SCIENTIFIC PROGRAMING IN PYTHON

Ari Maller

SCIENTIFIC PROGRAMING

- Python is used for many things, a very small fraction of users are using it for science.
- Importing various packages can make python a powerful tool for scientific programming. Some of the most import packages are:
  - Numpy - numerical python
  - Matplotlib - basic plotting
  - Scipy - scientific python, various functions not in numpy
  - Pandas - a data structure for tabular data
  - Seaborn - a fancy wrapper for Matplotlib
  - Sklearn - data science routines

NUMPY

- For scientific use one of the most important packages in python is the numpy (numerical python) package. This package creates a new data type called an array that is a sequence that behaves mathematically like a vector. Math operations generally work on the arrays element by element, which means all elements must be of the same type. Here we cover:
- array creation
- multidimensional arrays
- indexing and slicing
- reading/writing data
- On import numpy is often given the nickname np.

ARRAY CREATION

- There are many ways to create arrays. Lists can be converted to arrays, one can create arrays where all elements have the same value, or one can create arrays that increase value by a fixed amount (these are very useful for plotting and other calculations). You can specify the datatype of the elements with the dytpe key word.

```
a = np.array([1,2,3,4,5]) #create from list
ones = np.ones(5, dtype=np.bool) #fill with values
z = np.zeros(5, dtype=np.float) #dtype can set the data type
t = np.full(5,3) #an array of five 3s
x = np.arange(0,5,1) #step by 1
y = np.linspace(0,1,5) #make 5 elements
```

ARRAY OPERATIONS

```
print(a+5) #math on arrays works on each element

print(2*a)

print(a**2)

print(np.sqrt(a)) #numpy has functions also work in this way

print(np.cos(a))

print(a+t*y) #equal length arrays act element by element
```

MULTIDIMENSIONAL ARRAYS

- One can also create 2, 3 or any dimension arrays. This can be from a list of lists, but it is usually easier to create them using the filling functions, but to pass a list as the shape instead of just an int.

```
ones = np.ones([2,2], dtype=np.bool) #the shape can be a list
z = np.zeros((2,2,2), dtype=np.float) #it can also be a tuple
t = np.full([5,2],3) #an array of five 3s
```

ARRAY INFORMATION

- You can get information about at array with size, ndim and shape. You can rearrange the array into a new shape with the reshape() method or flatten it with flatten().

```
print(t.size,t.ndim,t.shape)

print(z.shape)

new_z=z.reshape([4,2]) #pass a list or tuple for the shape

print(new_z.shape)

print(z.flatten()) #same as reshape to 1D
```

INDEXING AND SLICING

- Just like other sequences the elements in arrays can be accessed by indexing and slicing. For 2+ dimensional arrays the index is a corresponding set of integers. Using colons of the form i:j:k creates a slice from i to j in steps of k. If k is omitted it steps by 1.

```
a = np.arange(0,9,1,dtype=np.int64)
b = a.reshape((3,3))
print(a[4],b[0,0],b[1,1])
print(b[1][1]) #you can also access like this, but it is inefficient
print(a[0:5],a[:5:2])
```

- A very powerful tool is to slice with a Truth array. This will only return the values that are True according to some condition

```
print(a[a > 5]) #only elements above 5
print(a[a % 2 == 0]) #only even elements
b = a[np.logical_and(a > 5, a % 2)] #elements above 5 and even
c = a[np.logical_or(a > 5, a % 2)] #above 5 or even
```

READING AND WRITING DATA

- You can read and write data in csv format (column data) from numpy using the loadtxt() and savetxt() functions.

```
data = np.loadtxt('hubble1929_table1.dat',comments='#')

obj,dis,vel = np.loadtxt('hubble1929_table1.dat',unpack=True)

np.savetxt('tmp.dat',data,fmt='%.3e') #fmt formats here 3 decimal places
```

- If you have missing data in your file you will have to use np.genfromtxt() instead. You can also read/write to other data formats like npy, hdf5 or netcdf.

VISUALIZATION

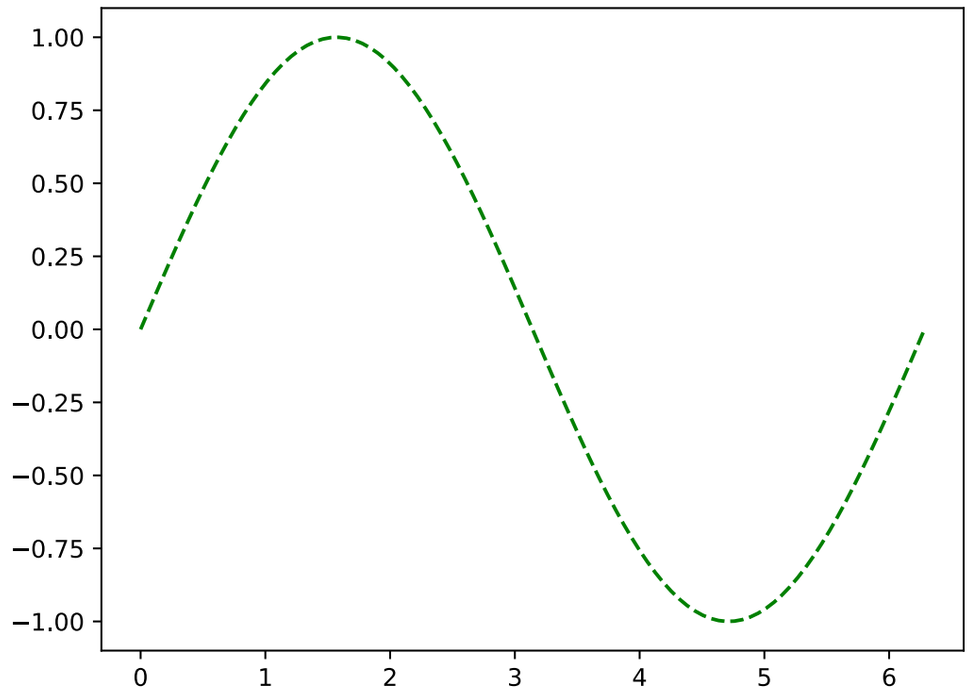
- Equally important for scientific analysis with python is a means to visualize one's data. There are many packages for visualization. One place to start is with matplotlib.
- One important thing to note is that in the notebook if the last line makes a plot it will be shown in the notebook, just like a variable. But for normal python the plt.show() function must be run for interactive plotting.
- Many of the plotting functions in matplotlib are found in the pyplot subpackage which is often given the nickname plt.

LINE PLOT

```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='dashed')
plt.show() #you could use plt.savefig(‘name.pdf’) instead
y2=np.cos(x)
y3=np.tan(x)
plt.plot(x,y2,label='cos') #labels can be used to make a legend
plt.plot(x,y3,label='tan')
plt.legend()
plt.xlabel('degrees')
plt.title('Trig Functions')
plt.ylim([-2,2])
plt.show()
```

LINE PLOT

```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='dash
plt.show() #you could use plt.savefig('na
y2=np.cos(x)
y3=np.tan(x)
plt.plot(x,y2,label='cos') #labels can be u
plt.plot(x,y3,label='tan')
plt.legend()
plt.xlabel('degrees')
plt.title('Trig Functions')
plt.ylim([-2,2])
plt.show()
```

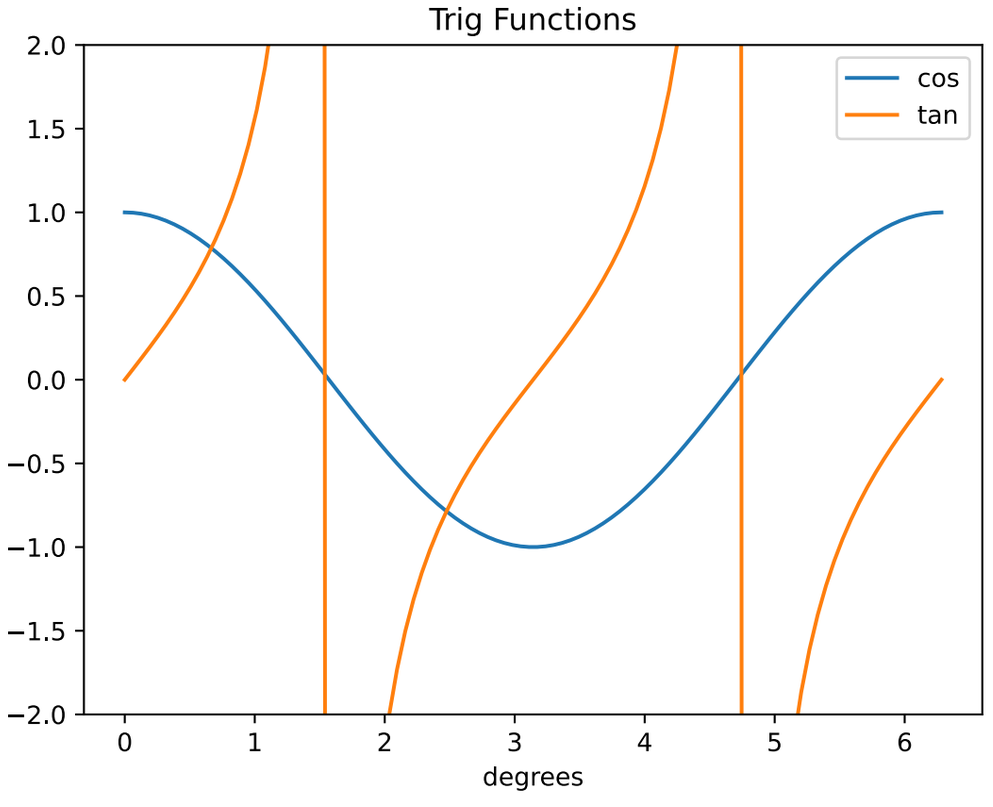


LINE PLOT

```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='dashed')
plt.show() #you could use plt.savefig(‘name.pdf’) instead
y2=np.cos(x)
y3=np.tan(x)
plt.plot(x,y2,label='cos') #labels can be used to make a legend
plt.plot(x,y3,label='tan')
plt.legend()
plt.xlabel('degrees')
plt.title('Trig Functions')
plt.ylim([-2,2])
plt.show()
```

LINE PLOT

```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='dashed')
plt.show() #you could use plt.savefig('name.pdf') instead
y2=np.cos(x)
y3=np.tan(x)
plt.plot(x,y2,label='cos') #labels can be u
plt.plot(x,y3,label='tan')
plt.legend()
plt.xlabel('degrees')
plt.title('Trig Functions')
plt.ylim([-2,2])
plt.show()
```



LINE PLOT

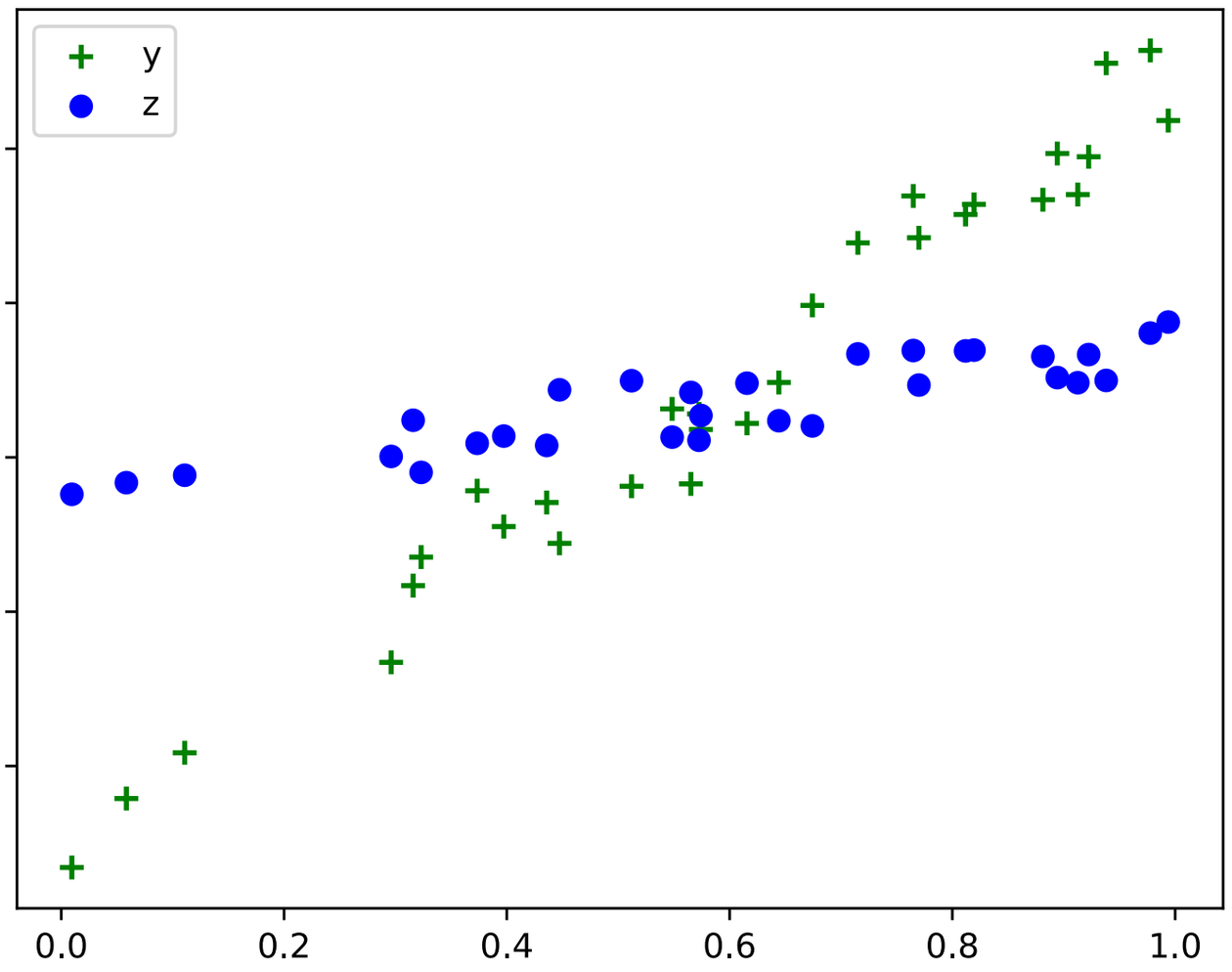
```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='dashed')
plt.show() #you could use plt.savefig(‘name.pdf’) instead
y2=np.cos(x)
y3=np.tan(x)
plt.plot(x,y2,label='cos') #labels can be used to make a legend
plt.plot(x,y3,label='tan')
plt.legend()
plt.xlabel('degrees')
plt.title('Trig Functions')
plt.ylim([-2,2])
plt.show()
```

SCATTER PLOTS

- The previous plots are plots of functions which is rarely what we have when doing scientific research. More often we have data which are discrete points and have a less clear relationship. In this case we should use a scatter plot to display them.
- x = np.random.random(30) #create some random data
- y = 0.5*x +0.1*np.random.random(30)
- z = 0.1*x + 0.25+0.05*np.random.random(30)
- plt.scatter(x,y,marker='+',c='green',label='y',s=50)
- plt.scatter(x,z,marker='o',c='blue',label='z')
- plt.legend()

SCATTER PLOTS

- The previous plots are plots of functions which is rarely what we have when doing scientific research. More often we have data which are discrete points and have a less clear relationship. In this case we should use a scatter plot to display them
- x = np.random.random
- y = 0.5*x +0.1*np.rand
- z = 0.1*x + 0.25+0.05
- plt.scatter(x,y,marker=
- plt.scatter(x,z,marker=
- plt.legend()



SCATTER PLOTS

- The previous plots are plots of functions which is rarely what we have when doing scientific research. More often we have data which are discrete points and have a less clear relationship. In this case we should use a scatter plot to display them.
- x = np.random.random(30) #create some random data
- y = 0.5*x +0.1*np.random.random(30)
- z = 0.1*x + 0.25+0.05*np.random.random(30)
- plt.scatter(x,y,marker='+',c='green',label='y',s=50)
- plt.scatter(x,z,marker='o',c='blue',label='z')
- plt.legend()

FITTING FUNCTIONS TO DATA

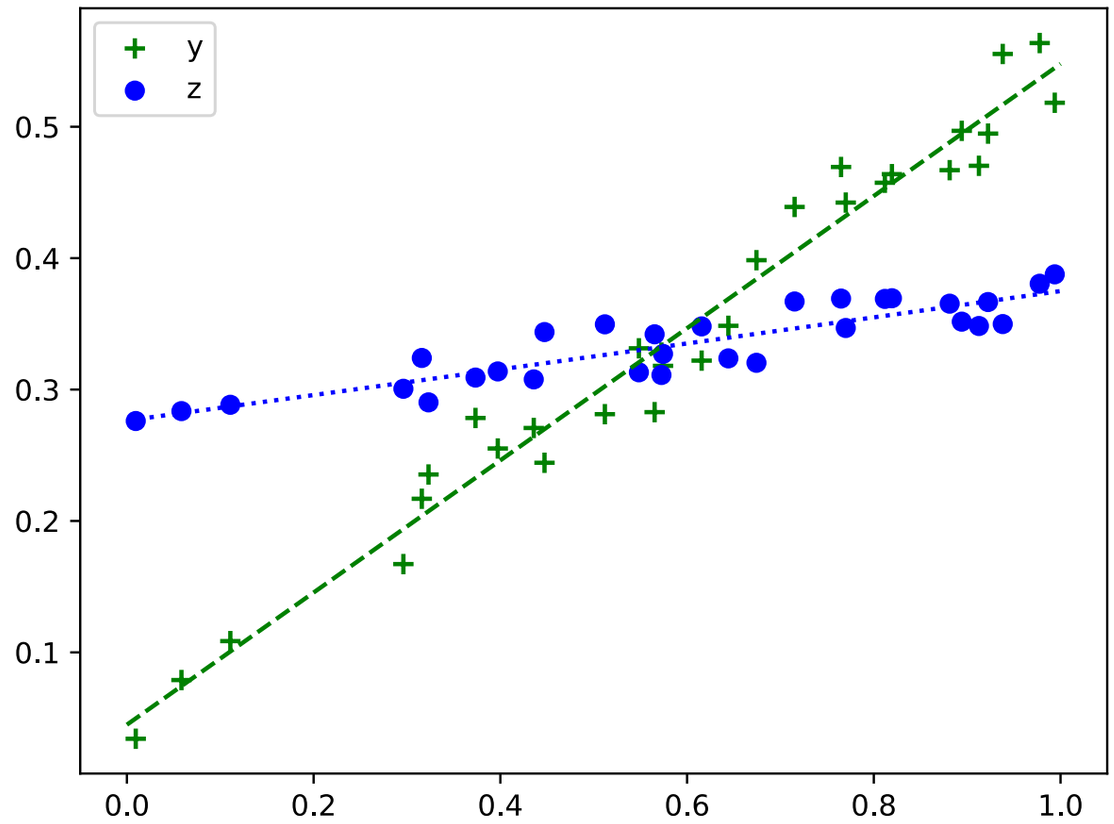
- One thing you may want to do is fit a function (like a line) to your data. There are 2 simple ways to do this. Numpy's polyfit() will fit a polynomial using least-squares, while curve_fit() from the scipy package will fit any function using a user specified function.

```
p1 = np.polyfit(x,y,1) #1 is the degree of the polynomial so a line (x^1)
p2 = np.polyfit(x,z,2)
xvals=np.linspace(0,1,10)
plt.plot(xvals,p1[1]+p1[0]*xvals,color='green',linestyle='dashed')
plt.plot(xvals,p2[2]+p2[1]*xvals+p2[0]*xvals**2,color='blue',linestyle='dotted')
plt.scatter(x,y,marker='+',c='green',label='y',s=50)
plt.scatter(x,z,marker='o',c='blue',label='z')
plt.legend()
plt.show()

import scipy.optimize as opt
def line(x,m,b):
    return m*x+b

popt,_ = opt.curve_fit(line,x,y)
plt.plot(xvals,line(xvals,popt[0],popt[1]),color='green',linestyle='dashed')
plt.scatter(x,y,marker='+',c='green',label='y')
```

FITTING FUNCTIONS TO DATA

- One thing you may want to do is fit a function (like a
  fit a polynomial using least-squares, while curve_fit()

```
p1 = np.polyfit(x,y,1) #1 is the degree of the polynomial
p2 = np.polyfit(x,z,2)
xvals=np.linspace(0,1,10)
plt.plot(xvals,p1[1]+p1[0]*xvals,color='green',linestyle=
plt.plot(xvals,p2[2]+p2[1]*xvals+p2[0]*xvals**2,color=
plt.scatter(x,y,marker='+',c='green',label='y',s=50)
plt.scatter(x,z,marker='o',c='blue',label='z')
plt.legend()
plt.show()
```




```
import scipy.optimize as opt
def line(x,m,b):
    return m*x+b

popt,_ = opt.curve_fit(line,x,y)
plt.plot(xvals,line(xvals,popt[0],popt[1]),color='green',linestyle='dashed')
plt.scatter(x,y,marker='+',c='green',label='y')
```

FITTING FUNCTIONS TO DATA

- One thing you may want to do is fit a function (like a line) to your data. There are 2 simple ways to do this. Numpy's polyfit() will fit a polynomial using least-squares, while curve_fit() from the scipy package will fit any function using a user specified function.

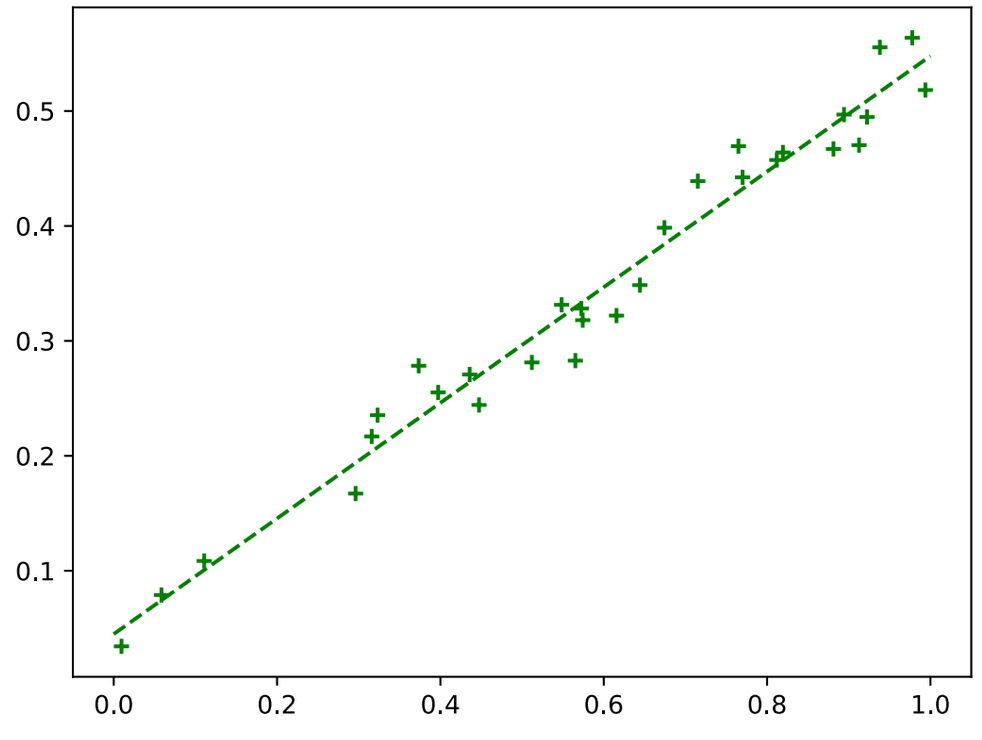
```
p1 = np.polyfit(x,y,1) #1 is the degree of the polynomial so a line (x^1)
p2 = np.polyfit(x,z,2)
xvals=np.linspace(0,1,10)
plt.plot(xvals,p1[1]+p1[0]*xvals,color='green',linestyle='dashed')
plt.plot(xvals,p2[2]+p2[1]*xvals+p2[0]*xvals**2,color='blue',linestyle='dotted')
plt.scatter(x,y,marker='+',c='green',label='y',s=50)
plt.scatter(x,z,marker='o',c='blue',label='z')
plt.legend()
plt.show()

import scipy.optimize as opt
def line(x,m,b):
    return m*x+b

popt,_ = opt.curve_fit(line,x,y)
plt.plot(xvals,line(xvals,popt[0],popt[1]),color='green',linestyle='dashed')
plt.scatter(x,y,marker='+',c='green',label='y')
```

FITTING FUNCTIONS TO DATA

- One thing you may want to do is fit a function (like a line) to your data. There are 2 simple ways to do this. Numpy's polyfit() will fit a polynomial using least-squares, while curve_fit() from the scipy package will fit any function using a user specified function.

```
p1 = np.polyfit(x,y,1) #1 is the degree of the polynomial so a line (x^1)
p2 = np.polyfit(x,z,2)
xvals=np.linspace(0,1,10)
plt.plot(xvals,p1[1]+p1[0]*xvals,color='green',linestyle='dashed')
plt.plot(xvals,p2[2]+p2[1]*xvals+p2[0]*xvals**2,color='blue',linestyle='dotted')
plt.scatter(x,y,marker='+',c='green',label='y',s=50)
plt.scatter(x,z,marker='o',c='blue',label='z')
plt.legend()
plt.show()
```

```
import scipy.optimize as opt
def line(x,m,b):
    return m*x+b

popt,_ = opt.curve_fit(line,x,y)
plt.plot(xvals,line(xvals,popt[0],popt[1]),color='green',linestyle=
plt.scatter(x,y,marker='+',c='green',label='y')
```



FITTING FUNCTIONS TO DATA

- One thing you may want to do is fit a function (like a line) to your data. There are 2 simple ways to do this. Numpy's polyfit() will fit a polynomial using least-squares, while curve_fit() from the scipy package will fit any function using a user specified function.

```
p1 = np.polyfit(x,y,1) #1 is the degree of the polynomial so a line (x^1)
p2 = np.polyfit(x,z,2)
xvals=np.linspace(0,1,10)
plt.plot(xvals,p1[1]+p1[0]*xvals,color='green',linestyle='dashed')
plt.plot(xvals,p2[2]+p2[1]*xvals+p2[0]*xvals**2,color='blue',linestyle='dotted')
plt.scatter(x,y,marker='+',c='green',label='y',s=50)
plt.scatter(x,z,marker='o',c='blue',label='z')
plt.legend()
plt.show()

import scipy.optimize as opt
def line(x,m,b):
    return m*x+b

popt,_ = opt.curve_fit(line,x,y)
plt.plot(xvals,line(xvals,popt[0],popt[1]),color='green',linestyle='dashed')
plt.scatter(x,y,marker='+',c='green',label='y')
```

FIGURE AND AXIS

- The examples of plotting we have just done can give a misleading sense of how matplotlib works, because we have been avoiding the figure and axis classes. However, these objects are the basis of plotting in matplotlib and for more advance plots you will need to understand them. The figure is just everything in the plotting window while the axis refers to the plotting axes and everything that goes with them. This becomes important when you want to have more than one axis on a figure. This is easiest to do with the subplots() function though there are many ways to create figures and axis.

```
fig,axes=plt.subplots(nrows=1,ncols=2)
axes[0].scatter(x,y)
axes[1].scatter(x,z)
plt.show() #and we get 2 plots on the same figure
```

FIGURE AND AXIS

- The examples of plotting we have just done can give a misleading sense of how matplotlib works, because we have been avoiding the figure and axis classes. However, these objects are the basis of plotting in matplotlib and for more advance plots you will need to understand them. The figure is just everything in the plotting window while the axis refers to the plotting axes and everything that goes with them. This becomes important when you want to have more than one axis on a figure. This is easi
  function though there are many ways

```
fig,axes=plt.subplots(nrows=1,ncols=
axes[0].scatter(x,y)
axes[1].scatter(x,z)
plt.show() #and we get 2 plots on the s
```

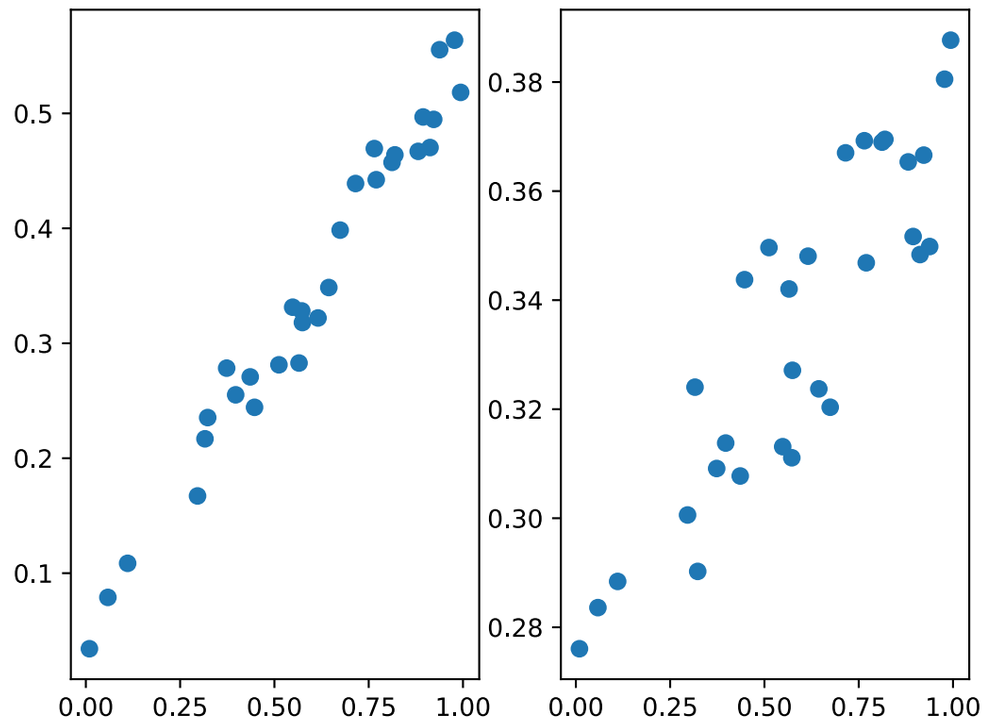


FIGURE AND AXIS

- The examples of plotting we have just done can give a misleading sense of how matplotlib works, because we have been avoiding the figure and axis classes. However, these objects are the basis of plotting in matplotlib and for more advance plots you will need to understand them. The figure is just everything in the plotting window while the axis refers to the plotting axes and everything that goes with them. This becomes important when you want to have more than one axis on a figure. This is easiest to do with the subplots() function though there are many ways to create figures and axis.

```
fig,axes=plt.subplots(nrows=1,ncols=2)
axes[0].scatter(x,y)
axes[1].scatter(x,z)
plt.show() #and we get 2 plots on the same figure
```

FIGURE AND AXIS

- We see that scatter() like almost all of the functions we have previously used is really a method associated with an axis. We can add legends, set limits, and add labels just like we've done above for each axis.

- You'll notice in the above plot that the y-axis range on the 2 axes is very different. We can force it to be the same with the shareY keyword.

```
fig,axis=plt.subplots(nrows=1,ncols=2,shareY=True,figsize=(6,4))
plt.subplots_adjust(hspace=0.0,wspace=0.0) #make no space between plots
axis[0].scatter(x,y)
axis[0].set_xlabel('x-stuff1')
axis[0].set_xlim([0.,0.95]) #if it goes to 1 the tickmarks overlap
axis[1].scatter(x,z)
axis[1].set_xlabel('x-stuff2')
```

FIGURE AND AXIS

- We see that scatter() like almost all of the functions we have previously used is really a method associated with an axis. We can add legends, set limits, and add labels just like we've done above for each axis.

- You'll notice in the above plot that the y-axis range on the 2 axes is very different. We can force it to be the

```
fig,axis=plt.subplots(nrows=1,ncols
plt.subplots_adjust(hspace=0.0,wspa
axis[0].scatter(x,y)
axis[0].set_xlabel('x-stuff1')
axis[0].set_xlim([0.,0.95]) #if it goes
axis[1].scatter(x,z)
axis[1].set_xlabel('x-stuff2')
```

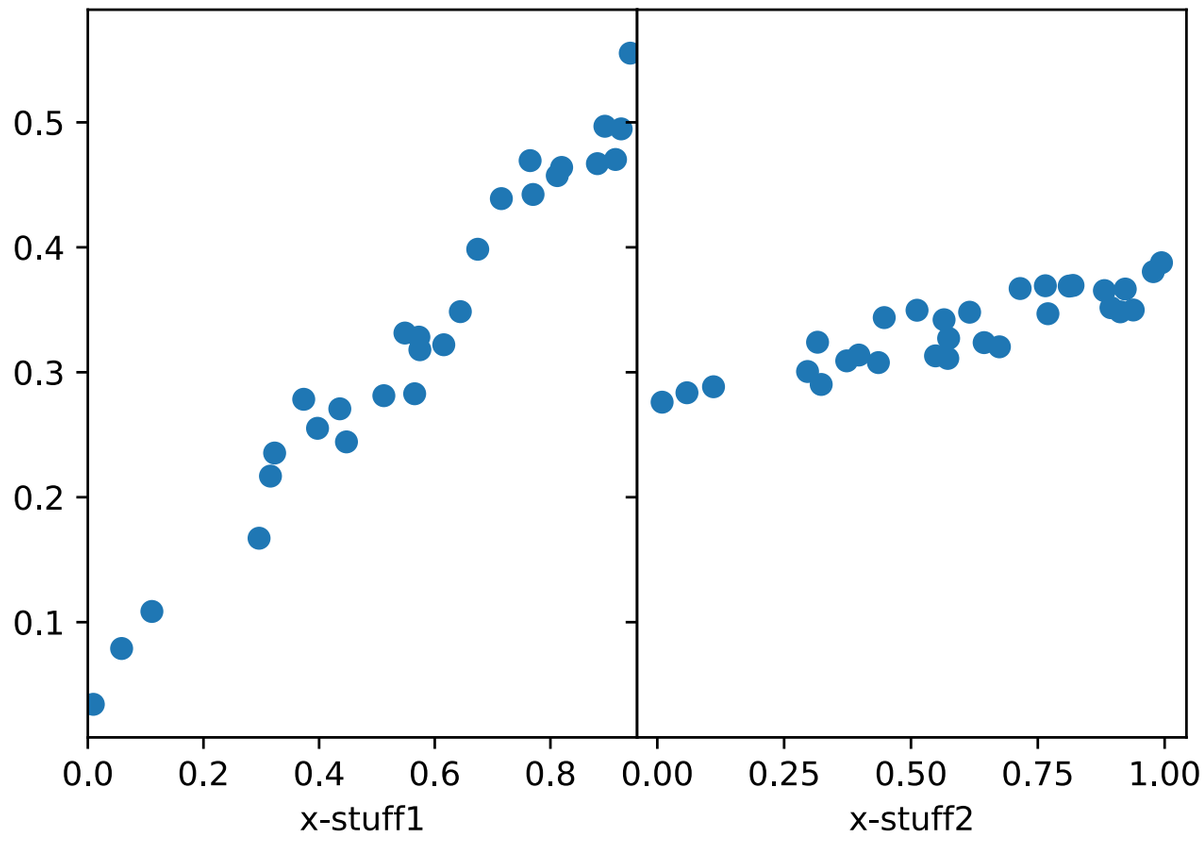


FIGURE AND AXIS

- We see that scatter() like almost all of the functions we have previously used is really a method associated with an axis. We can add legends, set limits, and add labels just like we've done above for each axis.

- You'll notice in the above plot that the y-axis range on the 2 axes is very different. We can force it to be the same with the shareY keyword.

```
fig,axis=plt.subplots(nrows=1,ncols=2,shareY=True,figsize=(6,4))
plt.subplots_adjust(hspace=0.0,wspace=0.0) #make no space between plots
axis[0].scatter(x,y)
axis[0].set_xlabel('x-stuff1')
axis[0].set_xlim([0.,0.95]) #if it goes to 1 the tickmarks overlap
axis[1].scatter(x,z)
axis[1].set_xlabel('x-stuff2')
```

EXERCISE 1

- Read in the data from the hubble1929_table1.dat file and plot velocity versus distance. Fit a line to the points. In a 2nd plot on the same figure plot the residuals (the difference between the points and the fit line) as a function of distance.

HISTOGRAM

- Another useful type of plot is the histogram. This can be made directly in matplotlib, or one can do the calculation in numpy and then plot the results with matplotlib.

```
h,xmids,_ = plt.hist(x)

h2,xedges = np.histogram(x, bins=10, range=[0,1])
xmids=0.5*(xedges[0:-1]+xedges[1:])
plt.bar(xmids,h2,width=0.1)
plt.xlabel(r'$\beta^2$') #you can use latex in labels
```

2D PLOTTING

- 2D plotting can be of 2 types. Either you have regularly gridded values of x and y with z values or irregular values. If you have irregular values they will have to be regularized in some way.
- There are many ways to display the 3rd dimension in your plot, brightness, color, contour lines, etc. What works best will depend on the values and taste.

2D HISTOGRAMS

- We can create regularly gridded data by making a histogram of a 2d distribution.

```
N=100000
x=np.random.normal(size=N)
y=np.random.normal(size=N)
hist,xedges,yedges=np.histogram2d(x,y,bins=25, range=[[-2,2],
[-2,2]]))
xmids=0.5*(xedges2[1:]+xedges[0:-1])
ymids=0.5*(yedges2[1:]+yedges[0:-1])
print(hist.shape)
```

- hist is now a 2d array of values which can be visualized without the x and y information.

IMSHOW

➤ Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]]) # change colormap
plt.colorbar() # add a color bar
```

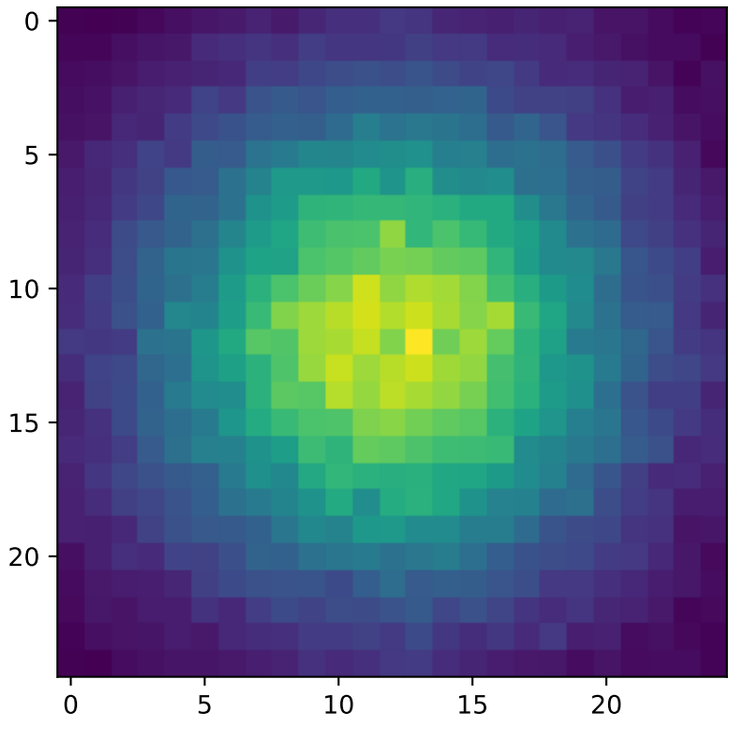
IMSHOW

- Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedg
```

e colormap

· bar



IMSHOW

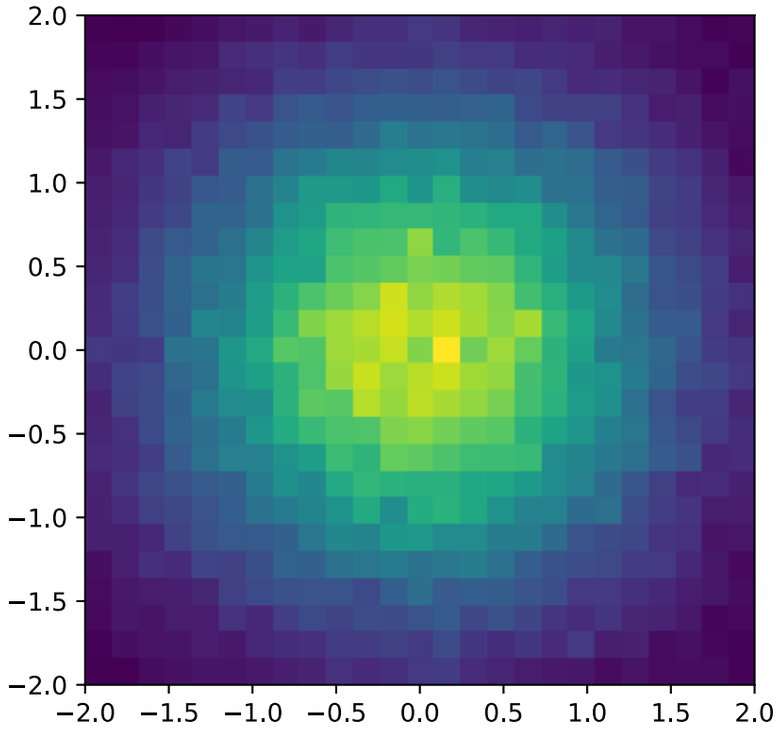
➤ Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]]) # change colormap
plt.colorbar() # add a color bar
```

IMSHOW

➤ Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[·
plt.colorbar()
```



IMSHOW

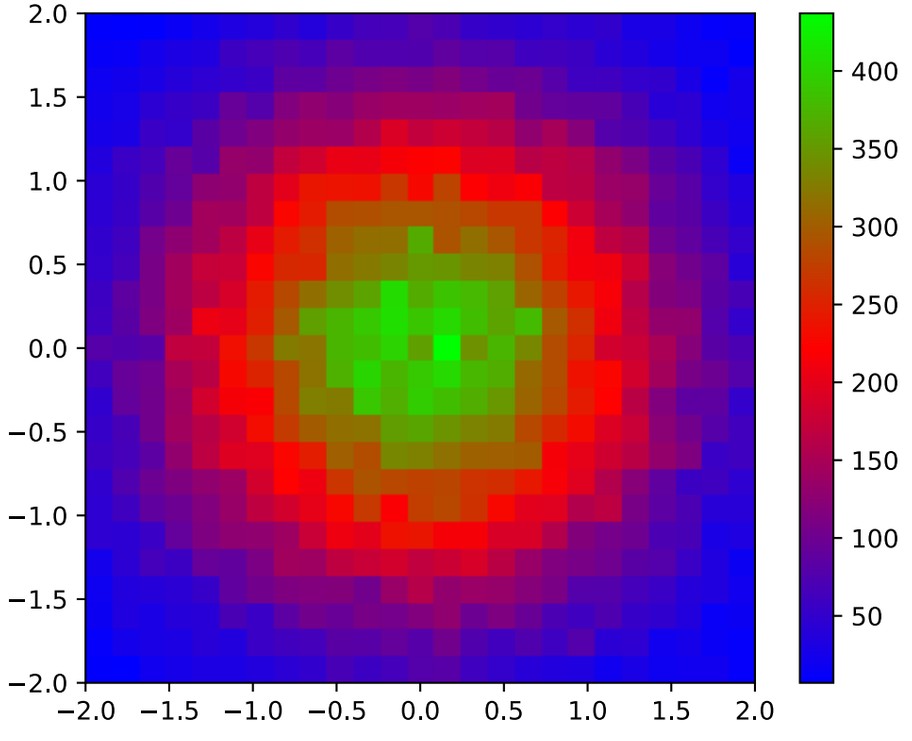
➤ Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]]) # change colormap
plt.colorbar() # add a color bar
```

IMSHOW

- Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]]) # change colormap
plt.colorbar() # add a color bar
```



IMSHOW

➤ Any 2d array can be displayed using matplotlib's imshow().

```
plt.imshow(hist) #x,y values will be indices
plt.imshow(hist,extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
plt.imshow(hist,cmap='brg',extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]]) # change colormap
plt.colorbar() # add a color bar
```

CONTOUR PLOTS

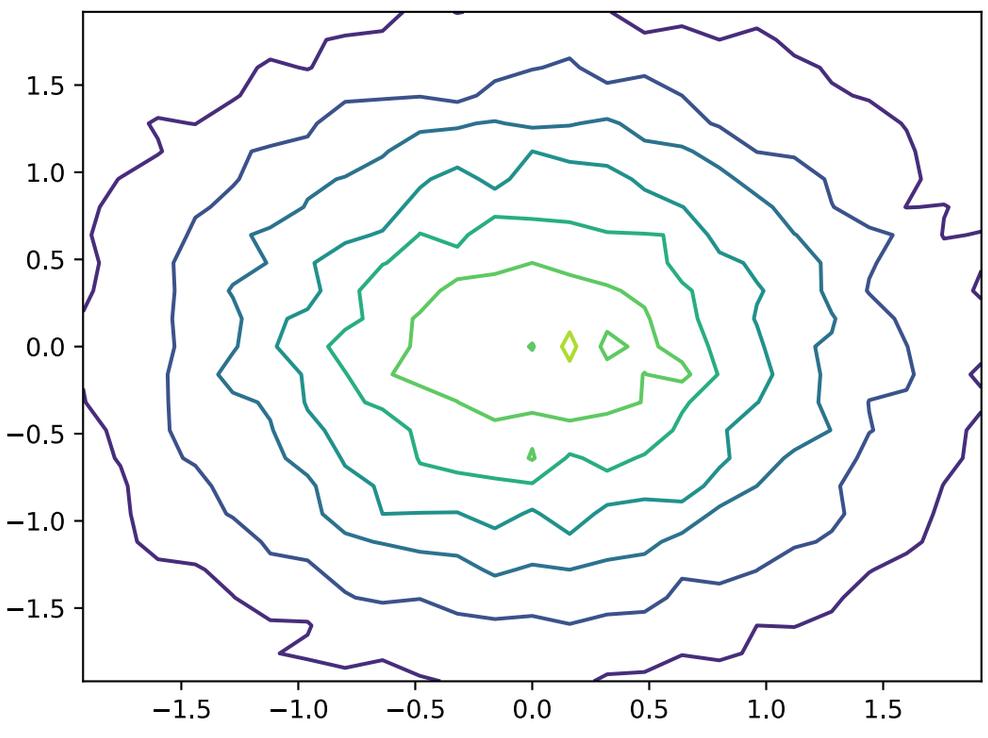
- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,200,300,400], colors=[‘b’,'c','r','m'],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
ax.clabel(cs,inline=1,fontsize=10,fmt='%d')
```

CONTOUR PLOTS

- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
```




```
0,300,400], colors=['b','c','r','m'],
-1],yedges[0],yedges[-1]])
ze=10,fmt='%d')
```

CONTOUR PLOTS

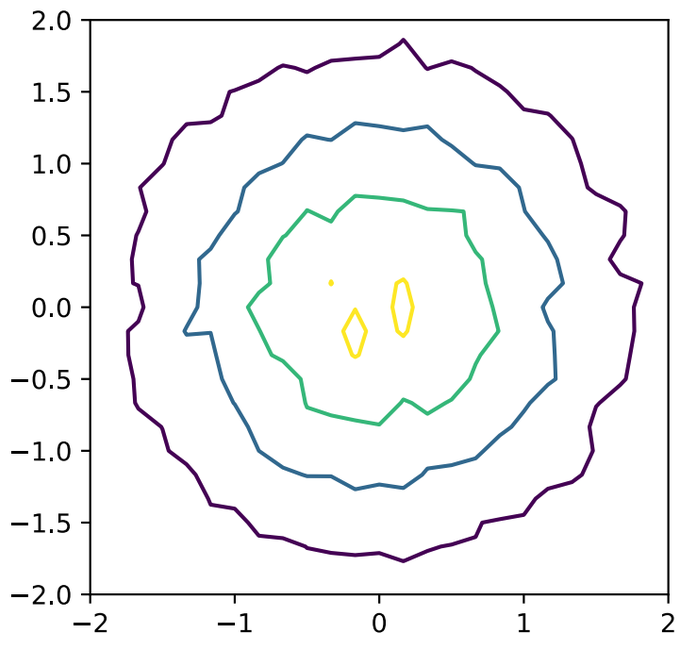
- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,200,300,400], colors=[‘b’,'c','r','m'],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
ax.clabel(cs,inline=1,fontsize=10,fmt='%d')
```

CONTOUR PLOTS

- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,2                    =['b','c','r','m'],
extent=[xedges[0],xedge                      es[-1]])
ax.clabel(cs,inline=1,font
```



CONTOUR PLOTS

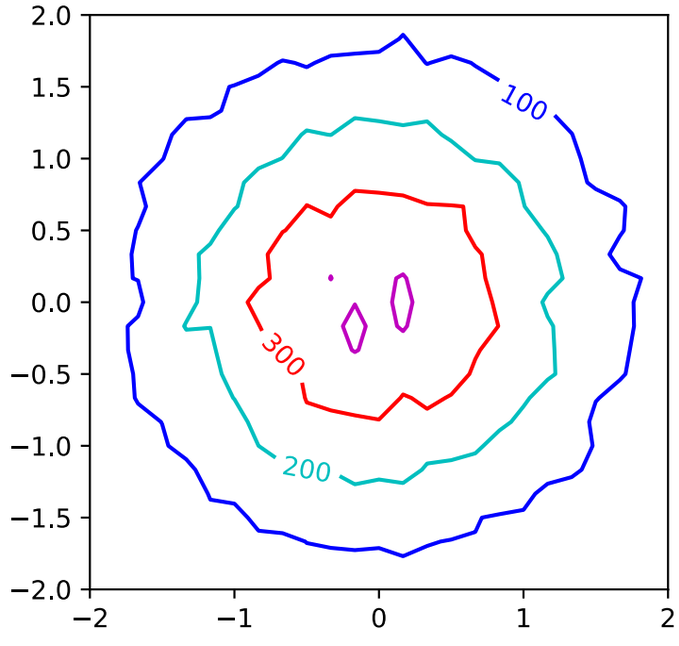
- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,200,300,400], colors=[‘b’,'c','r','m'],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
ax.clabel(cs,inline=1,fontsize=10,fmt='%d')
```

CONTOUR PLOTS

- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,200,300,400], colors=
extent=[xedges[0],xedges[-1],yedges[0],yedge
ax.clabel(cs,inline=1,fontsize=10,fmt='%d')
```



CONTOUR PLOTS

- Alternatively, one can show this type of information with a contour plot.

```
plt.contour(x,y,hist)
f,ax=plt.subplots(figsize=(4,4))
ax.contour(hist,[100,200,300,400],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
cs=ax.contour(hist,[100,200,300,400], colors=[‘b’,'c','r','m'],
extent=[xedges[0],xedges[-1],yedges[0],yedges[-1]])
ax.clabel(cs,inline=1,fontsize=10,fmt='%d')
```