2023-05-15_ChartTypes_002_Scatter

May 15, 2023

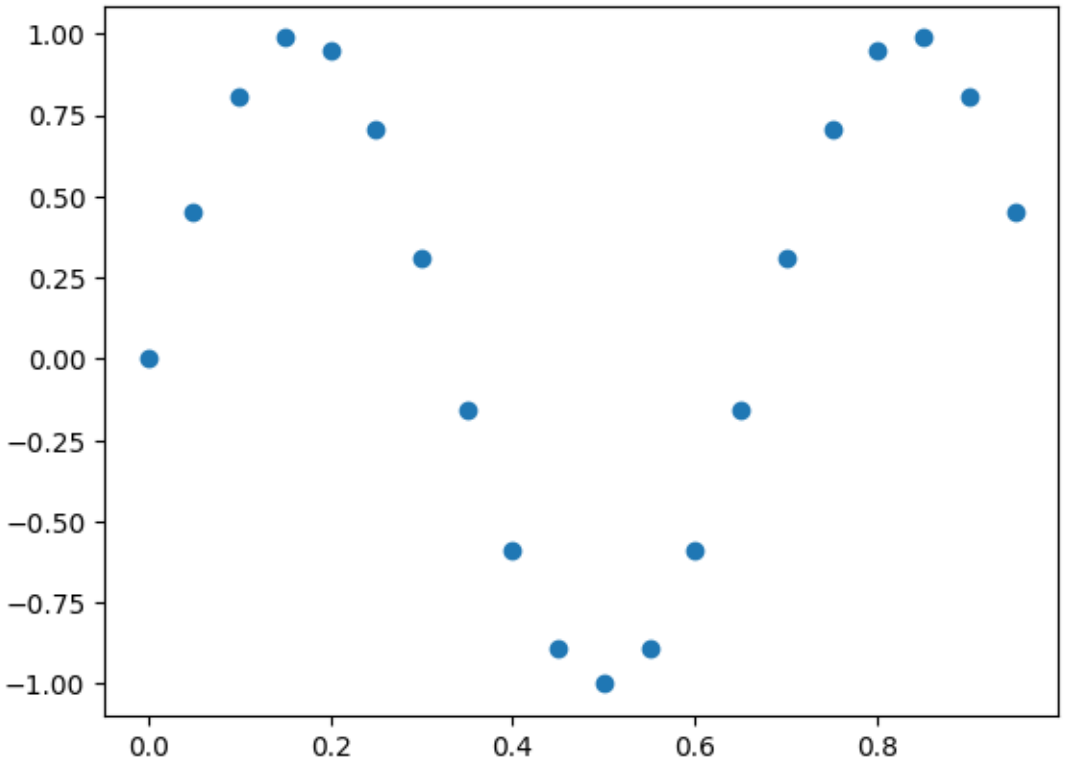
[1]:
```
import numpy as np
import scipy
import imageio

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm

matplotlib.rc('image', interpolation='nearest')
matplotlib.rc('figure',facecolor='white')
matplotlib.rc('image',cmap='viridis')
colors=plt.rcParams['axes.prop_cycle'].by_key()['color']
%matplotlib inline
```

[2]:
```
from sklearn.neighbors import KernelDensity
```

1 Scatter and line plots

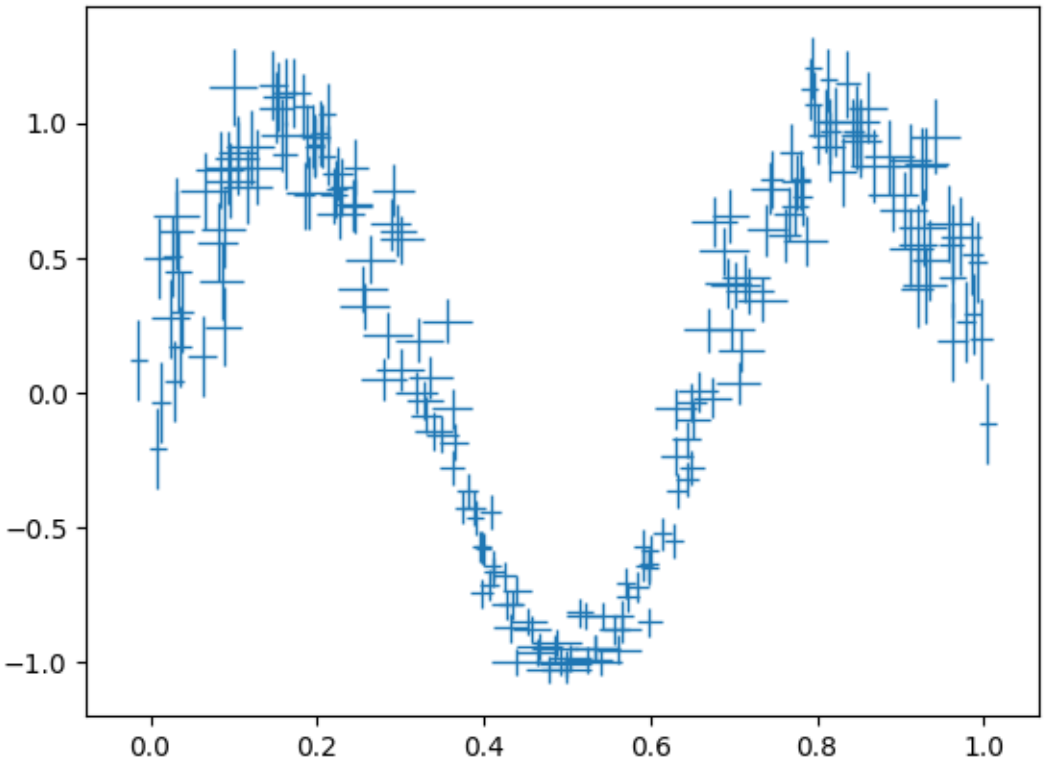
1.1 Standard simple scatter plot

[3]:
```
# generate toy data: samples from sinus curve
nX=20
x=np.arange(nX)/nX
y=np.sin(2*np.pi*x*1.5)

plt.scatter(x,y)
plt.show()
```



1.2 Error bars

[4]:
```
# error bars can represent level of uncertainty, works for both axes!

# use same data as before, now add Gaussian noise
nX=200
x=np.arange(nX)/nX
y=np.sin(2*np.pi*x*1.5)
# choose level of noise in x and y direction
# to make it interesting: the noise depends on x
stdX=0.02*np.sin(2*np.pi*x*2.5)**2+0.01
stdY=0.1*np.cos(2*np.pi*x*0.5)**2+0.05
# apply noise
xErr=x+stdX*np.random.normal(size=x.shape)
yErr=y+stdY*np.random.normal(size=y.shape)

plt.errorbar(xErr,yErr,yerr=stdY,xerr=stdX,lw=0,elinewidth=1)
plt.show()
```



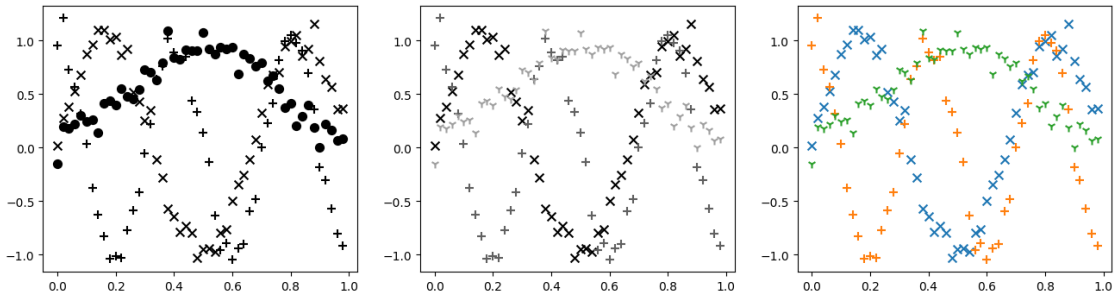
1.3 More graphical degrees of freedom

1.3.1 Markers

- ideal to encode nominal data axis, usually for separating multiple data series
- list of available markers at: https://matplotlib.org/stable/api/markers_api.html
- precise position easier to see with non-bulky marker
- if possible, use color in addition to marker for separating series (markers are merely "backup encoding")
- only a few series can be distinguished in this way

[5]:
```
# thee simple data series with some noise on top
nX=50
x=np.arange(nX)/nX
y1=np.sin(2*np.pi*x*1.5)
y2=np.cos(2*np.pi*x*2.5)
y3=np.exp(-(x-0.5)**2/.1)

stdY=0.1
for y in [y1,y2,y3]:
    y[...]+=stdY*np.random.normal(size=y.shape)
```

```
# plot with different markers
# note: markers alone quickly become messy, color usually works better, even␣
 ↪greyscale

fig=plt.figure(figsize=(16,4))
fig.add_subplot(1,3,1)
plt.scatter(x,y1,marker="x",c="#000000",s=50)
plt.scatter(x,y2,marker="+",c="#000000",s=50)
plt.scatter(x,y3,marker="o",c="#000000",s=50)


fig.add_subplot(1,3,2)
plt.scatter(x,y1,marker="x",c="#000000",s=50)
plt.scatter(x,y2,marker="+",c="#606060",s=50)
plt.scatter(x,y3,marker="1",c="#a0a0a0",s=50)

fig.add_subplot(1,3,3)
plt.scatter(x,y1,marker="x",c=colors[0],s=50)
plt.scatter(x,y2,marker="+",c=colors[1],s=50)
plt.scatter(x,y3,marker="1",c=colors[2],s=50)

plt.show()
```

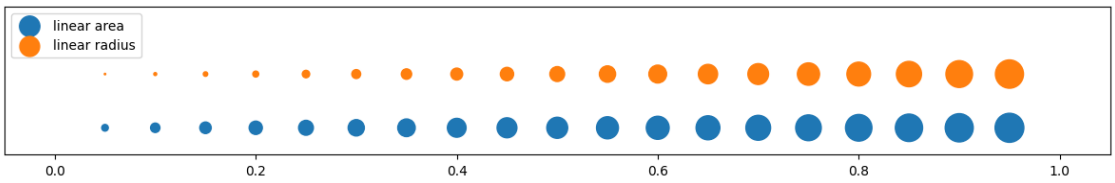


1.3.2 Markersize

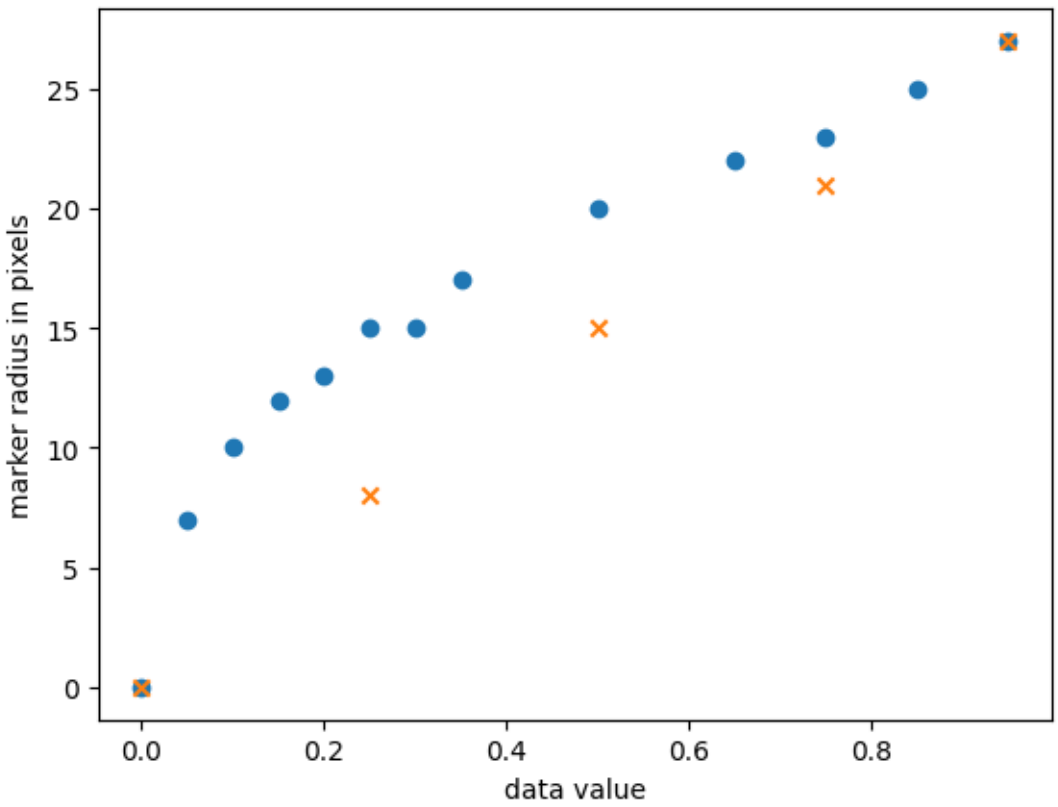
- can be used to encode third, continuous, positive dimension
- something that behaves like a mass, since points with zero marker size will be invisible
- careful: scale area (not radius!) of markers proportional to value (matplotlib does this automatically in newer versions)

Example: scale markersize area and radius

[6]:
```
fig=plt.figure(figsize=(15,2))
n=20
x=np.arange(n)/n

y=np.full(shape=n,fill_value=0.15)
plt.scatter(x,y,s=500*x,label="linear area")

y=np.full(shape=n,fill_value=0.55)
plt.scatter(x,y,s=500*(x**2),label="linear radius")

plt.xlim([-0.05,1.05])
plt.ylim([-0.05,1.05])
plt.yticks([])
plt.legend(loc=2)
plt.show()
```



[7]:
```
# for first sequence (blue): pairs of data point nr and measured radii in pixels
radii1=np.
  ↪array([[19,27],[17,25],[15,23],[13,22],[10,20],[7,17],[6,15],[5,15],[4,13],[3,12],[2,10],[1
radii2=np.array([[19,27],[15,21],[10,15],[5,8],[0,0]])
plt.scatter(radii1[:,0]/n,radii1[:,1])
plt.scatter(radii2[:,0]/n,radii2[:,1],marker="x")
plt.ylabel("marker radius in pixels")
plt.xlabel("data value")
plt.show()
```



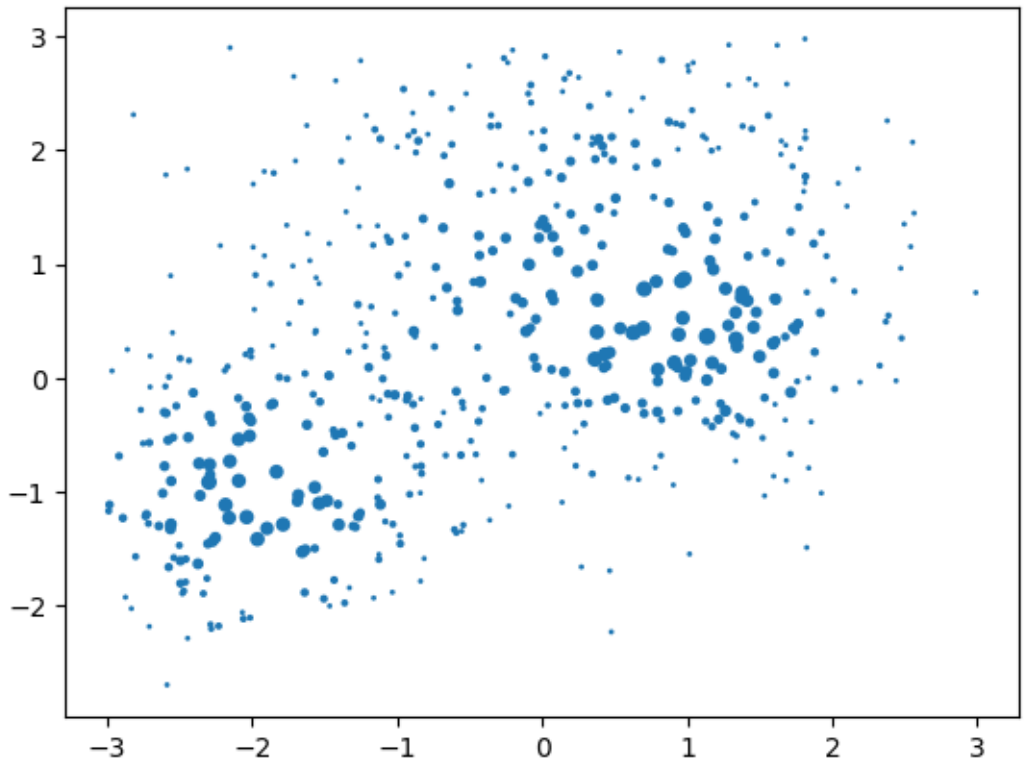
Example: visualize scalar function that is known at sampled points

[8]:
```
# define a 2d density function (based on a mixture of Gaussians)
dim=2
rng=[-3,3]
meanList=np.array([[-2,-1],[0,1],[1,0.5]])
stdList=np.array([0.5,0.9,0.5])
weightList=np.array([0.3,0.4,0.3])

def getDens(x):
    val=0
    for mean,std,weight in zip(meanList,stdList,weightList):
        val+=(1./np.sqrt(2*np.pi)**dim/std**dim)*np.exp(-0.5*np.
 ↪sum((x-mean)**2)/std**2)*weight
    return val
```

[9]:
```
# now, sample points uniformly from range, sample point mass from Poisson␣
 ↪distribution given by local density
nSamples=1000
lambdaPoisson=100
x=rng[0]+(rng[1]-rng[0])*np.random.random(size=nSamples)
```

```
y=rng[0]+(rng[1]-rng[0])*np.random.random(size=nSamples)
densxy=np.array([getDens([a,b]) for a,b in zip(x,y)])
m=np.random.poisson(lam=lambdaPoisson*densxy,size=densxy.shape)

# discard zero mass points
keep=(m>0)
x=x[keep]
y=y[keep]
m=m[keep]
```

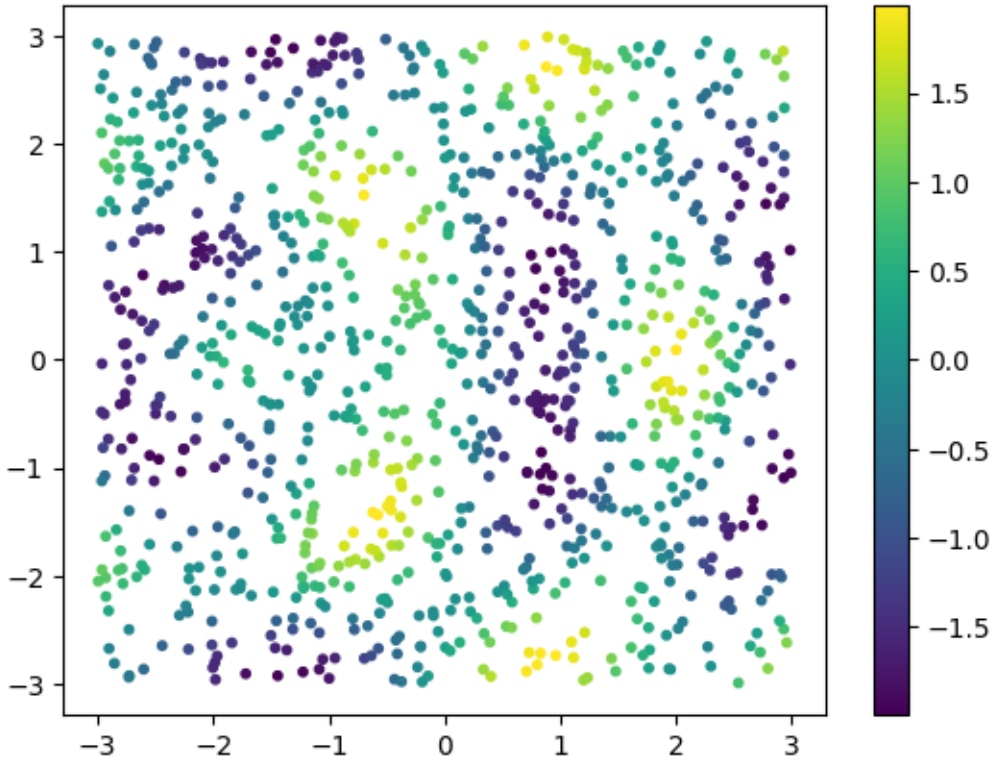
[10]:
```
plt.scatter(x,y,s=m)
plt.show()
```



1.3.3 Color

- separate different data series
- show an independent third dimension, such as height, temperature, density,...
- recall lesson on colors and color maps
- mostly a qualitative display

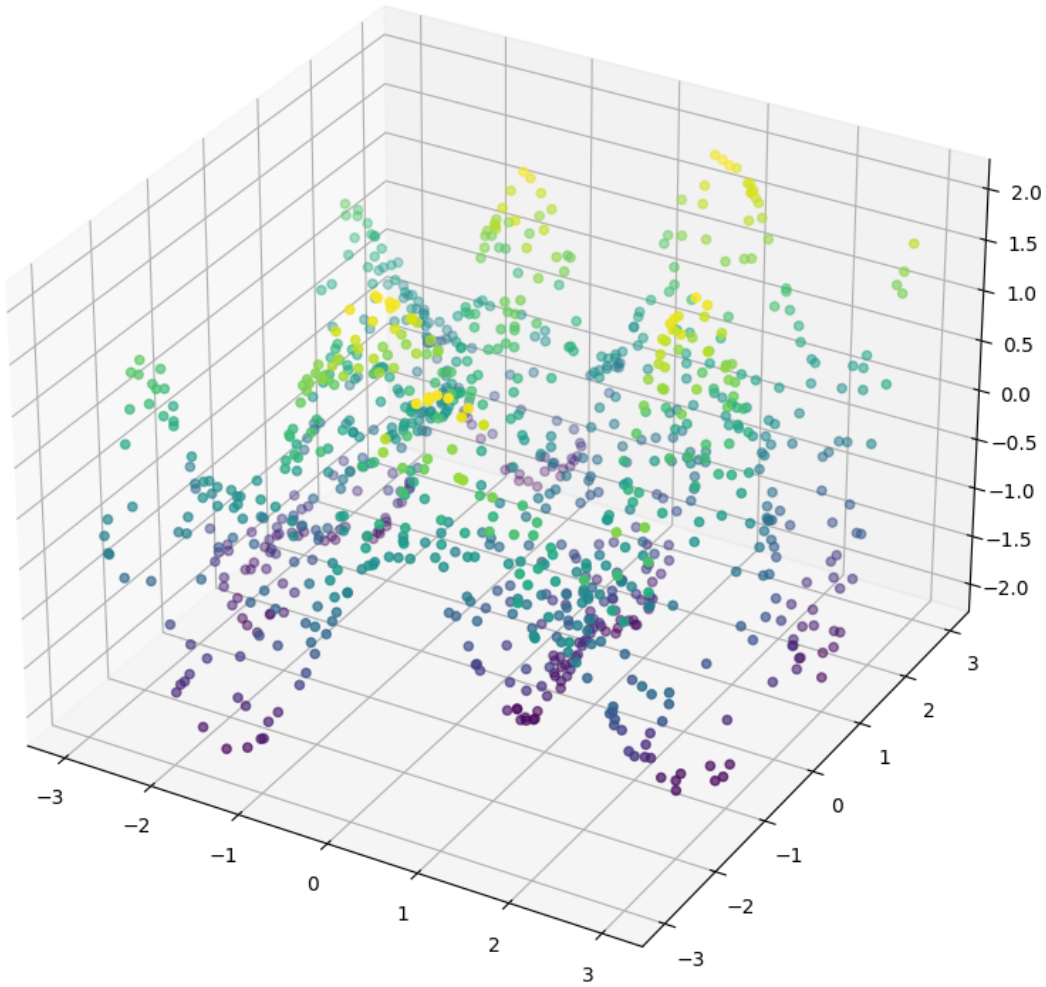
[11]:
```
# similar to marker size example: visualize scalar function on top of points
# toy data
nSamples=1000
rng=[-3,3]
# sample x and y coords uniformly from rectangle
x=rng[0]+(rng[1]-rng[0])*np.random.random(size=nSamples)
y=rng[0]+(rng[1]-rng[0])*np.random.random(size=nSamples)
# compute f as function of x and y
r1=((x-2)**2+y**2)**.5
r2=((x+2)**2+y**2)**.5
f=np.cos(2*np.pi*r1/3)+np.cos(2*np.pi*r2/2)

# visualize f as color in scatter plot of x and y
fig=plt.figure()
ax=fig.add_subplot()
pltobj=plt.scatter(x,y,c=f,s=10)
fig.colorbar(pltobj, ax=ax)

plt.show()
```



1.4 3d

- visualize additional dimension
- works best with animation or interactive rotation
- perspective is often problematic in 2d static renderings, use with caution; use auxiliary lines such as a mesh grid or color
- data has even more dimensions? will be addressed later

[12]:
```
#%matplotlib widget
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x,y,f,c=f)
plt.tight_layout()
plt.show()
```



[13]:
```
plt.close()
%matplotlib inline
```

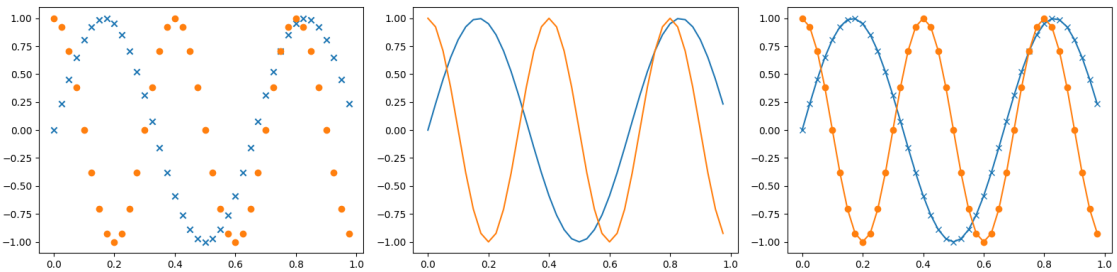
1.5 Lines

when is adding lines appropriate/helpful? * if x-axis is interval data, actually or practically continuous * to visually connect a data series for better distinction from other series * to emphasize trends (line gives explicit encoding of slope) → regression lines * if line is decent approximation of intermediate values that might also be measured

do not use lines if data is highly oscillating, or subject to strong stochastic fluctuations * on data with clear trend + overlap by noise, maybe a smoothed interpolation is appropriate (always be careful to mention how it is generated)

1.5.1 Better visual conectivity of series, emphasis of trends

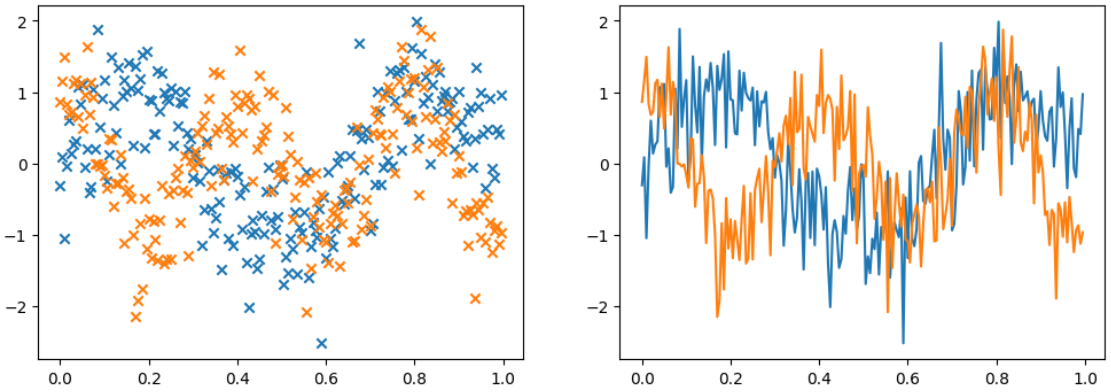
[14]:
```
# toy data
nX=40
x=np.arange(nX)/nX
y1=np.sin(2*np.pi*x*1.5)
y2=np.cos(2*np.pi*x*2.5)

fig=plt.figure(figsize=(16,4))
fig.add_subplot(1,3,1)
plt.scatter(x,y1,marker="x")
plt.scatter(x,y2,marker="o")
fig.add_subplot(1,3,2)
plt.plot(x,y1)
plt.plot(x,y2)
fig.add_subplot(1,3,3)
plt.plot(x,y1,marker="x")
plt.plot(x,y2,marker="o")

plt.tight_layout()
plt.show()
```



1.5.2 Be careful on noisy or oscillating data

- strictly speaking not meaningful, but possibly still better (subjective), be careful about misinterpretation
- below: regression curves

[15]:
```
# noisy toy data
nX=200
x=np.arange(nX)/nX
y1=np.sin(2*np.pi*x*1.5)+0.5*np.random.normal(size=nX)
y2=np.cos(2*np.pi*x*2.5)+0.5*np.random.normal(size=nX)

fig=plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
plt.scatter(x,y1,marker="x")
plt.scatter(x,y2,marker="x")
fig.add_subplot(1,2,2)
plt.plot(x,y1)
plt.plot(x,y2)

plt.show()
```

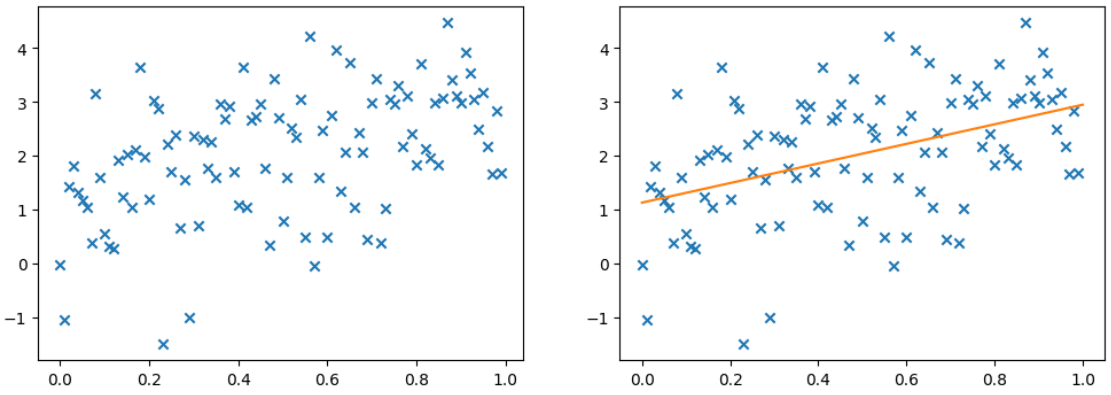


Linear regression

[16]:
```
import scipy.stats
```

[17]:
```
# toy data: y=2x+1 + noise
nX=100
x=np.arange(nX)/nX
```

```
y=2*x+1+np.random.normal(size=nX)

# linear regression
reg=scipy.stats.linregress(x,y)
xref=np.linspace(0,1,num=50)


fig=plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
plt.scatter(x,y,marker="x")

fig.add_subplot(1,2,2)
plt.scatter(x,y,marker="x")
plt.plot(xref,reg.slope*xref+reg.intercept,c=colors[1])

plt.show()
```



Kernel regression

- given: values y_i at locations x_i, interpolate to full space X via:

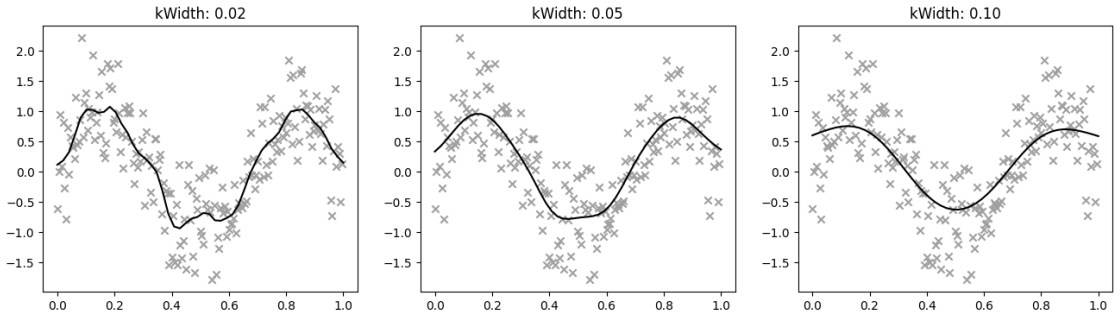
$$f(x) = \frac{\sum_i y_i \cdot k(x, x_i)}{\sum_i k(x, x_i)}$$

- here k is often a Gaussian kernel with a suitable kernel width
- if width is very small: becomes essentially nearest neighbour interpolation
- if width is very high: becomes low pass filter

[19]:
```
def simpleKernelInterpolator(x,y,z,kWidth):
    # works only for 1d arrays x,y,z; not very computationally efficient, only␣
 ↪works on small arrays
    # input empirical data pairs (x,y); interpolate y onto points given by z
    # via Gaussian kernels: at each z sum value of nearby y with Gaussian␣
 ↪weights; normalized to 1
```

```
    # use kWidth for Gaussian kernel width
    weights=np.exp(-0.5*(x.reshape(-1,1)-z.reshape(1,-1))**2/kWidth**2)
    signal=np.einsum(weights,[0,1],y,[0],[1])/np.sum(weights,axis=0)
    return signal
```

[20]:
```
nX=200
x=np.arange(nX)/nX
y1=np.sin(2*np.pi*x*1.5)+0.5*np.random.normal(size=nX)
z=np.linspace(0,1,num=50)

fig=plt.figure(figsize=(16,4))


for i,kWidth in enumerate([0.02,0.05,0.1]):
    fig.add_subplot(1,3,1+i)

    signal=simpleKernelInterpolator(x,y1,z,kWidth)
    plt.scatter(x,y1,marker="x",color="#A0A0A0")
    plt.plot(z,signal,color="#000000")
    plt.title("kWidth: {:.02f}".format(kWidth))
plt.show()
```

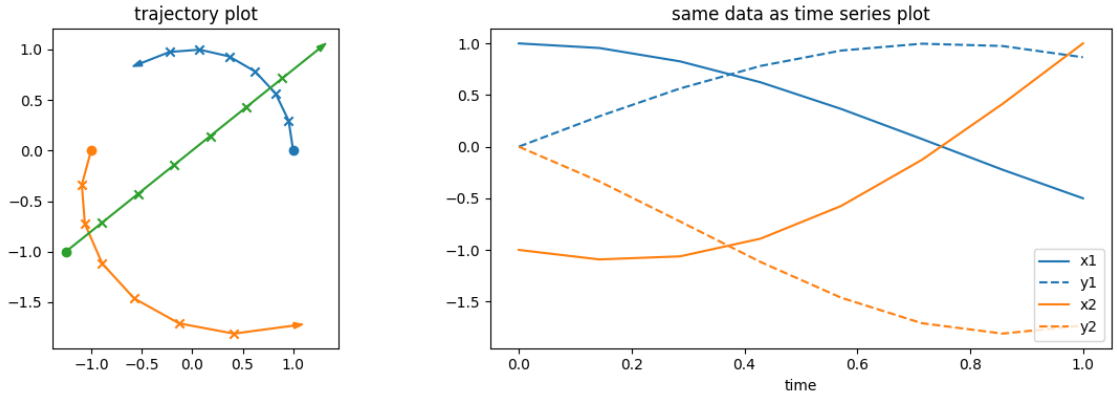


1.5.3 Trajectory plot / connected scatter plot

[21]:
```
def pltTrajectory(ax,x,y,**kwargs):
    ax.plot(x,y,**kwargs)
    ax.arrow(x[-1],y[-1],0.01*(x[-1]-x[-2]),0.01*(y[-1]-y[-2]),**kwargs,width=0.
 ↪02)
    ax.scatter(x[0],y[0],**kwargs,marker="o")
    ax.scatter(x[1:-1],y[1:-1],**kwargs,marker="x")
```

[22]:
```
nPts=8
t=np.linspace(0,1,nPts)
```

```
x1=np.cos(2*np.pi*t/3)
y1=np.sin(2*np.pi*t/3)

x2=(1+t)*np.cos(2*np.pi*(t/3+0.5))
y2=(1+t)*np.sin(2*np.pi*(t/3+0.5))

x3=2.5*(t-0.5)
y3=2*(t-0.5)


fig=plt.figure(figsize=(12,4))
# since now both axes represent length in same "physical space", aspect ratio␣
 ↪of axes becomes relevant!
ax=fig.add_subplot(1,2,1,aspect=1.)
plt.title("trajectory plot")
pltTrajectory(ax,x1,y1,color=colors[0])
pltTrajectory(ax,x2,y2,color=colors[1])
pltTrajectory(ax,x3,y3,color=colors[2])

fig.add_subplot(1,2,2)
plt.title("same data as time series plot")
plt.plot(t,x1,color=colors[0],label="x1")
plt.plot(t,y1,color=colors[0],ls="dashed",label="y1")
plt.plot(t,x2,color=colors[1],label="x2")
plt.plot(t,y2,color=colors[1],ls="dashed",label="y2")
plt.xlabel("time")
plt.legend(loc=4)

plt.tight_layout()
plt.show()
```



Example: interacting particles in potential Here we consider a simple model of n interacting particles in 2 dimensions. Each particle follows the following equation of motion:

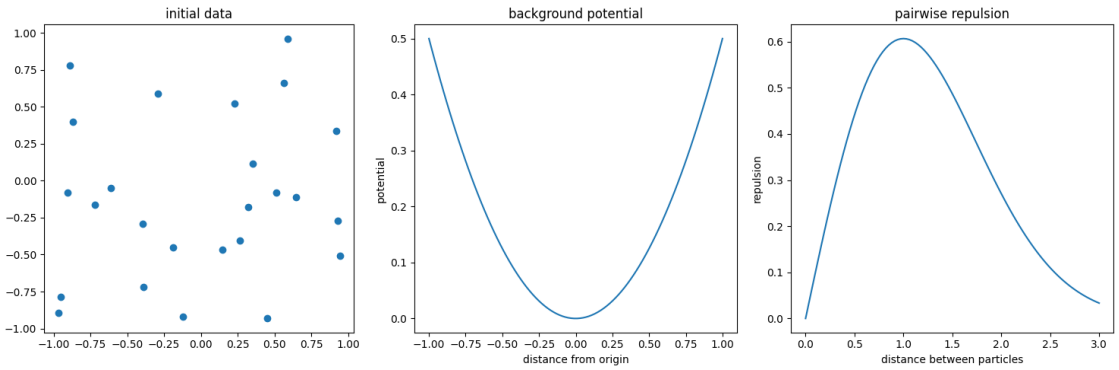
$$\partial_t x_i = -\nabla V(x_i) + \frac{1}{n}\sum_{j=1}^{n} F(x_j - x_i)$$

* for some initial positions $(x_1(0), \dots, x_n(0))$ * for some background potential V, here simply $V(x) = \frac{1}{2}\|x\|^2$ * for some interaction force, here $F(z) = -z \cdot \exp(-\|z\|^2/2)$

[22]:
```
# set up the simulation
dim=2

def gradV(x):
    """gradient of static background potential in which the particles want to␣
 ↪move
    (in this case: simple quadratic well around origin)"""
    return x

def g(s):
    """scalar function that modulates strength of pairwise interaction"""
    return np.exp(-0.5*s**2)

def pairwiseForce(x,y):
    """pairwise forces between all particles"""
    # relative position between all particles
    deltaij=x.reshape((-1,1,dim))-y.reshape((1,-1,dim))
    # relative distance
    dij=np.linalg.norm(deltaij,axis=2)
    fi=np.einsum(deltaij,[0,1,2],g(dij),[0,1],[0,2])/n
    return fi


# initial conditions
if False:
    # uniform on 2d grid
    n1d=5
    x0_1d=np.linspace(-1,1,num=n1d)
    x0=np.zeros((n1d,n1d,dim))
    x0[:,:,0]=x0_1d.reshape((1,-1))
    x0[:,:,1]=x0_1d.reshape((-1,1))
    x0=x0.reshape((-1,dim))
if True:
    # uniform random sampling from unit square
    n=25
    x0=2*(np.random.random(size=(n,dim))-0.5)
if False:
    # manual
    x0=np.array([[1,0],[-1,0]])
```

```
n=x0.shape[0]
```

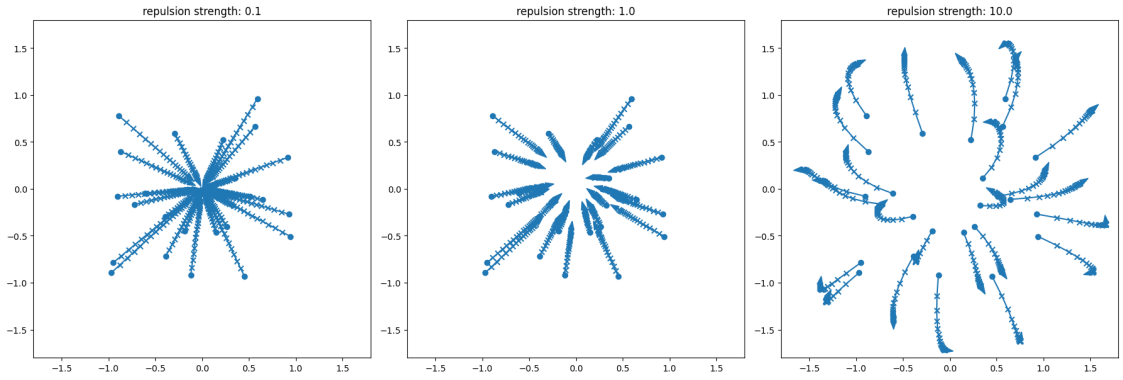
[23]:
```
# before we get to actual simulation:
# * visualize initial data
# * visualize potential
# * visualize pairwise force
fig=plt.figure(figsize=(15,5))

fig.add_subplot(1,3,1,aspect=1.)
plt.title("initial data")
plt.scatter(x0[:,0],x0[:,1])

fig.add_subplot(1,3,2)
plt.title("background potential")
z=np.linspace(-1,1,num=100)
plt.plot(z,0.5*z**2)
plt.xlabel("distance from origin")
plt.ylabel("potential")

fig.add_subplot(1,3,3)
plt.title("pairwise repulsion")
z=np.linspace(0,3,num=100)
plt.plot(z,z*np.exp(-z**2/2))
plt.xlabel("distance between particles")
plt.ylabel("repulsion")


plt.tight_layout()
plt.show()
```



[24]:
```
def getSimulation(x0,forceStrength):
    nT=1000
    tList=np.linspace(0,2,num=nT)
```

```
    def f(t,x):
        return (-gradV(x.reshape((-1,dim)))+\
                forceStrength*pairwiseForce(x.reshape((-1,dim)),x.
 ↪reshape((-1,dim)))).ravel()

    res=scipy.integrate.solve_ivp(f, tList[[0,-1]], x0.ravel(), method='RK45',␣
 ↪t_eval=tList)
    x=res["y"].transpose().reshape((nT,n,dim))
    return x
```

[25]:
```
fig=plt.figure(figsize=(18,6))

nSubT=50 # only show every nSubT-th point to avoid plot being too cluttered

# now do this simulation for various forces
# show them in same coordinate system (axis ranges)
for k,forceStrength in enumerate([0.1,1,10]):
    ax=fig.add_subplot(1,3,1+k,aspect=1.)
    plt.title("repulsion strength: {:.01f}".format(forceStrength))

    x=getSimulation(x0,forceStrength)
    for i in range(n):
        pltTrajectory(ax,x[::nSubT,i,0],x[::nSubT,i,1],color=colors[0])
    scale=1.8
    ax.set_xlim([-scale,scale])
    ax.set_ylim([-scale,scale])
plt.tight_layout()
plt.show()
```



1.6 Logarithmic axis scaling

1.6.1 Application 1: visualize exponential growth

consider following model:

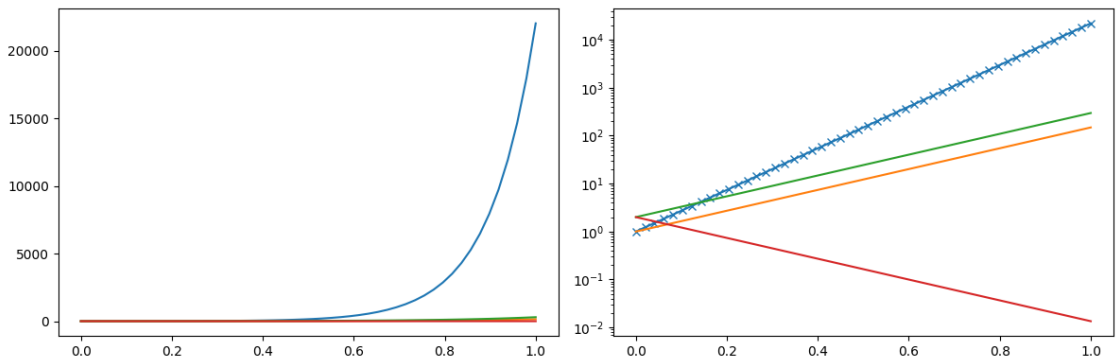
$$y(x) = a \cdot \exp(b \cdot x)$$

Now scale y-axis logarithmically, that is, plot $\log(y)$ instead of y. One finds:

$$\log(y(x)) = \log(a) + b \cdot x$$

* linear relation beween $\log(y)$ and x * prefactor a becomes intercept $\log(a)$ * time-scale b becomes slope

[26]:
```
nX=50
x=np.linspace(0,1,num=nX)
y1=np.exp(10*x)
y2=np.exp(5*x)
y3=2*np.exp(5*x)
y4=2*np.exp(-5*x)

fig=plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
plt.plot(x,y1,marker="")
plt.plot(x,y2,marker="")
plt.plot(x,y3,marker="")
plt.plot(x,y4,marker="")

fig.add_subplot(1,2,2)
plt.plot(x,y1,marker="x")
plt.plot(x,y2,marker="")
plt.plot(x,y3,marker="")
plt.plot(x,y4,marker="")
plt.yscale("log")

plt.tight_layout()
plt.show()
# note: markers of exponential data equally spaced in log plot
```



1.6.2 Application 2: visualize power laws

consider following model:

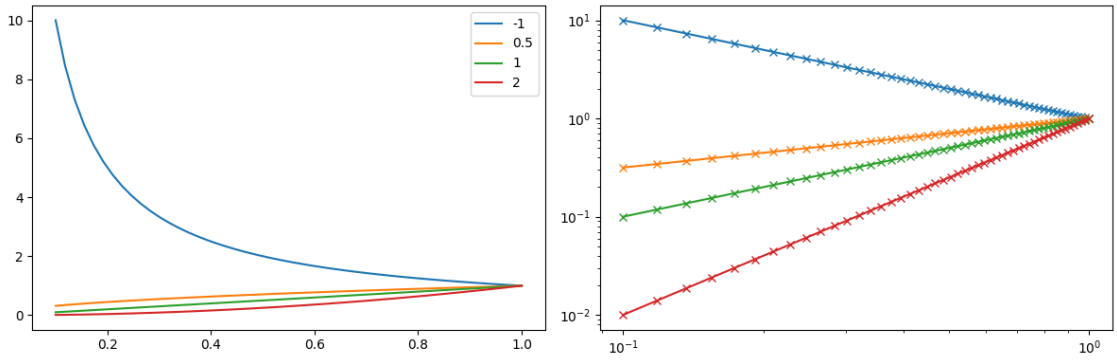
$$y(x) = ax^p$$

Now scale x and y-axis logarithmically. Now need to express $\log(y)$ as function of $\log(x)$:

$$\log(y) = \log(a) + p \cdot \log(x)$$

* linear relation beween $\log(y)$ and $\log(x)$ * prefactor a becomes intercept $\log(a)$ * exponent p becomes slope

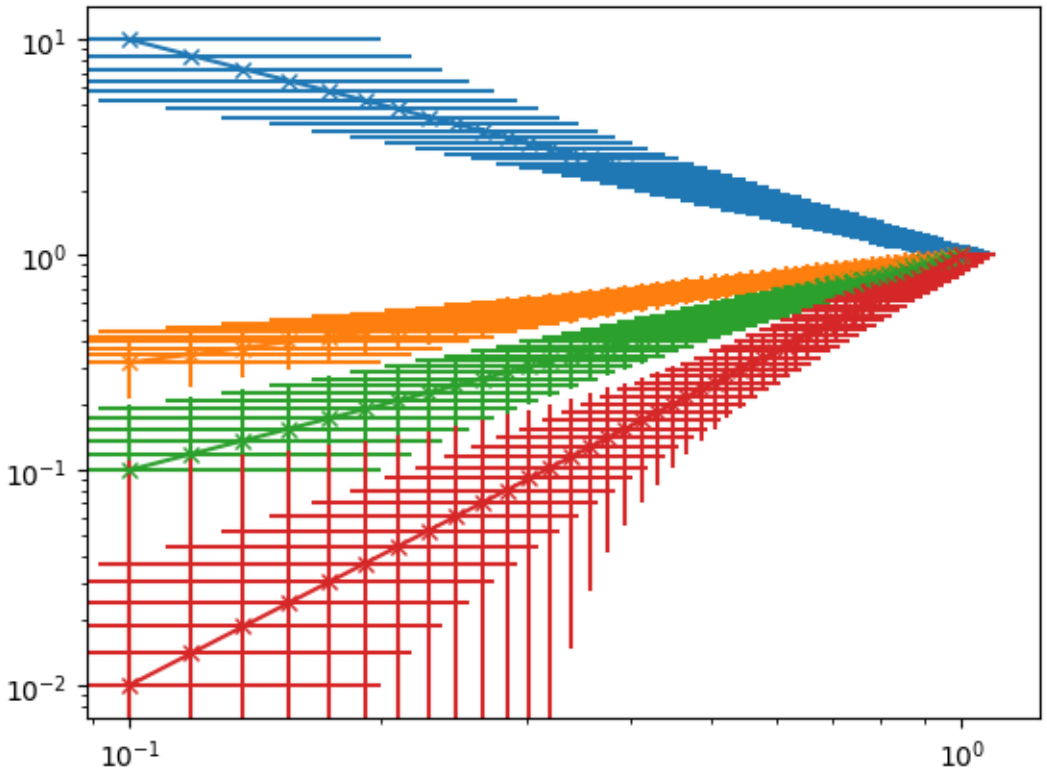
[27]:
```
nX=50
x=np.linspace(0.1,1,num=nX)
pList=[-1,0.5,1,2]
yList=[x**p for p in pList]

fig=plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
for p,y in zip(pList,yList):
    plt.plot(x,y,marker="",label=p)
plt.legend()
#plt.ylim([0,2])

fig.add_subplot(1,2,2)
for y in yList:
    plt.plot(x,y,marker="x")
plt.xscale("log")
plt.yscale("log")

plt.tight_layout()
plt.show()
# note: markers no longer equally spaced in x
```



Errorbars in log plot

- note: if errors are constant in size, they will look vastly different in log plot
- therefore: be very careful with regression on such data! linear regression on log-data is usually not a good idea

[28]:
```
fig=plt.figure()
fig.add_subplot()
for y in yList:
    plt.errorbar(x,y,xerr=0.1,yerr=0.1,marker="x")
plt.xscale("log")
plt.yscale("log")

plt.show()
```



[ ]:
```

```