    ax.quiver(0, 0, 0, eigenvectors[0][i], eigenvectors[1][i], eigenvectors[2][i], color='r')

# Set the axis labels
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
# Set the axis limits
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
ax.set_zlim([-1, 1])

# Show the plot
plt.show()
```

```
eigenvalues:
[1.  0.5 0.6]
eigenvectors:
[[ 1.         -0.8660254   0.90659683]
 [ 0.          0.          0.05234239]
 [ 0.          0.5        -0.41873914]]
```



In [92]:
```python
import numpy as np

# Define the stress tensor
stress_tensor = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 10]])

# Calculate the eigenvalues and eigenvectors of the stress tensor
eigenvalues, eigenvectors = np.linalg.eig(stress_tensor)

print(eigenvalues)
print('')
print(eigenvectors)
```

```
[16.70749332 -0.90574018  0.19824686]

[[-0.22351336 -0.86584578  0.27829649]
 [-0.50394563  0.0856512  -0.8318468 ]
 [-0.83431444  0.4929249   0.48018951]]
```

3D plot of AX=B solved. THIS is a good explination of eigen vectors.

In [93]:
```python
import numpy as np
from IPython.display import display, Math
```

```
import plotly.graph_objs as go

# Define the stress tensor
stress_tensor = np.array([[10, 0, 0], [0, 0, 2], [0, 3,10]])

# Calculate the eigenvalues and eigenvectors of the stress tensor
eigenvalues, eigenvectors = np.linalg.eig(stress_tensor)

# Format and display the eigenvalues and eigenvectors using LaTeX
display(Math(r'\text{{Eigenvalues:}}\quad\lambda = \begin{{bmatrix}} {:.2f} & {:.2f} & {:.2f}\end{{bmatrix}}'.format(*eigenvalues)
display(Math(r'\text{{Eigenvectors:}}\quad V = \begin{{bmatrix}} {:.2f} & {:.2f} & {:.2f} \\ {:.2f} & {:.2f} & {:.2f} \\ {:.2f} &

# Create a 3D plot
fig = go.Figure()

# Add the stress tensor and its eigenvectors to the plot
for i in range(3):
    fig.add_trace(go.Cone(x=[0], y=[0], z=[0], u=[stress_tensor[i][0]], v=[stress_tensor[i][1]], w=[stress_tensor[i][2]], name='St
    fig.add_trace(go.Cone(x=[0], y=[0], z=[0], u=[eigenvectors[0][i]], v=[eigenvectors[1][i]], w=[eigenvectors[2][i]], name='Eigen

# Set the axis labels
fig.update_layout(scene=dict(xaxis_title='X', yaxis_title='Y', zaxis_title='Z'))

# Show the plot
fig.show()
```

Eigenvalues: $\lambda = \begin{bmatrix} 10.57 & -0.57 & 10.00 \end{bmatrix}$

Eigenvectors: $V = \begin{bmatrix} 0.00 & 0.00 & 1.00 \\ 0.19 & 0.96 & 0.00 \\ 0.98 & -0.27 & 0.00 \end{bmatrix}$

3. Matplotlib

Matplotlib is the most commonly used graphics package in Python. We will usually use it to plot static 2D graphics, but it can also be used to construct 3D graphics or animations. Those are more advanced topics that we will discuss only if and when we need to. Here we just provide the basics of constructing 2D plots. NOTE doesn't work well in Colab!!!

As usual, the first step is to load the necessary packages. Here we load both Matplotlib and NumPy.

In [94]:
```
# import packages
import numpy as np
import matplotlib.pyplot as plt
```

When working in Jupyter notebook, we also need to tell Matplotlib which graphics backend to use. (NOTE: once you choose a backend, you cannot change it without restarting the notebook kernel.) The 'notebook' backend is the one we will use, as it is fairly standard. But you can experiment with otehrs if you like. To list the available backends and select a backend, use the following line magics.

In [95]:
```
# list available backends
%matplotlib --list

#works in VSCode - well now it works in COLAB
%matplotlib inline

# select a graphics backend compatable with jupyter notebooks -
# but does it work in Colab!!!
#%matplotlib notebook
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'wx', 'qt4', 'qt5', 'qt', 'osx', 'nbagg', 'notebook', 'agg', 'svg', 'pdf', 'p
s', 'inline', 'ipympl', 'widget']
```

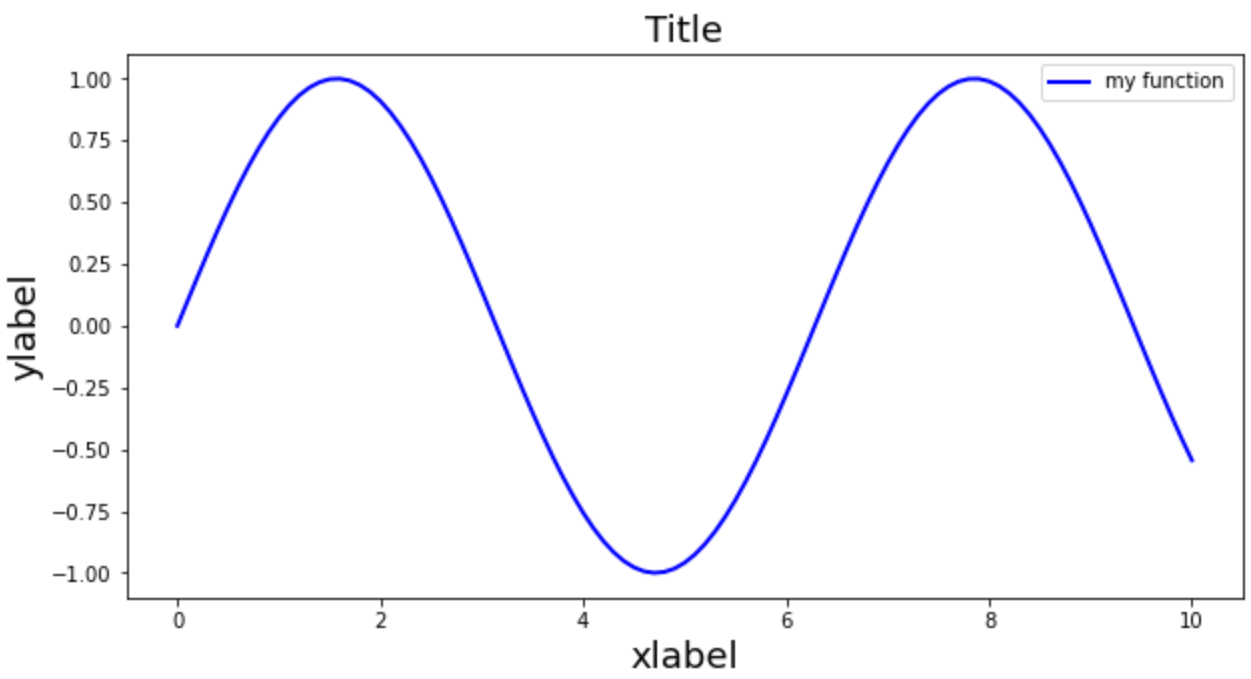
We're almost ready to plot. First, let's create a grid. Since we will often plot quantities as functions of time, here we create a uniform time grid as follows.

In [96]:
```
# create a uniform time grid from 0 to 10, with 101 grid points
t = np.linspace(0, 10, 101)
```

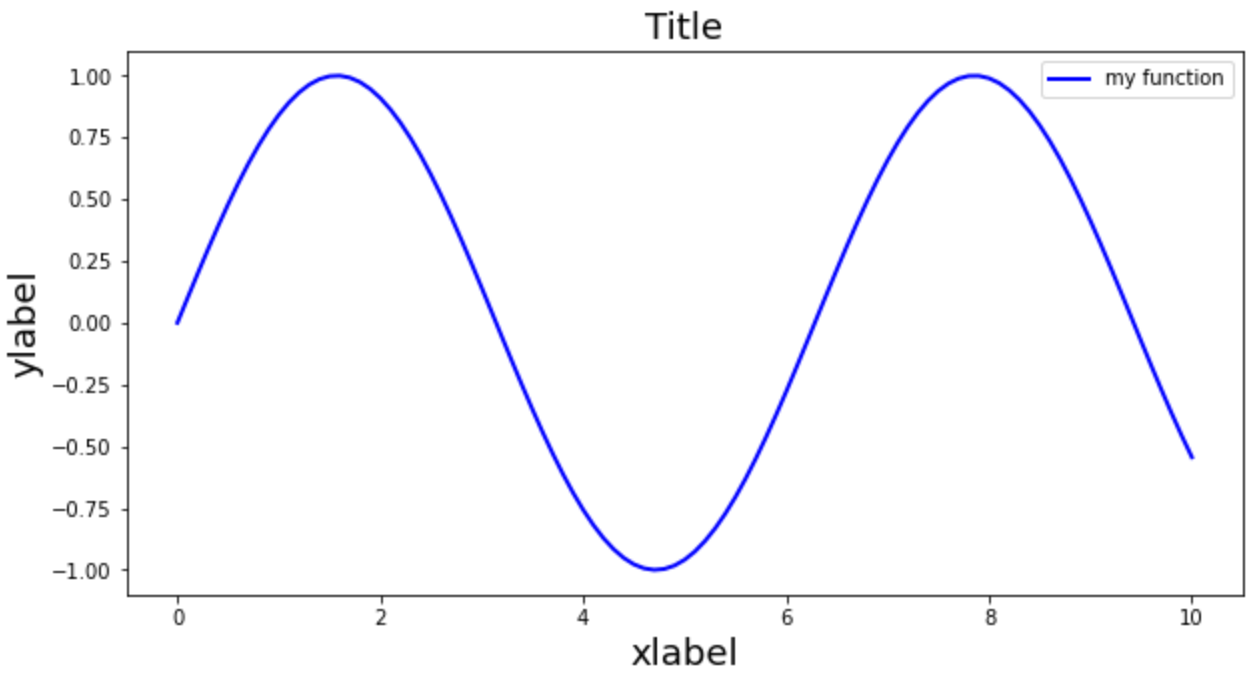
Now we show three common examples of static 2D plots: (1) a single function, (2) multiple functions on the same plot, (3) an array of plots.

single function plot

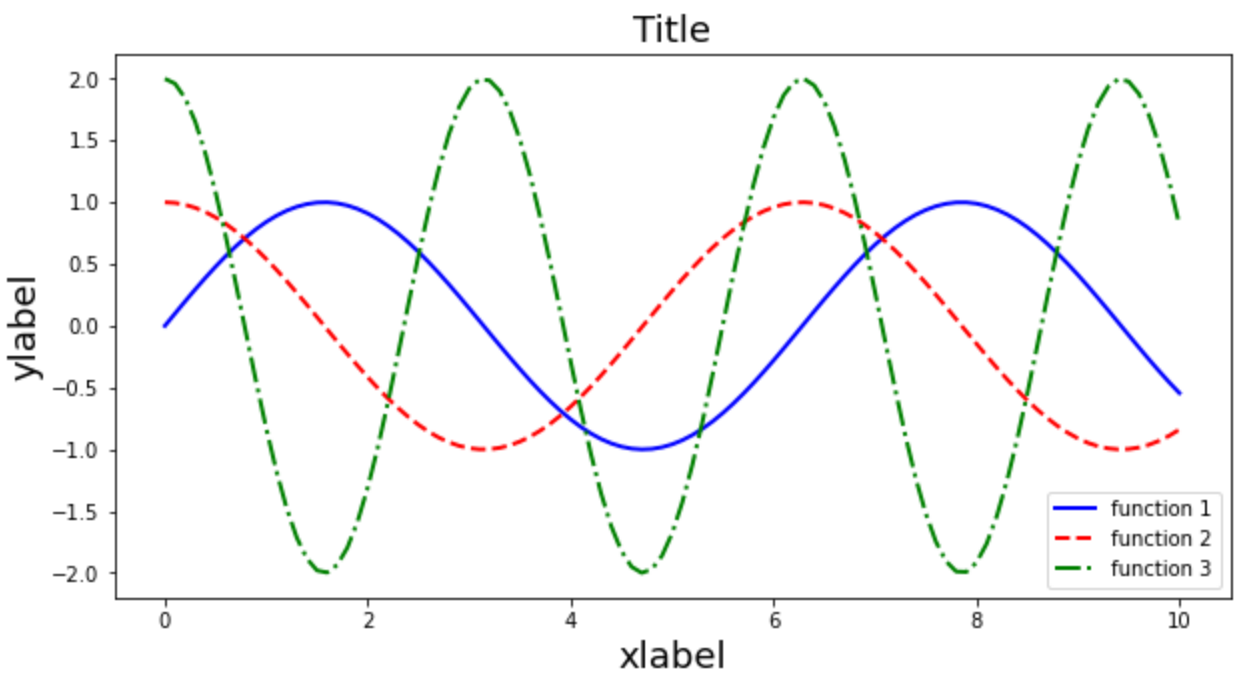
In [97]:
```
# define function
def fxn(t):
    return np.sin(t)
# plot
plt.figure(figsize=(10,5))
plt.plot(t, fxn(t), 'b-', linewidth=2, label='my function')
plt.title('Title', fontsize=18)
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.legend()
plt.show()
```



In [98]:
```
# define function
def fxn(t):
    return np.sin(t)
# plot
plt.figure(figsize=(10,5))
plt.plot(t, fxn(t), 'b-', linewidth=2, label='my function')
plt.title('Title', fontsize=18)
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.legend()
plt.show()
```

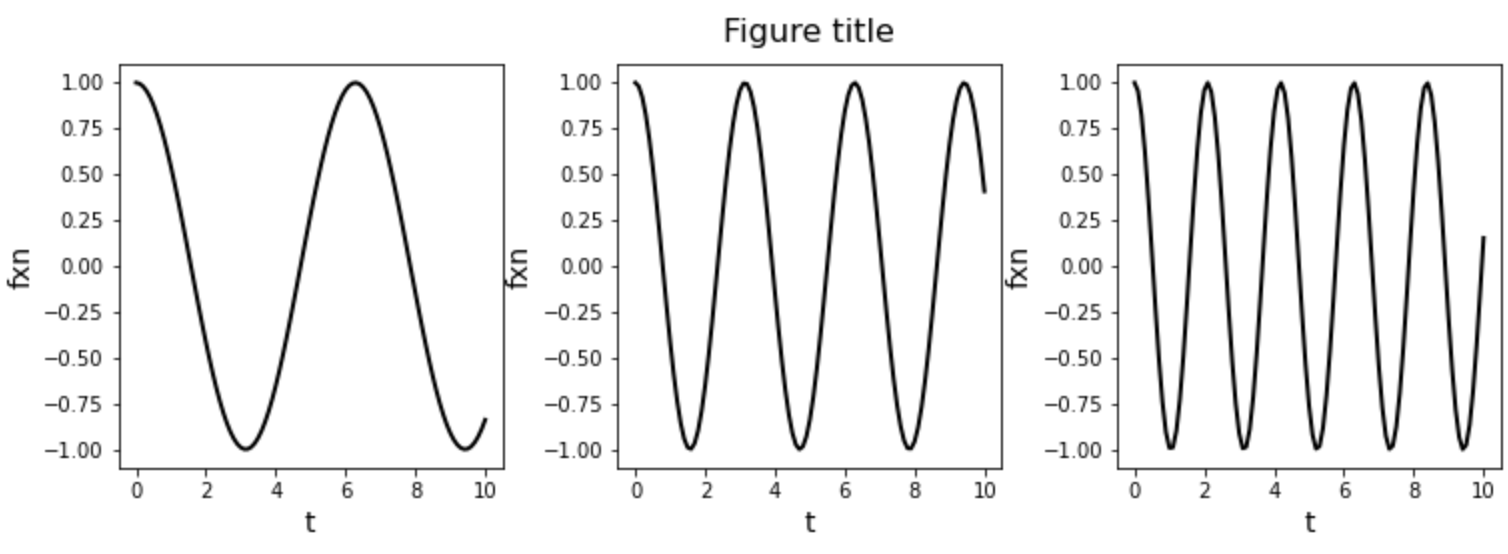


In [99]:
```
### multiple functions on the same plot ###
# this will not plot if you run this from Colab
# define functions
def fxn1(t):
    return np.sin(t)

def fxn2(t):
    return np.cos(t)

def fxn3(t):
    return 2*np.cos(2*t)

# plot
plt.figure(figsize=(10,5))
plt.plot(t, fxn1(t), 'b-', linewidth=2, label='function 1')
plt.plot(t, fxn2(t), 'r--', linewidth=2, label='function 2')
plt.plot(t, fxn3(t), 'g-.', linewidth=2, label='function 3')
plt.title('Title', fontsize=18)
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.legend()
plt.show()
```



In [100…

```
### array of plots ###
# this will not plot if you run this from Colab
# define function
def fxn4(t, omega):
    return np.cos(omega*t)

# array of parameters to use
omega = np.array([1, 2, 3])

# make a figure with multiple subplots
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(12,4))
fig.suptitle('Figure title', size=16)
fig.subplots_adjust(left=0.1, right=0.9, bottom=0.2, top=0.9, hspace=0.2, wspace=0.3)
# subplot 1
ax1.plot(t, fxn4(t, omega[0]), color='black', linestyle='-', linewidth=2)
ax1.set_xlabel('t',size=14)
ax1.set_ylabel('fxn',size=14)
# subplot 2
ax2.plot(t, fxn4(t, omega[1]), color='black', linestyle='-', linewidth=2)
ax2.set_xlabel('t',size=14)
ax2.set_ylabel('fxn',size=14)
# subplot 3
ax3.plot(t, fxn4(t, omega[2]), color='black', linestyle='-', linewidth=2)
ax3.set_xlabel('t',size=14)
ax3.set_ylabel('fxn',size=14)
# show plots
plt.show()
```



Further reading

To learn more about the features and capabilities of Matplotlib, see the official tutorials for beginners

https://matplotlib.org/tutorials/index.html,

or my Matplotlib primer

https://github.com/ejwest2/sandbox/blob/master/python/matplotlib-primer/matplotlib-primer.ipynb.

Matplotlib Primer

Matplotlib is the standard graphics package in Python. The examples below provide an introduction to the different sort of data visualization that can be done using Matplotlib.

We import matplotlib along with numpy. Numpy will be used extensively in many of the examples that follow.

In [101…

```
# import packages
import numpy as np
import matplotlib.pyplot as plt
```

When working in Jupyter notebook, we also need to tell matplotlib which graphics backend to use. We do this with the line magic as follows. The most common backends are 'inline' for non-interactive plotting, and 'notebook' for interactive plotting.

In [102…
```
# uncomment the following line to use the notebook backend
#%matplotlib notebook

# uncomment the following line to use the inline backend
#%matplotlib inline
### NOTE: savefig() does not currently appear to work with inline backend
```

There are many others backends available to choose from (gtk, tk, qt, ...). Experiment to find the one that works best for your purposes. (NOTE: changing the graphics backend usually requires restarting the kernel.) You can list the available backends using the following line magic command.

In [103…
```
# list available backends
%matplotlib --list
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'wx', 'qt4', 'qt5', 'qt', 'osx', 'nbagg', 'notebook', 'agg', 'svg', 'pdf', 'ps', 'inline', 'ipympl', 'widget']
```

We will also create an auxiliary directory, which is where output files will be saved to.

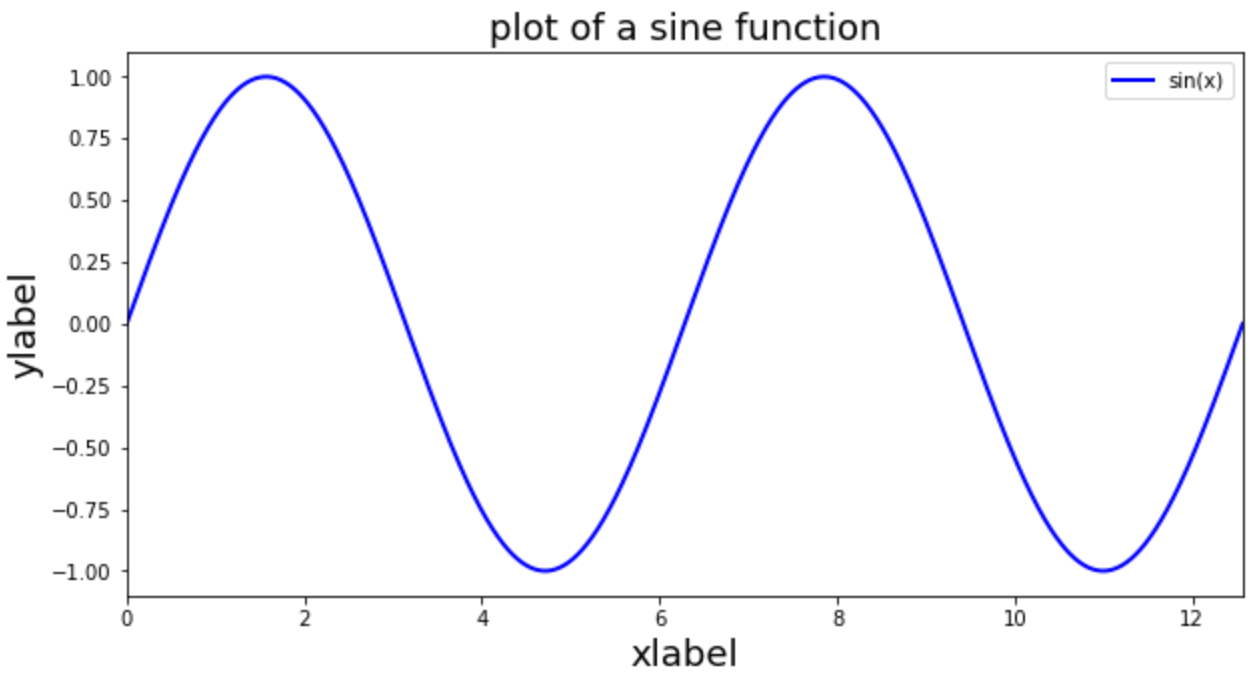
In [104…
```
# create auxiliary directory
import os
folder = 'matplotlib_files'
if not os.path.exists(folder):
    os.mkdir(folder)
```

2. Plotting in 2D

Single curve, single plot

In [105…
```
# create xgrid
xmin = 0
xmax = 4*np.pi
xsteps = 1001
x = np.linspace(xmin, xmax, xsteps)

# define a function (dependent variable)
y = np.sin(x)

# plot functions
plt.figure(figsize=(10,5))
plt.plot(x, y, 'b-', linewidth=2, label='sin(x)')
plt.xlim(xmin, xmax)
plt.title('plot of a sine function', fontsize=18)
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.legend()

# save to EPS file
#file = 'sine.eps'
#plt.savefig(folder + '/' + file)
```

Out[105… `<matplotlib.legend.Legend at 0x1e66cda0c10>`

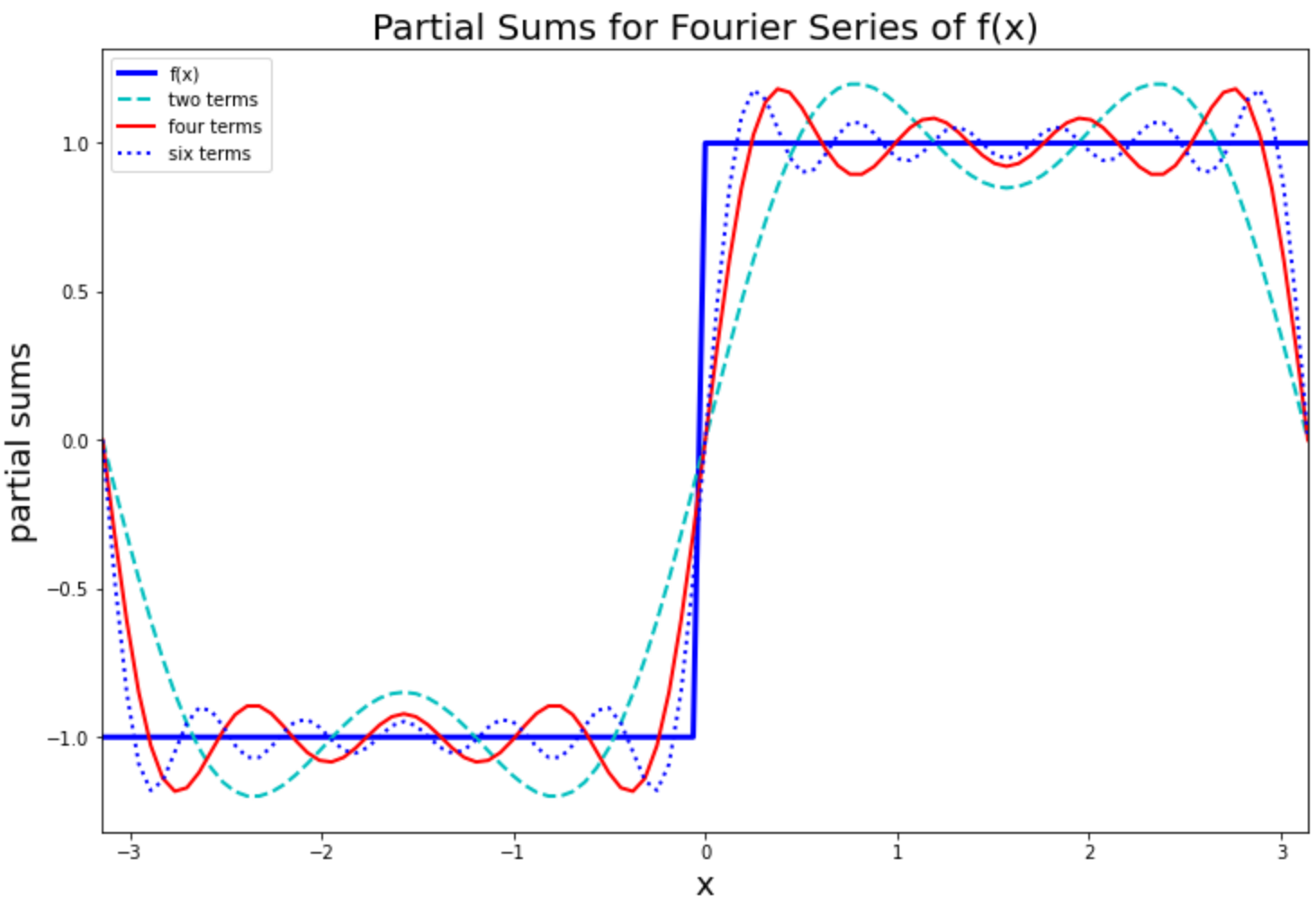


Multiple curves on the same plot

In [106…
```
# create xgrid
xmin = -np.pi
xmax = np.pi
xsteps = 101
x = np.linspace(xmin, xmax, 101)

# define a simple step function
f = np.ones_like(x) #initial array of ones, same size as x
f[x<0] = -1         #set values for x<0 to be -1

# construct partial sums in Fourier sine series
y1 = (4/np.pi)*(np.sin(x) + np.sin(3*x)/3.0)
y2 = y1 + (4/np.pi)*(np.sin(5*x)/5.0 + np.sin(7*x)/7.0)
y3 = y2 + (4/np.pi)*(np.sin(9*x)/9.0 + np.sin(11*x)/11.0)

# plot multiple curves on a single figure
plt.figure(figsize=(12,8))
```

```
plt.plot(x, f, 'b-', lw=3, label='f(x)')
plt.plot(x, y1, 'c--', lw=2, label='two terms')
plt.plot(x, y2, 'r-', lw=2, label='four terms')
plt.plot(x, y3, 'b:', lw=2, label='six terms')
plt.xlim(xmin, xmax)
plt.xlabel('x', fontsize=18)
plt.ylabel('partial sums', fontsize=18)
plt.title('Partial Sums for Fourier Series of f(x)', fontsize=20)
plt.legend(loc='best')

# save to file
#file = 'multicurve.eps'
#plt.savefig(folder + '/' + file)
```

Out[106… <matplotlib.legend.Legend at 0x1e66cdee430>



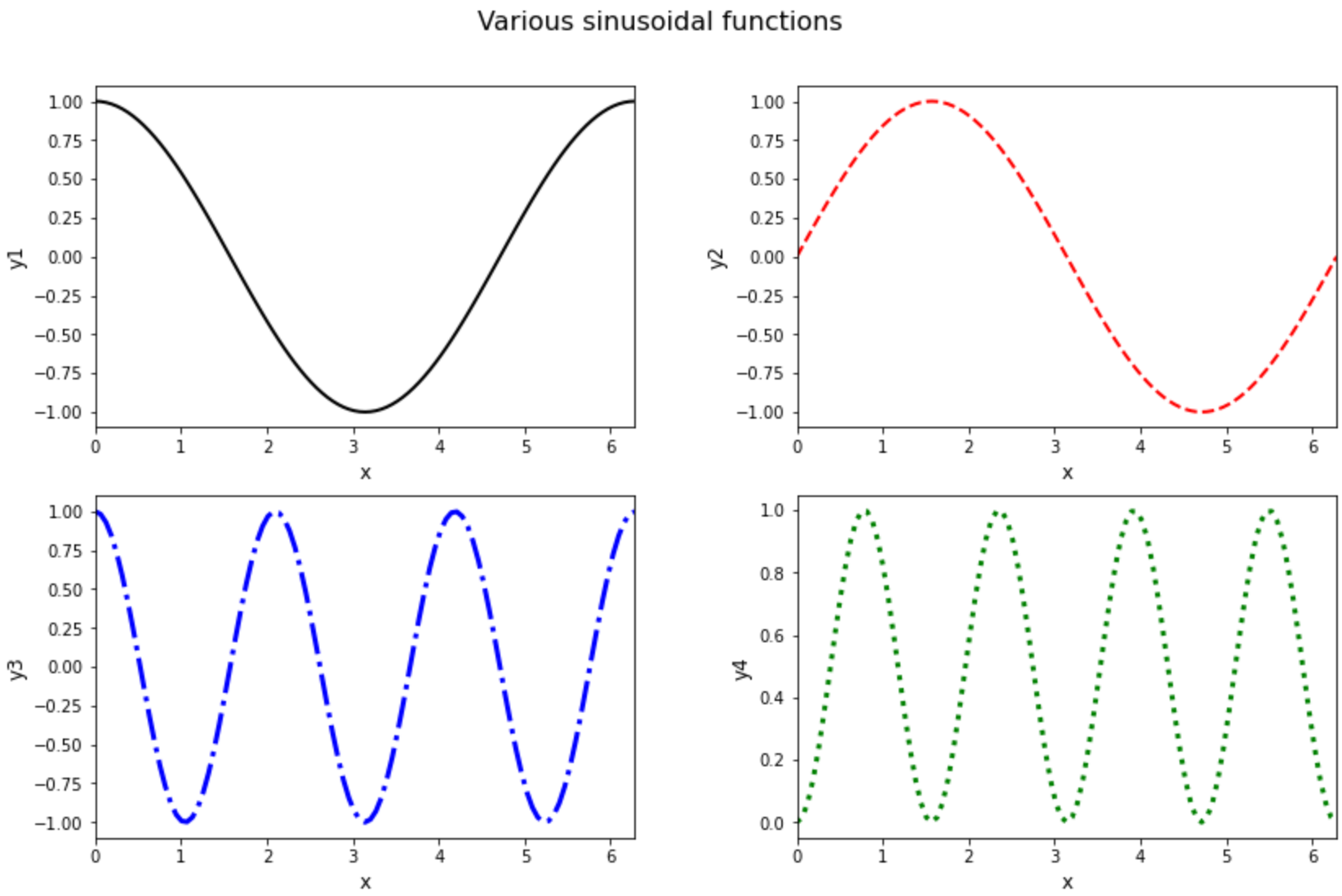
An array of plots

In [107…

```
# create xgrid
xmin= 0
xmax=2*np.pi
xsteps = 101
x = np.linspace(xmin, xmax, xsteps)

# define some functions
y1 = np.cos(x)
y2 = np.sin(x)
y3 = np.cos(3*x)
y4 = np.sin(2*x)**2

# make a figure with multiple subplots
fig = plt.figure(figsize=(12,8))
fig.suptitle('Various sinusoidal functions', size=16)
fig.subplots_adjust(left=0.1, right=0.98, bottom=0.1, top=0.9, hspace=0.2, wspace=0.3)

# plot function 1 on its own subplot
ax1 = fig.add_subplot(2,2,1)
ax1.plot(x, y1, color='black', linestyle='-', linewidth=2)
ax1.set_xlim(xmin, xmax)
ax1.set_xlabel('x',size=12)
ax1.set_ylabel('y1',size=12)

# plot function 2 on its own subplot
ax2 = fig.add_subplot(2,2,2)
ax2.plot(x, y2, color='red', linestyle='--', linewidth=2)
ax2.set_xlim(xmin, xmax)
ax2.set_xlabel('x',size=12)
ax2.set_ylabel('y2',size=12)

# plot function 3 on its own subplot
ax3 = fig.add_subplot(2,2,3)
ax3.plot(x, y3, color='blue', linestyle='-.', linewidth=3)
ax3.set_xlim(xmin, xmax)
ax3.set_xlabel('x',size=12)
ax3.set_ylabel('y3',size=12)

# plot function 4 on its own subplot
ax4 = fig.add_subplot(2,2,4)
ax4.plot(x, y4, color='green', linestyle=':', linewidth=3)
ax4.set_xlim(xmin, xmax)
ax4.set_xlabel('x',size=12)
ax4.set_ylabel('y4',size=12)

# save to file
```

```
#file = 'multiplot.eps'
#plt.savefig(folder + '/' + file)
```

Out[107… Text(0, 0.5, 'y4')

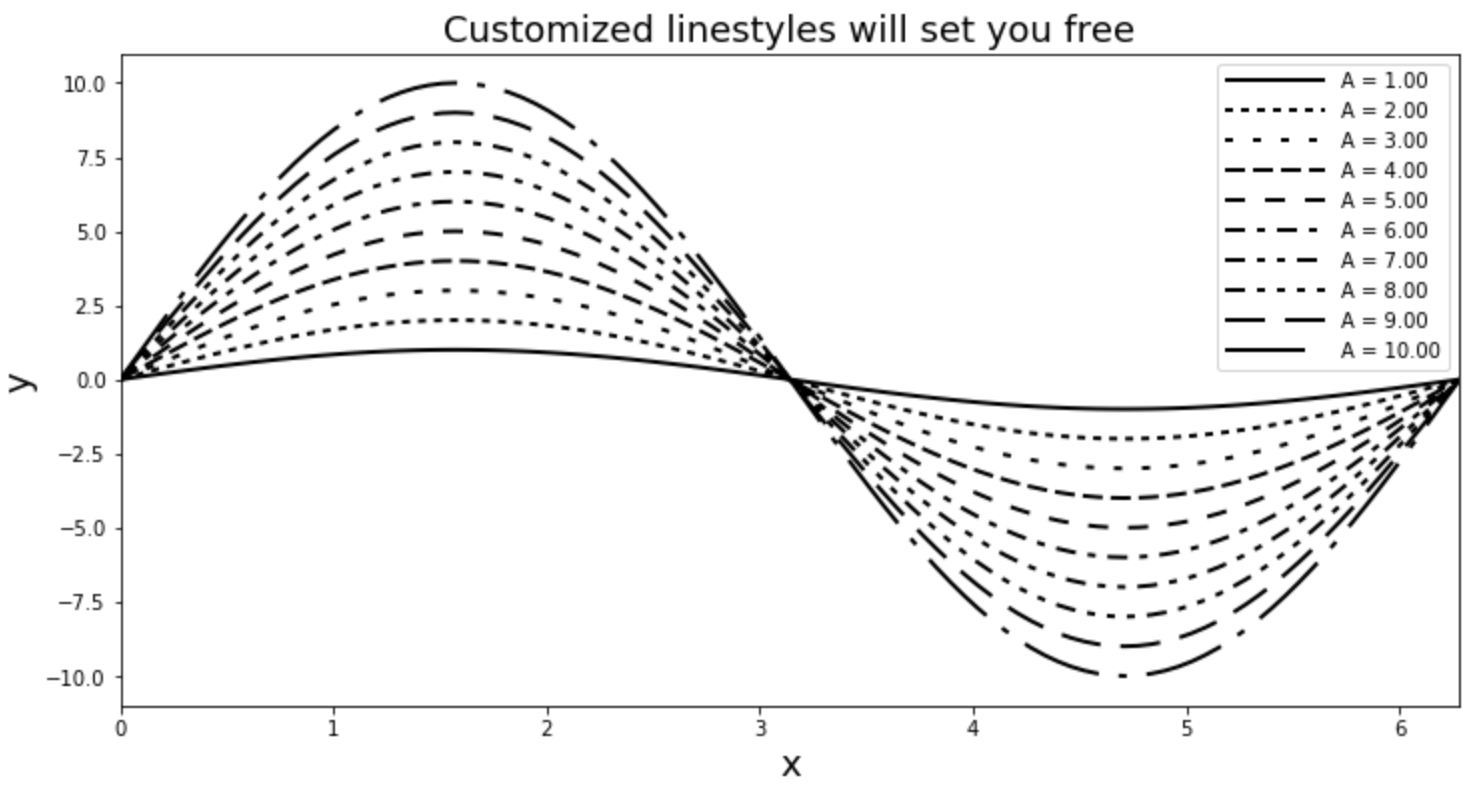


Customized linestyles

In [108…

```
# create xgrid
xmin= 0
xmax=2*np.pi
xsteps = 101
x = np.linspace(xmin, xmax, xsteps)

# amplitudes
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# plot multiple curves with customized linestyles
# use ls=(offset, onoffseq), where onoffseq=(on, off, on, off, ...)
plt.figure(figsize=(12,6))
plt.plot(x, A[0]*np.sin(x), 'k', ls=(0,()), lw=2, label='A = %.2f' % A[0])
plt.plot(x, A[1]*np.sin(x), 'k', ls=(0,(2, 2)), lw=2, label='A = %.2f' % A[1])
plt.plot(x, A[2]*np.sin(x), 'k', ls=(0,(2, 5)), lw=2, label='A = %.2f' % A[2])
plt.plot(x, A[3]*np.sin(x), 'k', ls=(0,(5, 2)), lw=2, label='A = %.2f' % A[3])
plt.plot(x, A[4]*np.sin(x), 'k', ls=(0,(5, 5)), lw=2, label='A = %.2f' % A[4])
plt.plot(x, A[5]*np.sin(x), 'k', ls=(0,(5, 3, 2, 3)), lw=2, label='A = %.2f' % A[5])
plt.plot(x, A[6]*np.sin(x), 'k', ls=(0,(5, 3, 2, 3, 2, 3)), lw=2, label='A = %.2f' % A[6])
plt.plot(x, A[7]*np.sin(x), 'k', ls=(0,(5, 3, 2, 3, 2, 3, 2, 3)), lw=2, label='A = %.2f' % A[7])
plt.plot(x, A[8]*np.sin(x), 'k', ls=(0,(10, 5)), lw=2, label='A = %.2f' % A[8])
plt.plot(x, A[9]*np.sin(x), 'k', ls=(0,(20, 5, 2, 5)), lw=2, label='A = %.2f' % A[9])
plt.xlim(xmin, xmax)
plt.xlabel('x', fontsize=18)
plt.ylabel('y', fontsize=18)
plt.title('Customized linestyles will set you free', fontsize=18)
plt.legend(loc='upper right', handlelength=5)

# save to file
#file = 'linestyles.eps'
#plt.savefig(folder + '/' + file)
```

Out[108… <matplotlib.legend.Legend at 0x1e66d16ee20>

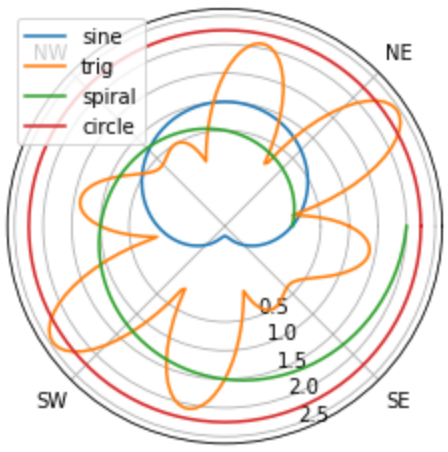


Polar plot

In [109…

```
# create theta grid
theta = np.linspace(0,2*np.pi,201)

# define some polar functions
r1 = np.sin(theta)
r2 = np.abs(np.cos(5.0*theta) - 1.5*np.sin(3.0*theta))
r3 = theta/np.pi
r4 = 2.25*np.ones_like(theta)
ax = plt.polar()
#ax.thetagrids()
#ax.thetagrids(np.arange(45, 360, 90) , ('NE', 'NW', 'SW', 'SE'))
#ax.thetagrids(np.arange(45,360,90), ('NE','NW','SW','SE'))


# plot
plt.polar(theta,r1,label='sine')
plt.polar(theta,r2,label='trig')
plt.polar(theta,r3,label='spiral')
plt.polar(theta,r4,label='circle')
plt.thetagrids(np.arange(45,360,90),('NE','NW','SW','SE'))
plt.rgrids((0.5, 1.0, 1.5, 2.0, 2.5), angle=290)
plt.legend(loc='best')

# save to file
#file = 'polar.eps'
#plt.savefig(folder + '/' + file)
```

Out[109… <matplotlib.legend.Legend at 0x1e66d707e80>

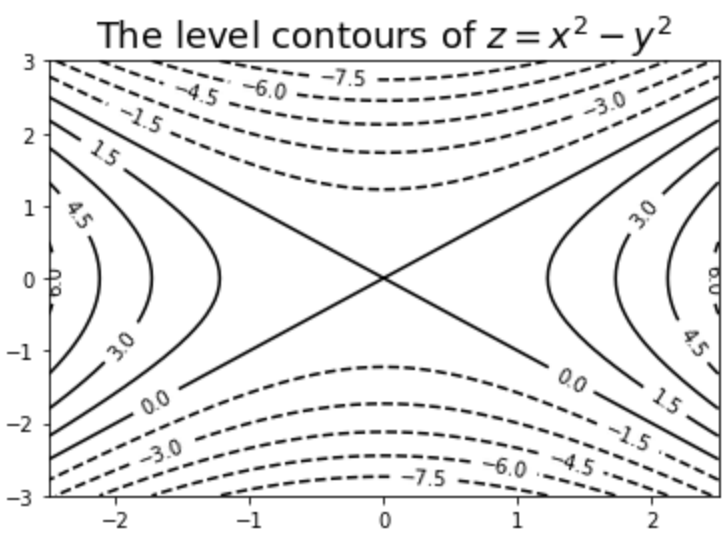


Contour plot

In [110…

```
# create mesh grid
[X,Y] = np.mgrid[-2.5:2.5:51j, -3:3:61j]
Z = X*X - Y*Y

# plot
curves = plt.contour(X,Y,Z, 12, colors='k')
plt.clabel(curves)
plt.title(r'The level contours of $z=x^2-y^2$',size=18)

# save to file
#file = 'contour.eps'
#plt.savefig(folder + '/' + file)
```

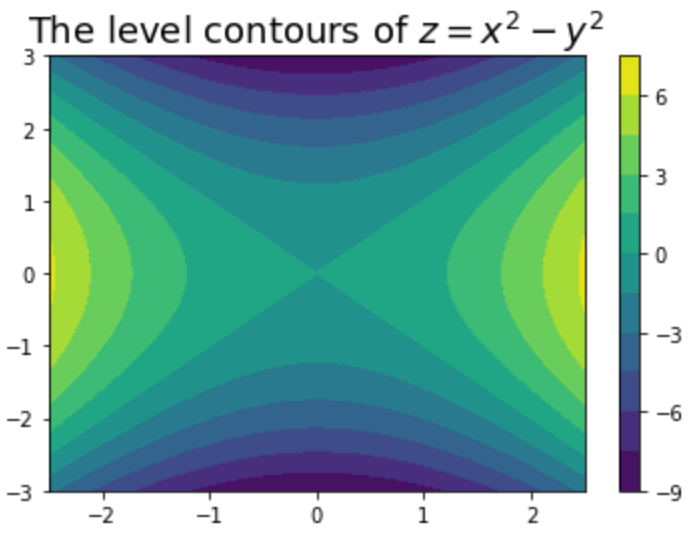
Out[110… Text(0.5, 1.0, 'The level contours of $z=x^2-y^2$')



Filled contour plot

In [111…
```
# create mesh grid
[X,Y] = np.mgrid[-2.5:2.5:51j, -3:3:61j]
Z = X*X - Y*Y

# plot
curves = plt.contourf(X,Y,Z, 12)
plt.colorbar()
plt.title(r'The level contours of $z=x^2-y^2$',size=18)

# save to file
#file = 'contour_filled.eps'
#plt.savefig(folder + '/' + file)
```

Out[111…
```
Text(0.5, 1.0, 'The level contours of $z=x^2-y^2$')
```



3. Plotting in 3D

For 3D graphics, it is necessary to load the following package.

In [112…
```
from mpl_toolkits.mplot3d import Axes3D
```

Surface plots

In [113…
```
# create 1D grids (independent variables)
X = np.arange(-5, 5, 1.)
Y = np.arange(-5, 5, 1.)
```

In [114…
```
X
```

Out[114…
```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

In [115…
```
Y
```

Out[115…
```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

In [116…
```
# create mesh grid
Y, X = np.meshgrid(X, Y)
```

In [117…
```
X
```

Out[117…
```
array([[-5., -5., -5., -5., -5., -5., -5., -5., -5., -5.],
       [-4., -4., -4., -4., -4., -4., -4., -4., -4., -4.],
       [-3., -3., -3., -3., -3., -3., -3., -3., -3., -3.],
       [-2., -2., -2., -2., -2., -2., -2., -2., -2., -2.],
       [-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.]])
```

In [118…
```
Y
```

Out[118…

```
array([[-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.],
       [-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

In [119…

```
# define a grid function
R = np.sqrt(X*X + Y*Y)
Z = np.sin(R)
Z
```

Out[119…

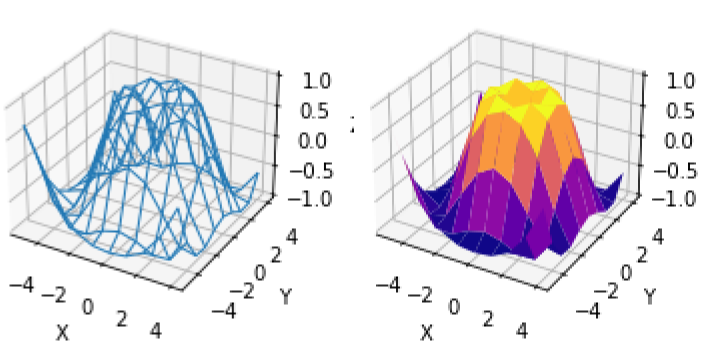
```
array([[ 0.70886129,  0.11965158, -0.43697552, -0.7820949 , -0.92618484,
        -0.95892427, -0.92618484, -0.7820949 , -0.43697552,  0.11965158],
       [ 0.11965158, -0.58617619, -0.95892427, -0.9712778 , -0.83133918,
        -0.7568025 , -0.83133918, -0.9712778 , -0.95892427, -0.58617619],
       [-0.43697552, -0.95892427, -0.89168225, -0.44749175, -0.02068353,
         0.14112001, -0.02068353, -0.44749175, -0.89168225, -0.95892427],
       [-0.7820949 , -0.9712778 , -0.44749175,  0.30807174,  0.78674913,
         0.90929743,  0.78674913,  0.30807174, -0.44749175, -0.9712778 ],
       [-0.92618484, -0.83133918, -0.02068353,  0.78674913,  0.98776595,
         0.84147098,  0.98776595,  0.78674913, -0.02068353, -0.83133918],
       [-0.95892427, -0.7568025 ,  0.14112001,  0.90929743,  0.84147098,
         0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
       [-0.92618484, -0.83133918, -0.02068353,  0.78674913,  0.98776595,
         0.84147098,  0.98776595,  0.78674913, -0.02068353, -0.83133918],
       [-0.7820949 , -0.9712778 , -0.44749175,  0.30807174,  0.78674913,
         0.90929743,  0.78674913,  0.30807174, -0.44749175, -0.9712778 ],
       [-0.43697552, -0.95892427, -0.89168225, -0.44749175, -0.02068353,
         0.14112001, -0.02068353, -0.44749175, -0.89168225, -0.95892427],
       [ 0.11965158, -0.58617619, -0.95892427, -0.9712778 , -0.83133918,
        -0.7568025 , -0.83133918, -0.9712778 , -0.95892427, -0.58617619]])
```

In [120…

```
#plot grid function as 2D surface

# create figure
fig = plt.figure()

# wireframe
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_wireframe(X, Y, Z, linewidth=1, rstride=1, cstride=1)
ax1.set_zlim(-1.01, 1.01)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

# colormap
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(X, Y, Z, cmap='plasma', linewidth=0, rstride=1, cstride=1)
ax2.set_zlim(-1.01, 1.01)
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_zlabel('Z')

# save to file
#file = 'surface2D.eps'
#plt.savefig(folder + '/' + file)
```

Out[120…

```
Text(0.5, 0, 'Z')
```



3D curve

In [121…

```
# define theta grid
theta = np.linspace(0, 2*np.pi, 401)

# define a parametric curve
a = 0.3
m = 11.0
n = 9.0
x = (1 + a*np.cos(n*theta))*np.cos(m*theta)
y = (1 + a*np.cos(n*theta))*np.sin(m*theta)
z = a*np.sin(n*theta)

# plot with mplot3d (via matplotlib)
fig = plt.figure()


ax = Axes3D(fig)
ax.plot(x,y,z, 'g', lw=2)
ax.set_zlim3d(-1.0, 1.0)
```

```
ax.set_xlabel('x', size=16)
ax.set_ylabel('y', size=16)
ax.set_zlabel('z', size=16)
ax.set_title('A spiral as a parametric curve', size=18)

# save to file
#file = 'curve3D.pdf'
#plt.savefig(folder + '/' + file)
```

Out[121… Text(0.5, 0.92, 'A spiral as a parametric curve')

<Figure size 432x288 with 0 Axes>

4. SymPy

SymPy allows you to perform symbolic calculations and manipulations. Sympy behaves a little differently - there is only one line of output and the answers are concatenated. Unlike latex where you can write text and add charecters and write out equations, a code block can only output a single sting.

4.1 QUICK INTRO

In [122…
```
# sympy preliminaries
import sympy as sym   #imports sympy
sym.init_printing()   #turns on fancy printing
```

In [123…
```
# examples of symbol declarations
x, y, z = sym.symbols('x, y, z')
alpha, beta, gamma = sym.symbols('alpha, beta, gamma')
a1, a2, a3 = sym.symbols('a1, a2, a3')
x, y, z, alpha, beta, gamma, a1, a2, a3
```

Out[123… $\left(x,\ y,\ z,\ \alpha,\ \beta,\ \gamma,\ a_{1},\ a_{2},\ a_{3}\right)$

In [124…
```
# mathematical constants
sym.pi, sym.E, sym.I, sym.oo
```

Out[124… $(\pi,\ e,\ i,\ \infty)$

Now you can define functions and expressions in terms of the defined symbols. Note that SymPy has its own versions of most mathematical functions, which need to be called using the appropriate namespace prefix (above I defined this as 'sym').

In [125…
```
# some elementary functions
f1 = sym.cos(x)
f2 = sym.sin(x)
f3 = sym.exp(x)
f4 = sym.log(x)
f5 = x + y**2 + 1/z
f6 = sym.sqrt(x)
f1, f2, f3, f4, f5, f6
```

Out[125… $\left(\cos(x),\ \sin(x),\ e^{x},\ \log(x),\ x + y^{2} + \frac{1}{z},\ \sqrt{x}\right)$

Finally, here are some common operations to get you started using SymPy.

In [126…
```
# differentiate
diff1 = sym.diff(f1, x)
diff2 = sym.diff(f1, x, 2)
diff1, diff2
```

Out[126… $(-\sin(x),\ -\cos(x))$

In [127…
```
# integrate
int1 = sym.integrate(f4, x)
int2 = sym.integrate(f4, (x, 1, 10))
int1, int2
```

Out[127… $(x\log(x) - x,\ -9 + 10\log(10))$

In [128…
```
# series expansion around x0, to order n
f2.series(x, x0=0, n=10)
```

Out[128… $x - \frac{x^{3}}{6} + \frac{x^{5}}{120} - \frac{x^{7}}{5040} + \frac{x^{9}}{362880} + O\left(x^{10}\right)$

In [129…
```
# substitution
f2.subs(x, y*z + 3)
```

Out[129… $\sin(yz + 3)$

In [130…
```
# evaluate to n digits
n = 100
f1.subs(x, sym.pi/4).evalf(n)     #remember f1 is COS(X)
#This means that the expression f1.subs(x, sym.pi/4) is equivalent to the expression f1(sym.pi/4).
#so this is like taking the Cos of 45 degrees... which is (root2)/2
```

Out[130…
$$0.70710678118654752440084436210484903928483593768847403658833986899536623923105351942519376716382078 64$$

lambdify converts a SymPy expression into a function that allows for fast numeric evaluation.

In [131…
```
import numpy as np
# convert a sympy function to a numpy-friendly function using lambdify()
x, A, k = sym.symbols('x, A, k')
symfxn = A*sym.cos(k*x)
npfxn = sym.lambdify((x, A, k), symfxn, 'numpy')
# test the new lambdified function
npfxn(np.pi/3, 2, 2)
```

Out[131…
$$-1.0$$

DO THIS! I DON"T FOLLOW THE LAMBDIFY !!!!!!!!!!!!!!!!!

For more information about SymPy, see the official SymPy tutorial for beginners https://docs.sympy.org/latest/tutorial/index.html,

or my SymPy primer https://github.com/ejwest2/sandbox/blob/master/python/sympy-primer.ipynb.

4.2 Fuller Details

SymPy is an add-on package to Python for handling symbolic calculations - meant to be a fully featured computer algebra system (CAS) like Mathematica or Maple.

In [132…
```
# load all sympy commands all at once (not recommended)
# from sympy import *
```

In [133…
```
# introduce an alias for the numpy namespace (recommended)
import sympy as sym
```

In order to display output in an easy-to-read format, we need to add the following line.

In [134…
```
sym.init_printing()
```

---

2. Creating symbol objects

The basic elements used in sympy calculations are symbol objects, which are instances of the class sympy.Symbols. There are multiple ways to create new sympy.Symbols objects. (From now on, Symbols objects will simply be called symbols.)

Symbol( )

In [135…
```
# simplest way to create a symbol
x = sym.Symbol("x")
x
```

Out[135…
$$x$$

In [136…
```
# create a symbol with some specified attribute
alpha = sym.Symbol("alpha", real=True)
beta = sym.Symbol("beta", integer=True)
alpha, beta
```

Out[136…
$$(\alpha,\ \beta)$$

---

symbols( )

In [137…
```
# create multiple symbols all at once, using sympy.symbols()
alpha, beta, gamma = sym.symbols("alpha, beta, gamma", positive=True, real=True)
alpha, beta, gamma
```

Out[137…
$$(\alpha,\ \beta,\ \gamma)$$

In [138…
```
# the symbol name does not need to match the symbol
a, b, c = sym.symbols("alpha2, beta2, gamma2")
a, b, c
```

Out[138…
$$\left(\alpha_2,\ \beta_2,\ \gamma_2\right)$$

In [139…
```
# the symbol name does not need to be unique
a, b = sym.symbols("alpha3, alpha3")
a, b
```

Out[139…

$$\left(\alpha_3,\ \alpha_3\right)$$

var( )

In [140…
```
# create multiple symbols all at once, using sympy.var()
alpha1, beta1, gamma1 = sym.var("alpha1, beta1, gamma1", positive=True, real=True)
alpha1, beta1, gamma1
```

Out[140…

$$\left(\alpha_1,\ \beta_1,\ \gamma_1\right)$$

Symbol attributes

In [141…
```
alpha = sym.var("alpha", real=True, integer=False, positive=True)
alpha
```

Out[141…

$$\alpha$$

In [142…
```
alpha.is_real
```

Out[142… True

In [143…
```
alpha.is_complex
```

Out[143… True

In [144…
```
alpha.is_imaginary
```

Out[144… False

In [145…
```
alpha.is_positive
```

Out[145… True

In [146…
```
alpha.is_negative
```

Out[146… False

In [147…
```
alpha.is_zero
```

Out[147… False

In [148…
```
alpha.is_nonzero
```

Out[148… True

In [149…
```
alpha.is_even
```

Out[149… False

In [150…
```
alpha.is_odd
```

Out[150… False

In [151…
```
alpha.is_integer
```

Out[151… False

In [152…
```
alpha.is_prime
```

Out[152… False

In [153…
```
alpha.is_finite
```

Out[153… True

In [154…
```
alpha.is_infinite
```

Out[154… False

Numbers

SymPy provides its own integer and float classes, **sympy.Integer** and **sympy.Float**, respectively. These are distinct from the int and float types in Python. One of the main differences is that **sympy.Integer** and **sympy.Float** have arbitrary precision, whereas int and float do not.

In addition to integers and floats, sympy also provides a class for rational numbers, **sympy.Rational**.

Finally, sympy provides predefined symbols for various special constants used in mathematics, such as π and e.

Integer( )

In [155…
```
# this is a python integer
a = 3
type(a)
```

Out[155… int

In [156…
```
# this is a sympy integer
b = sym.Integer(3)
type(b)
```

Out[156… sympy.core.numbers.Integer

In [157…
```
# instances of sympy.Integer have the attribute Integer (capital "I"),
# as well as the attribute integer (lower case "i")
b = sym.Integer(8)
b.is_Integer, b.is_integer
```

Out[157… (True, True)

In [158…
```
# instances of sympy.Symbols with integer=True,
# have the attribute integer (lower case "i"),
# but do NOT have the attribute Integer (capital "I")
c = sym.symbols("c", integer=True)
c.is_integer, c.is_Integer
```

Out[158… (True, False)

Float( )

In [159…
```
# this is a python float
a = 3.0
type(a)
```

Out[159… float

In [160…
```
# this is a sympy float
b = sym.Float(3.0)
type(b)
```

Out[160… sympy.core.numbers.Float

In [161…
```
# this is a python float to 25 digit precision
a = "%.25f" % 0.3
a
```

Out[161… '0.2999999999999999888977698'

In [162…
```
# this is a sympy float with 25 digit precision
b = sym.Float('0.3', 25)
b
```

Out[162… 0.3

Rational( )

In [163…
```
# here are examples of sympy rationals
a = sym.Rational(2, 3)
b = sym.Rational(3, 2)
c = sym.Rational(3.001, 2)
a, b, c
```

Out[163… $\left(\frac{2}{3}, \frac{3}{2}, \frac{6757651240869429}{4503599627370496}\right)$

In [164…
```
# more examples of sympy rationals
d = sym.Rational('0.1')
e = sym.Rational('1.23')
f = sym.Rational('1e-4')
d, e, f
```

Out[164… $\left(\frac{1}{10}, \frac{123}{100}, \frac{1}{10000}\right)$

In [165…
```
# algebraic operations between sympy rationals preserve the type
a = sym.Rational(2,3)
b = sym.Rational(3,4)
a+b, a-b, a*b, a/b
```

Out[165… $\left(\frac{17}{12}, -\frac{1}{12}, \frac{1}{2}, \frac{8}{9}\right)$

Special symbols

In [166…
```
# predefined sympy symbol for pi
sym.pi
```

Out[166… π

In [167…
```
# predefined sympy symbol for e
sym.E
```

Out[167… e

In [168…
```
# predefined sympy symbol for Euler's constant
sym.EulerGamma
```

Out[168… γ

In [169…
```
# predefined sympy symbol for i
sym.I
```

Out[169… i

In [170…
```
# predefined sympy symbol for infinity
sym.oo
```

Out[170… ∞

4.3 Functions

Sympy objects that represent functions can be created with sympy.Function( ). User-defined functions can be created with sympy.Lambda( ). Sympy also has a wide variety of predefined functions, such as sympy.sin( ) and sympy.cos( ).

Elementary functions

In [171…
```
# square root
sym.sqrt(x + 1)
```

Out[171… $\sqrt{x + 1}$

In [172…
```
# sine
sym.sin(x)
```

Out[172… $\sin(x)$

In [173…
```
#cosine
sym.cos(x)
```

Out[173… $\cos(x)$

In [174…
```
# log
sym.log(1 - x)
```

Out[174… $\log(1 - x)$

In [175…
```
# exponential
sym.exp(x)
```

Out[175… e^x

Function( )

In [176…
```
# create an undefined function with an arbitrary number of input variables
f = sym.Function("f")
f
```

Out[176… f

In [177…
```
# create an undefined function of one variable
x = sym.symbols("x")
f = sym.Function("f")(x)
f
```

Out[177… $f(x)$

In [178…
```
# create an undefined function of two variables
x, y = sym.symbols("x, y")
f = sym.Function("f")(x, y)
f
```

Out[178… $f(x, y)$

Lambda( )

In [179…
```
# create a user-defined function
f = sym.Lambda(x, x**2)
f, f(x), f(3), f(y + 4)
```

Out[179… $\left(\left(x \mapsto x^2\right),\ x^2,\ 9,\ (y + 4)^2\right)$

In [180…
```
# it likes to combine all results into a single output line
f
```

Out[180…
$$\left(x \mapsto x^2\right)$$

why do we use lambda? -twd 2023-09-30

4.4 Manipulating Expressions

Expressions are formed by combining symbols and functions. There are numerous ways of manipulating expressions. Many of these are illustrated in the examples below.

In [181…
```
# create an expression
x, y = sym.symbols("x, y")
expr = 1 + 2*x*y + 3*x**2
expr
```

Out[181…
$$3x^2 + 2xy + 1$$

simplify( )

In [182…
```
# simplify an expression
x = sym.symbols("x")
expr1 = 2*(x**2 - x) - x*(x + 1)
expr2 = sym.simplify(expr1)
expr1, expr2
```

Out[182…
$$\left(2x^2 - x(x+1) - 2x,\ x(x-3)\right)$$

In [183…
```
expr1
```

Out[183…
$$2x^2 - x(x+1) - 2x$$

In [184…
```
# another way to simplify an expression
x = sym.symbols("x")
expr1 = 2*(x**2 - x) - x*(x + 1)
expr2 = expr1.simplify()
expr1, expr2
```

Out[184…
$$\left(2x^2 - x(x+1) - 2x,\ x(x-3)\right)$$

trigsimp( )

In [185…
```
# simplify an expression using trig identities
x = sym.symbols("x")
expr = 2*sym.cos(x)*sym.sin(x)
expr, expr.trigsimp()
```

Out[185…
$$(2\sin(x)\cos(x),\ \sin(2x))$$

powsimp( )

In [186…
```
# simplify an expression using laws of powers
x, y = sym.symbols("x, y")
expr = sym.exp(x)*sym.exp(y)
expr, expr.powsimp()
```

Out[186…
$$\left(e^x e^y,\ e^{x+y}\right)$$

ratsimp( )

In [187…
```
# simplify an expression by combining common denominator
x, y = sym.symbols("x, y")
expr = 1/x + 1/y
expr, expr.ratsimp()
```

Out[187…
$$\left(\frac{1}{y} + \frac{1}{x},\ \frac{x+y}{xy}\right)$$

expand( )

In [188…
```
# expand an expression
x = sym.symbols("x")
expr = (1 + x)*(2 + x)
sym.expand(expr)
```

Out[188…
$$x^2 + 3x + 2$$

In [189…
```
expr #which came from this dude
```

Out[189…
$$(x + 1)(x + 2)$$

In [190…
```
# another way to expand an expression
x = sym.symbols("x")
expr = (1 + x)*(2 + x)
expr.expand()
```

Out[190…
$$x^2 + 3x + 2$$

In [191…
```
# expand a trig expression
x, y = sym.symbols("x, y")
expr = sym.sin(x + y)
expr.expand(trig=True)
```

Out[191…
$$\sin(x)\cos(y) + \sin(y)\cos(x)$$

In [192…
```
# expand a log expression
x, y = sym.symbols("x, y",positive=True)
expr = sym.log(x*y)
expr, expr.expand(log=True)
```

Out[192…
$$(\log(xy),\ \log(x) + \log(y))$$

In [193…
```
# separate a complex expression into real and imaginary parts
x, y = sym.symbols("x, y", real=True)
expr = sym.exp(x + y*sym.I)
expr, expr.expand(complex=True)
```

Out[193…
$$\left(e^{x+iy},\ ie^{x}\sin(y) + e^{x}\cos(y)\right)$$

In [194…
```
# expand the base of a power expression
a, b = sym.symbols("a, b", positive=True)
x = sym.symbols("x")
expr = (a*b)**x
expr, expr.expand(power_base=True)
```

Out[194…
$$\left((ab)^{x},\ a^{x}b^{x}\right)$$

In [195…
```
# expand the exponent of a power expression
a, b = sym.symbols("a, b", positive=True)
x = sym.symbols("x")
expr = sym.exp((a - b)*x)
expr, expr.expand(power_exp=True)
```

Out[195…
$$\left(e^{x(a-b)},\ e^{ax}e^{-bx}\right)$$

factor( )

In [196…
```
# factor an expression
x = sym.symbols("x")
expr = x**2 -1
expr, expr.factor()
```

Out[196…
$$\left(x^2 - 1,\ (x - 1)(x + 1)\right)$$

logcombine( )

In [197…
```
# combine a log expression
x, y = sym.symbols("x, y",positive=True)
expr = sym.log(x) + sym.log(y)
sym.logcombine(expr)
```

Out[197…
$$\log(xy)$$

collect( )

In [198…
```
# collecting terms containing a given symbol
x, y, z = sym.symbols("x, y, z")
expr = x + y + x*y*z
expr, expr.collect(x), expr.collect(y)
```

Out[198…
$$(xyz + x + y,\ x(yz + 1) + y,\ x + y(xz + 1))$$

apart( )

In [199…
```
# break a fraction into a partial fraction
x = sym.symbols("x")
expr = 1/(x**2 + 3*x + 2)
expr, expr.apart(x)
```

Out[199…

$$\left(\frac{1}{x^2+3x+2},\ -\frac{1}{x+2}+\frac{1}{x+1}\right)$$

together( )

In [200…
```
# combine partial fractions into a single fraction
x = sym.symbols("x")
expr = 1/(x + 1) - 1/(x + 2)
expr, expr.together()
```

Out[200…

$$\left(-\frac{1}{x+2}+\frac{1}{x+1},\ \frac{1}{(x+1)(x+2)}\right)$$

cancel( )

In [201…
```
# break a fraction into a partial fraction
x, y = sym.symbols("x, y")
expr = y/(y + y**2)
expr, expr.cancel(y)
```

Out[201…

$$\left(\frac{y}{y^2+y},\ \frac{1}{y+1}\right)$$

subs( )

In [202…
```
# substitute one symbol for another in an expression
x, y = sym.symbols("x, y")
expr = x + 2
expr, expr.subs(x,y)
```

Out[202…

$(x+2,\ y+2)$

In [203…
```
# make multiple symbol substitutions in an expression
x, y, z = sym.symbols("x, y, z")
a, b, c = sym.symbols("a, b, c")
expr = sym.sin(x) + sym.exp(y) + z
expr, expr.subs({x:a, y:b, z:c})
```

Out[203…

$$\left(z+e^{y}+\sin(x),\ c+e^{b}+\sin(a)\right)$$

In [204…
```
# make numeric substitutions in an expression
x = sym.symbols("x")
a, b, c = sym.symbols("a, b, c")
expr = a + b*x + c*x**2
params = {a:1.2, b:2, c:-5}
expr, expr.subs(params)
```

Out[204…

$$\left(a+bx+cx^{2},\ -5x^{2}+2x+1.2\right)$$

In [205…
```
# substitute expressions in an expression
x, y, z = sym.symbols("x, y, z")
expr = sym.sin(x) + sym.exp(y) + z
expr, expr.subs({sym.sin(x): x**2, sym.exp: sym.cos, z: sym.log(x)})
```

Out[205…

$$\left(z+e^{y}+\sin(x),\ x^{2}+\log(x)+\cos(y)\right)$$

4.5 Numerical evaluation

N( )

In [206…
```
# evaluate pi
pi = sym.pi
sym.N(pi)
```

Out[206…

3.14159265358979

In [207…
```
# evaluate pi to 50 digits
pi = sym.pi
sym.N(pi, 50)
```

Out[207…

3.1415926535897932384626433832795028841971693993751

In [208…
```
# evaluate pi in an expression
x = sym.symbols("x")
pi = sym.pi
expr = x + pi
sym.N(expr)
```

Out[208… $x + 3.14159265358979$

In [209…
```
# evaluate a user-defined function
f = sym.Lambda(x, expr)
f, f(x), f(2),  sym.N(f(2))
```

Out[209… $((x \mapsto x + \pi),\ x + \pi,\ 2 + \pi,\ 5.14159265358979)$

evalf( )

In [210…
```
# evaluate pi
pi = sym.pi
pi.evalf()
```

Out[210… 3.14159265358979

In [211…
```
# evaluate pi to 50 digits
pi = sym.pi
pi.evalf(50)
```

Out[211… 3.14159265358979323846264338327950288419716939937751

In [212…
```
# evaluate pi in an expression
x = sym.symbols("x")
pi = sym.pi
expr = x + pi
expr.evalf()
```

Out[212… $x + 3.14159265358979$

In [213…
```
# evaluate a user-defined function
f = sym.Lambda(x, expr)
f, f(x), f(2),  f(2).evalf()
```

Out[213… $((x \mapsto x + \pi),\ x + \pi,\ 2 + \pi,\ 5.14159265358979)$

lambdify( )

In [214…
```
# make a user-defined scalar function that will return numerical evaluations
x = sym.symbols("x")
pi = sym.pi
expr = x + pi
g = sym.lambdify(x, expr)
g, g(x), g(2)
```

Out[214…
```
(<function _lambdifygenerated(x)>, x + 3.14159265358979, 5.141592653589793)
```

note - above - you can ask it to give back the original function name, the original functions expressed as a function of its variables, and the literal output of the function. You can process something symbolically and analytrically with numbers at the same time. COOL BEANS!

In [215…
```
# make a user-defined vectorized function that will return numerical evaluations
x = sym.symbols("x")
pi = sym.pi
expr = x + pi

f = sym.lambdify(x, expr, 'numpy')
import numpy as np

xvals = np.arange(0,10)

f(xvals)
```

Out[215…
```
array([ 3.14159265,  4.14159265,  5.14159265,  6.14159265,  7.14159265,
        8.14159265,  9.14159265, 10.14159265, 11.14159265, 12.14159265])
```

4.6 Calculus Expressions

---

diff( )

In [216…
```
# oridinary derivative of an undefined function in one variable
x = sym.symbols("x")
f = sym.Function("f")(x)
sym.diff(f, x)
```

Out[216… $\frac{d}{dx} f(x)$

In [217…
```
# another way of taking a derivative
x = sym.symbols("x")
f = sym.Function("f")(x)
f.diff()
```

Out[217…

$$\frac{d}{dx}f(x)$$

In [218…

```
# higher-order derivatives
x = sym.symbols("x")
f = sym.Function("f")(x)
f.diff(x, 2), f.diff(x, 3), f.diff(x, 6)
```

Out[218…

$$\left(\frac{d^2}{dx^2}f(x),\ \frac{d^3}{dx^3}f(x),\ \frac{d^6}{dx^6}f(x)\right)$$

In [219…

```
# partial derivatives
x, y = sym.symbols("x, y")
g = sym.Function("g")(x, y)
g.diff(x), g.diff(y)
```

Out[219…

$$\left(\frac{\partial}{\partial x}g(x,y),\ \frac{\partial}{\partial y}g(x,y)\right)$$

In [220…

```
# second-order partial derivatives
x, y = sym.symbols("x, y")
g = sym.Function("g")(x, y)
g.diff(x, 2), g.diff(x,y), g.diff(y,x), g.diff(y,2)
```

Out[220…

$$\left(\frac{\partial^2}{\partial x^2}g(x,y),\ \frac{\partial^2}{\partial y\partial x}g(x,y),\ \frac{\partial^2}{\partial y\partial x}g(x,y),\ \frac{\partial^2}{\partial y^2}g(x,y)\right)$$

In [221…

```
# higher-order partial derivatives
x, y = sym.symbols("x, y")
g = sym.Function("g")(x, y)
g.diff(x, 3), g.diff(x, 2, y), g.diff(x, y, 2), g.diff(y, 3)
```

Out[221…

$$\left(\frac{\partial^3}{\partial x^3}g(x,y),\ \frac{\partial^3}{\partial y\partial x^2}g(x,y),\ \frac{\partial^3}{\partial y^2\partial x}g(x,y),\ \frac{\partial^3}{\partial y^3}g(x,y)\right)$$

Integrate( )

In [222…

```
# indefinite integral of an undefined function in one variable
x = sym.symbols("x")
f = sym.Function("f")(x)
sym.integrate(f)
```

Out[222…

$$\int f(x)\,dx$$

In [223…

```
# another way to take an indefinite integral
x = sym.symbols("x")
f = sym.Function("f")(x)
f.integrate()
```

Out[223…

$$\int f(x)\,dx$$

In [224…

```
# definite integral of an undefined function in one variable
x, a, b = sym.symbols("x, a, b")
f = sym.Function("f")(x)
f.integrate((x,a,b))
```

Out[224…

$$\int_a^b f(x)\,dx$$

In [225…

```
# indefinite multiple integral
x, y = sym.symbols("x, y")
f = sym.Function("f")(x, y)
f.integrate(x, y)
```

Out[225…

$$\iint f(x,y)\,dx\,dy$$

In [226…

```
# definite multiple integral
x, y, a, b, c, d = sym.symbols("x, y, a, b, c, d")
f = sym.Function("f")(x, y)
f.integrate((x, a, b), (y, c, d))
```

Out[226…

$$\int_c^d\int_a^b f(x,y)\,dx\,dy$$

In [227…

```
# infinite bounds of integration
oo = sym.oo
x, y = sym.symbols("x, y")
f = sym.Function("f")(x, y)
f.integrate((x, -oo, oo), (y, -oo, oo))
```

Out[227…
$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\,dx\,dy$$

In [228…
```
# integrate an expression, indefinite integral
x = sym.symbols("x")
expr = x**2 + sym.sin(x)
expr.integrate()
```

Out[228…
$$\frac{x^3}{3} - \cos(x)$$

In [229…
```
expr #it was integrated from this
```

Out[229…
$$x^2 + \sin(x)$$

In [230…
```
# integrate an expression, definite integral
x = sym.symbols("x")
expr = x**2 + sym.sin(x)
expr
```

Out[230…
$$x^2 + \sin(x)$$

In [231…
```
expr.integrate(x)
```

Out[231…
$$\frac{x^3}{3} - \cos(x)$$

In [232…
```
expr.integrate((x,0,1))
```

Out[232…
$$\frac{4}{3} - \cos(1)$$

Series( )

In [233…
```
# create a series expansion of an undefined function
x = sym.symbols("x")
f = sym.Function("f")(x)
sym.series(f, x)
```

Out[233…
$$f(0) + x\frac{d}{d\xi}f(\xi)\Big|_{\xi=0} + \frac{x^2\frac{d^2}{d\xi^2}f(\xi)\Big|_{\xi=0}}{2} + \frac{x^3\frac{d^3}{d\xi^3}f(\xi)\Big|_{\xi=0}}{6} + \frac{x^4\frac{d^4}{d\xi^4}f(\xi)\Big|_{\xi=0}}{24} + \frac{x^5\frac{d^5}{d\xi^5}f(\xi)\Big|_{\xi=0}}{120} + O\left(x^6\right)$$

In [234…
```
# another way to create a series expansion of an undefined function
x = sym.symbols("x")
f = sym.Function("f")(x)
f.series(x)
```

Out[234…
$$f(0) + x\frac{d}{d\xi}f(\xi)\Big|_{\xi=0} + \frac{x^2\frac{d^2}{d\xi^2}f(\xi)\Big|_{\xi=0}}{2} + \frac{x^3\frac{d^3}{d\xi^3}f(\xi)\Big|_{\xi=0}}{6} + \frac{x^4\frac{d^4}{d\xi^4}f(\xi)\Big|_{\xi=0}}{24} + \frac{x^5\frac{d^5}{d\xi^5}f(\xi)\Big|_{\xi=0}}{120} + O\left(x^6\right)$$

In [235…
```
# series expansion to order 2
x = sym.symbols("x")
f = sym.Function("f")(x)
f.series(x, n=2)
```

Out[235…
$$f(0) + x\frac{d}{d\xi}f(\xi)\Big|_{\xi=0} + O\left(x^2\right)$$

In [236…
```
# series expansion around point p
x, x0 = sym.symbols("x, x0")
f = sym.Function("f")(x)
f.series(x, x0, n=3)
```

Out[236…
$$f\left(x_0\right) + \left(x-x_0\right)\frac{d}{d\xi_1}f\left(\xi_1\right)\Big|_{\xi_1=x_0} + \frac{\left(x-x_0\right)^2\frac{d^2}{d\xi_1^2}f\left(\xi_1\right)\Big|_{\xi_1=x_0}}{2} + O\left(\left(x-x_0\right)^3; x\to x_0\right)$$

In [237…
```
# series with order object removed
x, x0 = sym.symbols("x, x0")
f = sym.Function("f")(x)
f.series(x, x0, n=3).removeO()
```

Out[237…

$$\frac{\left(x-x_0\right)^2 \frac{d^2}{d\xi_1^2} f\left(\xi_1\right)\Big|_{\xi_1=x_0}}{2} + \left(x-x_0\right)\frac{d}{d\xi_1} f\left(\xi_1\right)\Big|_{\xi_1=x_0} + f\left(x_0\right)$$

In [238…

```
# series expansion of sin(x) around x=0
x = sym.symbols("x")
f = sym.sin(x)
f.series(x, 0, n=9)
```

Out[238…

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + O\left(x^9\right)$$

In [239…

```
# series expansion of cos(x) around x=0
x = sym.symbols("x")
f = sym.cos(x)
f.series(x, 0, n=9)
```

Out[239…

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320} + O\left(x^9\right)$$

In [240…

```
# series expansion of exp(x) around x=0
x = sym.symbols("x")
f = sym.exp(x)
f.series(x, 0, n=9)
```

Out[240…

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + O\left(x^9\right)$$

In [241…

```
# binomial expansion
x = sym.symbols("x")
f = 1/(1 + x)**2
f.series(x, 0, n=9)
```

Out[241…

$$1 - 2x + 3x^2 - 4x^3 + 5x^4 - 6x^5 + 7x^6 - 8x^7 + 9x^8 + O\left(x^9\right)$$

---

limit( )

In [242…

```
# limit of a function
x = sym.symbols("x")
f = sym.sin(x)/x
f.limit(x, 0)
```

Out[242…

$$1$$

In [243…

```
# asymptotic limit of a function
oo = sym.oo
x = sym.symbols("x")
f = sym.sin(x)/x
f.limit(x, oo)
```

Out[243…

$$0$$

Derivative( )

In [244…

```
# a formal derivative, delaying its evaluation
x = sym.symbols("x")
f = sym.cos(x)
df = sym.Derivative(f, x)
ddf = sym.Derivative(f, x, 2)
dddf = sym.Derivative(f, x, 3)
f, df, ddf, dddf
```

Out[244…

$$\left(\cos(x),\ \frac{d}{dx}\cos(x),\ \frac{d^2}{dx^2}\cos(x),\ \frac{d^3}{dx^3}\cos(x)\right)$$

Integral( )

In [245…

```
# a formal integral
x = sym.symbols("x")
f = sym.cos(x)
myint = sym.Integral(f, x)
myint, f.integrate()
```

Out[245…

$$\left(\int \cos(x)\,dx,\ \sin(x)\right)$$

note: it is solved in this block, but in the previous code block, it shows the expressions of the functions without evaluating them

---

Sum( )

In [246…
```
# a formal sum, delaying its evaluation
oo = sym.oo
n = sym.symbols("n", integer=True)
mysum = sym.Sum(1/n**2, (n, 1, oo))
mysum
```

Out[246…
$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

how would you get the RESULT OF THIS?????

Product( )

In [247…
```
# a formal product, delaying its evaluation
n = sym.symbols("n", integer=True)
myprod = sym.Product(n, (n, 1, 7))
myprod
```

Out[247…
$$\prod_{n=1}^{7} n$$

4.7 Solving equation in symbolic form `doit( )`

expicitly evaluate!!!

In [248…
```
# try to explicitly evaluate a formal derivative
x = sym.symbols("x")
f = sym.cos(x)
df = sym.Derivative(f, x)
df, df.doit()
```

Out[248…
$$\left(\frac{d}{dx}\cos(x),\ -\sin(x)\right)$$

In [249…
```
# try to explicitly evaluate a formal integral
x = sym.symbols("x")
f = sym.cos(x)
myint = sym.Integral(f, x)
myint, myint.doit()
```

Out[249…
$$\left(\int \cos(x)\,dx,\ \sin(x)\right)$$

In [250…
```
# try to explicitly evaluate a formal sum
oo = sym.oo
n = sym.symbols("n", integer=True)
mysum = sym.Sum(1/n**2, (n, 1, oo))
mysum, mysum.doit()
```

Out[250…
$$\left(\sum_{n=1}^{\infty} \frac{1}{n^2},\ \frac{\pi^2}{6}\right)$$

In [251…
```
# try to explicitly evaluate a formal sum
n = sym.symbols("n", integer=True)
myprod = sym.Product(n, (n, 1, 7))
myprod, myprod.doit()
```

Out[251…
$$\left(\prod_{n=1}^{7} n,\ 5040\right)$$

solve( )

In [252…
```
# try to solve an equation in one unknown variable
x = sym.symbols("x")
expr = x**2 + 2*x - 3
sym.solve(expr)
```

Out[252…
$$[-3,\ 1]$$

In [253…
```
expr
```

Out[253…
$$x^2 + 2x - 3$$

In [254…
```
# try to solve an equation containing more than one symbol
x, a, b, c = sym.symbols("x, a, b, c")
expr = a*x**2 + b*x + c
sym.solve(expr, x)
```

Out[254…

$$\left[\frac{-b-\sqrt{-4ac+b^2}}{2a}, \frac{-b+\sqrt{-4ac+b^2}}{2a}\right]$$

In [255…
```
# polynomial with no complex solutions     huh????
x, a, b, c = sym.symbols("x, a, b, c")
expr = a*x**2 + b*x + c
vals = {a:1, b:-3, c:4}
sym.solve(expr.subs(vals), x)
```

Out[255…

$$\left[\frac{3}{2}-\frac{\sqrt{7}i}{2}, \frac{3}{2}+\frac{\sqrt{7}i}{2}\right]$$

In [256…
```
expr
```

Out[256…

$$ax^2+bx+c$$

In [257…
```
vals
```

Out[257…

$$\{a:1,\ b:-3,\ c:4\}$$

In [258…
```
# solve a system of equations for more than one unknown
x, y = sym.symbols("x, y")
expr1 = x**2 - y**2 - 4
expr2 = x**2 + y**2 - 4
solns = sym.solve([expr1, expr2],[x,y])
solns
```

Out[258…

$$[(-2,\ 0),\ (2,\ 0)]$$

$$x^2-y^2-4$$

$$x^2+y^2-4$$

In [259…
```
# solve, returning the solutions in dictionary form
x, y = sym.symbols("x, y")
expr1 = x**2 - y**2 - 4
expr2 = x**2 + y**2 - 4
solns = sym.solve([expr1, expr2],[x,y], dict=True)
solns
```

Out[259…

$$[\{x:-2,\ y:0\},\ \{x:2,\ y:0\}]$$

In [260…
```
# check solutions using subs()
x, y = sym.symbols("x, y")
expr1 = x**2 - y**2 - 4
expr2 = x**2 + y**2 - 4
solns = sym.solve([expr1, expr2],[x,y], dict=True)
[expr1.subs(soln) == 0 and expr2.subs(soln) == 0 for soln in solns]
```

Out[260…
```
[True, True]
```

4.8 Linear algebra

Matrix( )

In [261…
```
# a simple column matrix
sym.Matrix([1,2])
```

Out[261…

$$\begin{bmatrix}1\\2\end{bmatrix}$$

In [262…
```
# a simple row matrix
sym.Matrix([[1,2]])
```

Out[262…

$$\begin{bmatrix}1 & 2\end{bmatrix}$$

In [263…
```
# a simple 2 x 2 matrix
sym.Matrix([[1,2],[3,4]])
```

Out[263…

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$$

In [264…
```
# create a matrix using lambda function of rows and columns
sym.Matrix(3, 4, lambda m, n: 10*m + n)
```

Out[264…

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \end{bmatrix}$$

In [265…
```
# a symbolic matrix
a, b, c, d = sym.symbols("a, b, c, d")
M = sym.Matrix([[a,b],[c,d]])
M
```

Out[265…

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Matrix operations

In [266…
```
# matrix algebra
A = sym.Matrix([[1,2],[3,4]])
B = sym.Matrix([[0,1],[1,1]])
A+B, A-B, A*B, B*A
```

Out[266…

$$\left(\begin{bmatrix} 1 & 3 \\ 4 & 5 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}, \begin{bmatrix} 2 & 3 \\ 4 & 7 \end{bmatrix}, \begin{bmatrix} 3 & 4 \\ 4 & 6 \end{bmatrix}\right)$$

In [267…
```
# determinant
A = sym.Matrix([[1,2],[3,4]])
A.det()
```

Out[267…

$$-2$$

In [268…
```
# trace
A = sym.Matrix([[1,2],[3,4]])
A.trace()
```

Out[268…

$$5$$

In [269…
```
# transpose
A = sym.Matrix([[1,2],[3,4]])
A.transpose(), A.T
```

Out[269…

$$\left(\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}, \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\right)$$

In [270…
```
# adjoint
A = sym.Matrix([[1,2],[3,4]])
A.adjoint(), A.H
```

Out[270…

$$\left(\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}, \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\right)$$

In [271…
```
# inverse
A = sym.Matrix([[1,2],[3,4]])
A.inv()
```

Out[271…

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

In [272…
```
# norm
A = sym.Matrix([[1,2],[3,4]])
A.norm()
```

Out[272…

$$\sqrt{30}$$

In [273…
```
# rank
A = sym.Matrix([[1,2],[3,4]])
A.rank()
```

Out[273…

$$2$$

In [274…
```
# LU decomposition
A = sym.Matrix([[1,2],[3,4]])
A.LUdecomposition()
```

Out[274…

$$\left(\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix}, []\right)$$

In [275…
```
# QR decomposition
A = sym.Matrix([[1,2],[3,4]])
A.QRdecomposition()
```

Out[275…

$$\left(\begin{bmatrix} \frac{\sqrt{10}}{10} & \frac{3\sqrt{10}}{10} \\ \frac{3\sqrt{10}}{10} & -\frac{\sqrt{10}}{10} \end{bmatrix}, \begin{bmatrix} \sqrt{10} & \frac{7\sqrt{10}}{5} \\ 0 & \frac{\sqrt{10}}{5} \end{bmatrix}\right)$$

In [276…
```
# solve Ax = b
A = sym.Matrix([[1,2],[3,4]])
b = sym.Matrix([1,0])
A.solve(b)
```

Out[276…

$$\begin{bmatrix} -2 \\ \frac{3}{2} \end{bmatrix}$$

In [277…
```
# solve Ax = b, using LU factorization
A = sym.Matrix([[1,2],[3,4]])
b = sym.Matrix([1,0])
A.LUsolve(b)
```

Out[277…

$$\begin{bmatrix} -2 \\ \frac{3}{2} \end{bmatrix}$$

In [278…
```
# solve Ax = b, using QR factorization
A = sym.Matrix([[1,2],[3,4]])
b = sym.Matrix([1,0])
A.QRsolve(b)
```

Out[278…

$$\begin{bmatrix} -2 \\ \frac{3}{2} \end{bmatrix}$$

In [279…
```
# diagonalize
A = sym.Matrix([[1,2],[3,4]])
A.diagonalize()
```

Out[279…

$$\left(\begin{bmatrix} -\frac{\sqrt{33}}{6} - \frac{1}{2} & -\frac{1}{2} + \frac{\sqrt{33}}{6} \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} \frac{5}{2} - \frac{\sqrt{33}}{2} & 0 \\ 0 & \frac{5}{2} + \frac{\sqrt{33}}{2} \end{bmatrix}\right)$$

In [280…
```
# singular values
A = sym.Matrix([[1,2],[3,4]])
A.singular_values()
```

Out[280…

$$\left[\sqrt{\sqrt{221} + 15},\ \sqrt{15 - \sqrt{221}}\right]$$

In [ ]:

4.9 Find the Null Space of a matrix

`A * NullSpace = 0` https://www.geeksforgeeks.org/null-space-and-nullity-of-a-matrix/

In [281…
```
from sympy import Matrix

A = [[1, 2, 0], [2, 4, 0], [3, 6, 1]] # List A
A = Matrix(A) # Matrix A

# Null Space of A
NullSpace = A.nullspace() # Here NullSpace is a list
NullSpace = Matrix(NullSpace) # Here NullSpace is a Matrix
print("Null Space : ", NullSpace)
print(A * NullSpace) # checking whether NullSpace satisfies the given condition or not
```

```
Null Space :  Matrix([[-2], [1], [0]])
Matrix([[0], [0], [0]])
```

Fin the Nullity of a mnatrix using this pythong code:

In [282…
```
from sympy import Matrix

A = [[1, 2, 0], [2, 4, 0], [3, 6, 1]]
A = Matrix(A)

# Number of Columns
NoC = A.shape[1]

# Rank of A
rank = A.rank()
```

```
# Nullity of the Matrix
nullity = NoC - rank

print("Nullity : ", nullity)
```

```
Nullity :  1
```

---

using the Matrix.nullspace() method

In [283…

```
# import sympy
from sympy import *

M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
print("{}".format(M))

# Use sympy.nullspace() method
M_nullspace = M.nullspace()

print("Nullspace of a matrix : {}".format(M_nullspace))
```

```
Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
Nullspace of a matrix : [Matrix([
[  -1],
[-2/3],
[   1],
[   0]]), Matrix([
[  -3],
[-1/3],
[   0],
[   1]])]
```

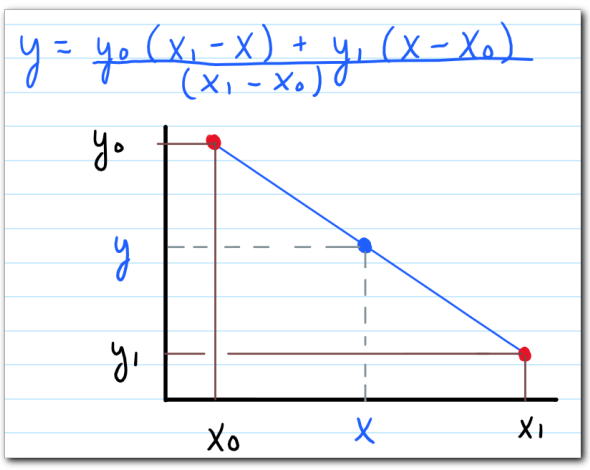
In [284…

```
# import sympy
from sympy import *
M = Matrix([[14, 0, 11, 3], [22, 23, 4, 7], [-12, -34, -3, -4]])
print("{}".format(M))

# Use sympy.nullspace() method
M_nullspace = M.nullspace()

print("{}".format(M_nullspace))
```

```
Matrix([[14, 0, 11, 3], [22, 23, 4, 7], [-12, -34, -3, -4]])
[Matrix([
[-1405/4254],
[   -10/709],
[  314/2127],
[         1]])]
```

5. SciPy

5.1 Linear Interpolation



Interpolate a 1-D function

In [285…

```
from scipy import interpolate

# Define the x and y coordinates of the two locations
x = [1, 2]  # point1(1,3) and point2(2,4)
y = [3, 4]

# Create a linear interpolation function
f = interpolate.interp1d(x, y)

# Now you can find the y value for any x within the range
print(f(1.5))  # This will output 3.5
```

```
3.5
```

5.2 Integrate a function

$$\int_0^5 x^3 dx \approx [\frac{1}{4}x^4]_0^5 = 156.25$$

In [286…
```
import numpy as np
import scipy
```

In [287…
```
a = 0         # lower limit
b = 5        # upper limit
n = 100         # number of divisions/subintervals
h = (b-a)/n   # step size

x = np.linspace(a,b,num=n+1)
y = x**3

Integral = scipy.integrate.simpson(y, x,dx=h)

print(Integral)
```

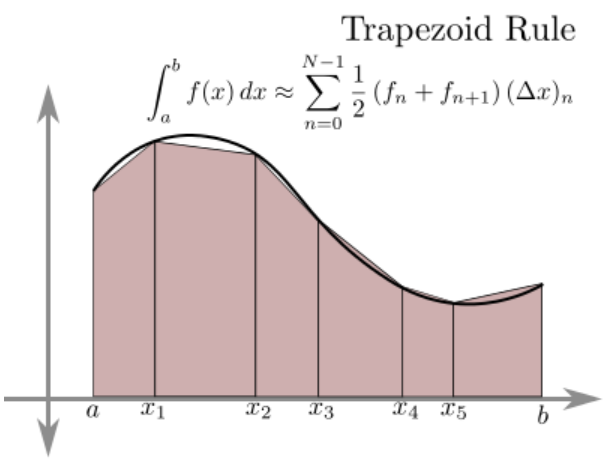
156.25

$$\int_0^{\pi} sin(x)\,dx \approx [-\sin x]_0^{\pi} = 2$$

In [288…
```
a = 0         # lower limit
b = np.pi       # upper limit
n = 10         # number of divisions/subintervals
h = (b-a)/n   # step size

x = np.linspace(a,b,num=n+1)
y = np.sin(x)

Integral = scipy.integrate.simpson(y, x,dx=h)

print(Integral)
```

2.0001095173150043

5.3 Numerical intergration (Trapezoid, Simpson)



$$\int_a^b f(x)dx \approx T_n = \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right],$$

$$\Delta x = h = \frac{b-a}{n}\ ,\ x_i = a + i\Delta x.$$

numpy.trapz (y, x=None, dx=1.0, axis=-1)

numpy.linspace (start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Reference Trapezoidal Rule Simpson's Rule

$$\text{Then } \int_a^b f(x)dx = \int_{x_0}^{x_0+nh} y\,dx$$

$$= h\left[\frac{1}{2}(y_0 + y_n) + (y_1 + y_2 + \ldots\ldots + y_{n-1})\right]$$

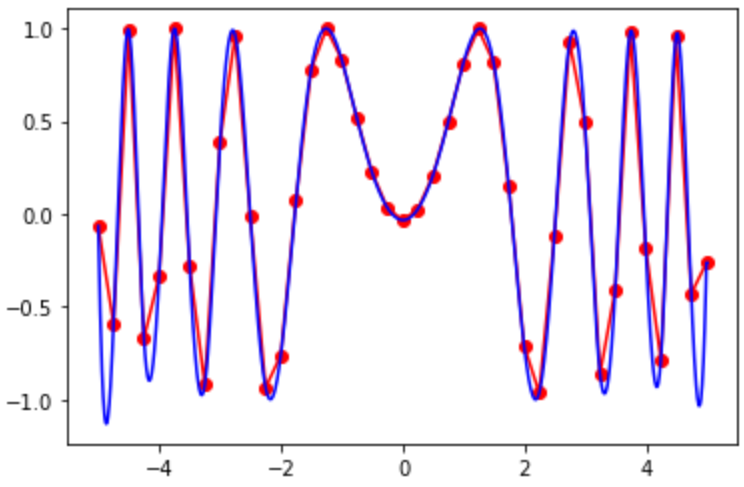
$$= \frac{h}{2}\left[(y_0 + y_n) + 2(y_1 + y_2 + \ldots\ldots + y_{n-1})\right]$$

(h is width of subinterval, n must be even)

In [289…
```
# !pip install scipy    #install if needed
```

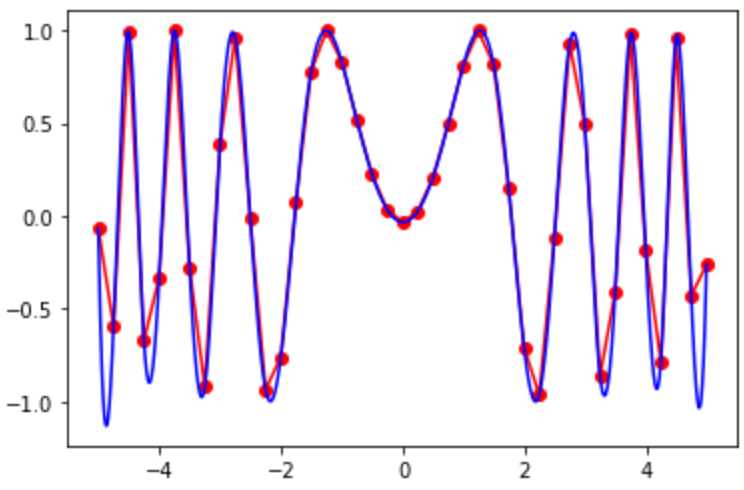
In [290…
```
%matplotlib inline
import numpy as np
import scipy as scipy
import scipy.interpolate
import matplotlib.pyplot as plt

x = np.arange(-5.01, 5.01, 0.25) #
y = np.arange(-5.01, 5.01, 0.25)
```

```
xx, yy = np.meshgrid(x, y)
z = np.sin(xx**2+yy**2)
f = scipy.interpolate.RectBivariateSpline(x, y, z)

xnew = np.arange(-5.01, 5.01, 1e-2)
ynew = np.arange(-5.01, 5.01, 1e-2)
znew = f(xnew, ynew)
plt.plot(x, z[0, :], 'ro-', xnew, znew[0, :], 'b-')
plt.show()

# NOT PLOTTING IN COLAB
```



In [291…

```
import numpy as np
import scipy as scipy
import scipy.interpolate
import matplotlib.pyplot as plt

x = np.arange(-5.01, 5.01, 0.25)
y = np.arange(-5.01, 5.01, 0.25)
xx, yy = np.meshgrid(x, y)
z = np.sin(xx**2+yy**2)
f = scipy.interpolate.RectBivariateSpline(x, y, z)

xnew = np.arange(-5.01, 5.01, 1e-2)
ynew = np.arange(-5.01, 5.01, 1e-2)
znew = f(xnew, ynew)
plt.plot(x, z[0, :], 'ro-', xnew, znew[0, :], 'b-')
plt.show()
```



It looks like you're trying to interpolate a 2D function and plot the result. Your code is correct and should work as expected. Here's what it does:

1. It creates a grid of points in the x-y plane using `numpy.meshgrid`. 2.At those points, it computes the values of the function

$$z = \sin(x^2 + y^2)$$

2. It creates an interpolating function using `scipy.interpolate.RectBivariateSpline`.
3. It evaluates this function at a new, finer grid of points.
4. Finally, it plots a slice at $y = 0$ of the original and interpolated function values along the line

The plot will show the original function values (in red) and the interpolated function values (in blue) along this line. The 'ro-' argument in the plot function call indicates that the original points should be plotted as red circles connected by lines, while 'b-' indicates that the interpolated points should be plotted as a blue line.

If you have any other questions or need further clarification, feel free to ask! 😊 -chatGPT

In [292…

```
from math import sin,pi
#TRAPEZOIDAL RULE
f = lambda x: x*sin(x)
a = 0          # lower limit
b = pi/2       # upper limit
n = 5          # number of divisions/subintervals
h = (b-a)/n    # step size

S = 0.5*(f(a)+f(b))
for i in range(1,n): #this loop will run until (n-1)
    S+=f(a+i*h)
integral = h*S

print('h = ',round(h,6))
print("Integral = {}".format(integral))
```

```
h =  0.314159
Integral = 1.0082654169662284
```

In [293…
```
#TRAPEZOIDAL RULE
a = 0          # lower limit
b = pi/2          # upper limit
n = 5          # number of divisions/subintervals
h = (b-a)/n    # step size

x = np.linspace(a,b,num=n+1)
y = x*np.sin(x)

trap = np.trapz(y,x,dx=h)
print('h= ',round(h,6))
print("Integral = {}".format(trap))
```

```
h=  0.314159
Integral = 1.0082654169662284
```

In [294…
```
f = lambda x: x**2
#TRAPEZOIDAL RULE
a = 0          # lower limit
b = 2          # upper limit
n = 3          # number of divisions/subintervals
h = (b-a)/n    # step size

S = 0.5*(f(a)+f(b))
for i in range(1,n): #this loop will run until (n-1)
    S+=f(a+i*h)

integral = h*S
print('h= ',round(h,6))
print("Integral = {}".format(integral))
```

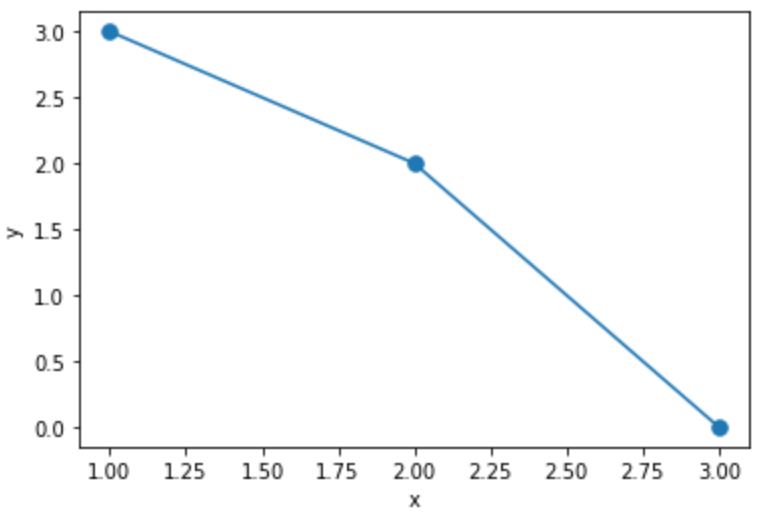
```
h=  0.666667
Integral = 2.8148148148148148
```

In [295…
```
#TRAPEZOIDAL RULE
a = 0          # lower limit
b = 2          # upper limit
n = 3          # number of divisions/subintervals
h = (b-a)/n    # step size

x = np.linspace(a,b,num=n+1)
y = x**2

trap = np.trapz(y,x,dx=h)
print('h= ',round(h,6))
print("Integral = {}".format(trap))
```

```
h=  0.666667
Integral = 2.8148148148148148
```

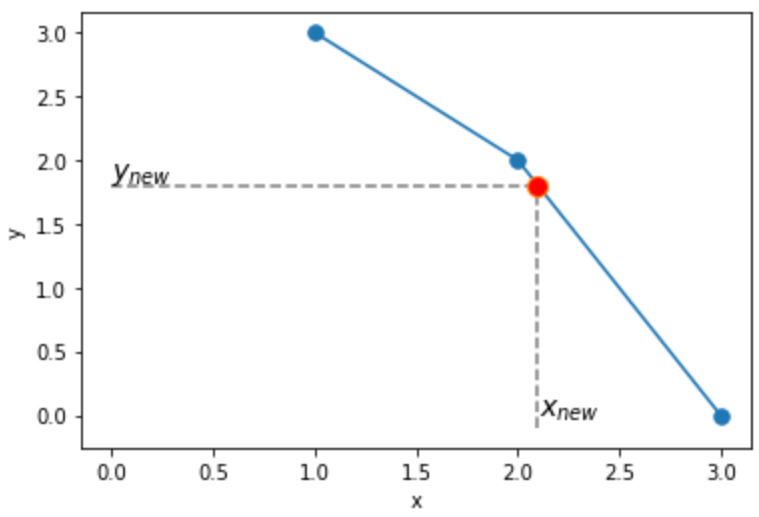
In [296…
```
# plot (1,3) (2,2) (3,0) points
x = [1,2,3]
y = [3,2,0]
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x,y,marker='.',ms = 15)
```

Out[296… `[<matplotlib.lines.Line2D at 0x1e6710f3a60>]`



In [298…
```
# plot (1,3) (2,2) (3,0) points
x = [1,2,3]
y = [3,2,0]

x_new = 2.1 # INPUT new value of x ,so we need to find new value of y
y_new = np.interp(x_new,x,y) # using NumPy library
print('Given x_new value : {}'.format(x_new))
print('New interpolated value (y_new): {}'.format(y_new))

plt.xlabel('x')
plt.ylabel('y')
plt.plot(x,y,marker='.',ms = 15)

plt.plot([x_new,x_new],[-0.1,y_new],'--',color="gray")
plt.plot([0,x_new],[y_new,y_new],'--',color="gray")
plt.plot(x_new,y_new,marker='o',ms = 10,mfc='red') # ms:markersize ,mfc:markerfacecolor
plt.text(x_new+0.01,0,r'$x_{new}$',fontsize=14)
plt.text(0,y_new+0.05,r'$y_{new}$',fontsize=14)
plt.show()
```

```
Given x_new value : 2.1
New interpolated value (y_new): 1.7999999999999998
```



In [299…

```
# plot (1,3) (2,2) (3,0) points
x = [1,2,3,4]
y = [3,2,0,4]

x_new = 2.1 # INPUT new value of x ,so we need to find new value of y
y_new = np.interp(x_new,x,y) # using NumPy library
print('Given x_new value : {}'.format(x_new))
print('New interpolated value (y_new): {}'.format(y_new))

plt.xlabel('x')
plt.ylabel('y')
plt.plot(x,y,marker='.',ms = 15)

plt.plot([x_new,x_new],[-0.1,y_new],'--',color="gray")
plt.plot([0,x_new],[y_new,y_new],'--',color="gray")
plt.plot(x_new,y_new,marker='o',ms = 10,mfc='red') # ms:markersize ,mfc:markerfacecolor
plt.text(x_new+0.01,0,r'$x_{new}$',fontsize=14)
plt.text(0,y_new+0.05,r'$y_{new}$',fontsize=14)
plt.show()
```

```
Given x_new value : 2.1
New interpolated value (y_new): 1.7999999999999998
```