# 2023-05-15_ChartTypes_004_StochasticFunctionalRelation

May 25, 2023

[43]:
```
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
```

## 1 Stochastic functional relation

- In previous scatter plot examples data was often of form $y = f(x)$, show $y$ as function of $x$
- Maybe with 'a little bit of noise': $y = f(x) + \varepsilon$, in most examples $\varepsilon$ was normal distributed. In these cases means and error bars were enough to visualize distribution of $y$ for each $x$.
- Then went to histograms: visualize distribution of some random $y$ distributed according to some distribution $\mu$
- Now combine the two: Let $y$ be distributed like $\mu_x$ where $\mu_x$ depends on $x$ (and probably is more complicated than a simple normal distribution), will call this 'stochastic functional relation'. So we need to visualize distribution $\mu_x$ of $y$ for each $x$. How can we do this?
- reminder: usually full $\mu_x$ is not available, only sampled data

### 1.1 Data examples

#### 1.1.1 Data example 1: unimodal relation

$y = \sin(4\pi x) + \sigma_x \cdot z$ where $\sigma_x$ is a standard deviation, depending on $x$, and $z$ is normal distributed.

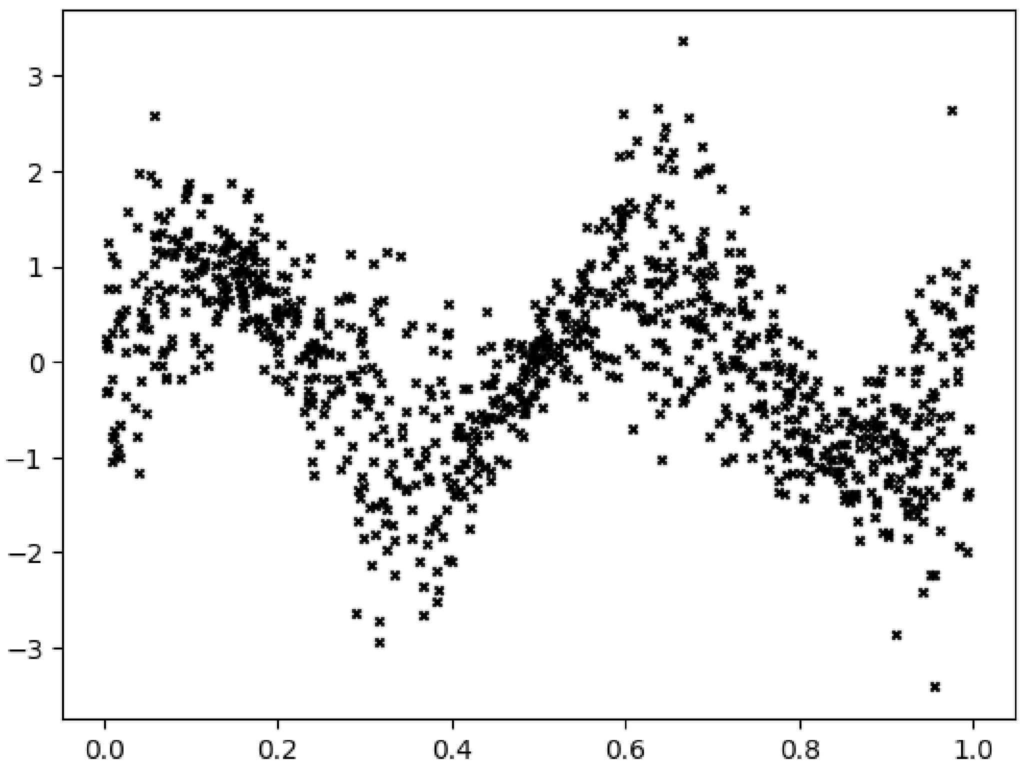
[44]:
```
# nr of points
nP=1000
```

```
# X: uniform sampling from 0 to 1
dataX=np.random.random(size=(nP,))
# Y
dataMean=np.sin(2*np.pi*dataX*2)
dataStd=np.sin(2*np.pi*(dataX+0.5)*1.5)**2*0.7+0.3
dataY=dataMean+dataStd*np.random.normal(size=nP)
```

## 1.2 Visualization of data

### 1.2.1 Preview: simple scatter plot

[45]:
```
plt.scatter(dataX,dataY,marker="x",c="k",s=10)
plt.show()
```




### 1.2.2 Kernel density estimation

- as in histogram chapter, could try and estimate 2d density of $(x, y)$ distribution via kernel density estimation and visualize level lines of this
- something similar will be done on problem sheet, here we proceed with other strategies instead

### 1.2.3 Binning the data

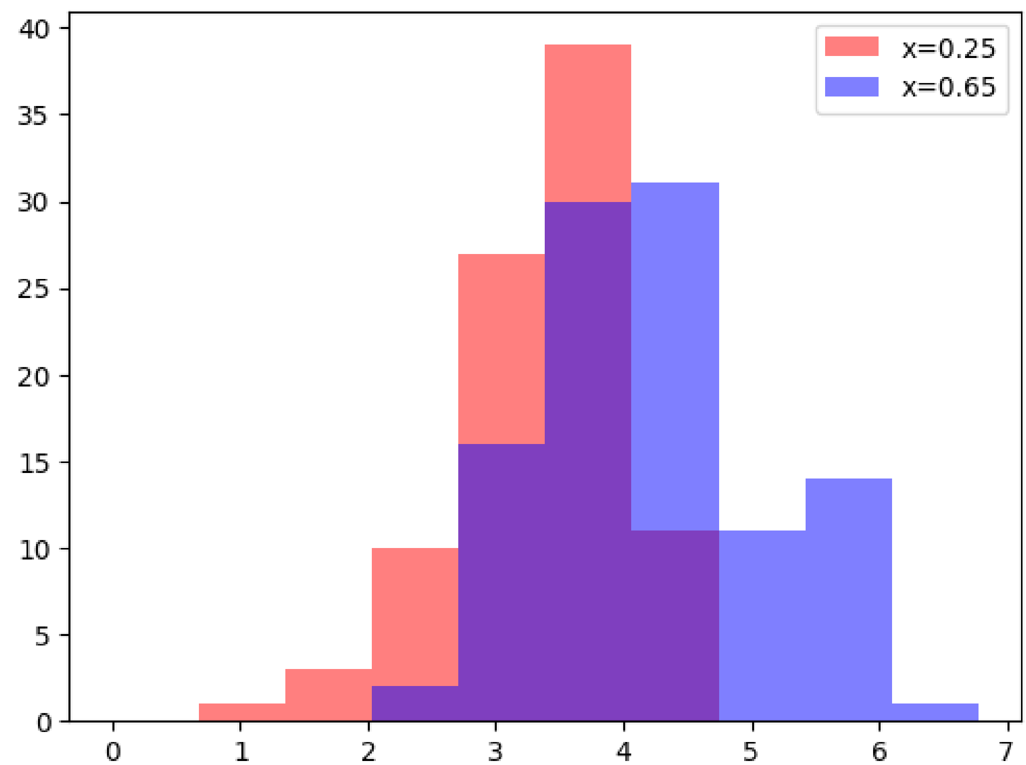
- one strategy is to partition data into bins along x-axis, and try to visualize distribution along y for each bin
- for this, need to bin data first, use simple custom function (will not scale to very large examples)

[46]:
```
def binData(dataX,dataY,nBins):
    """Divide range of dataX values into nBins even sized bins.
    Generate sublists of dataY for the points that lie in each of the bins.
    Returns (midpoints,binnedYLists).
    midpoints: list of midpoints of the X-bins, binnedYLists: list of Y-values␣
 ↪within each bin"""
    vMax=np.max(dataX)
    vMin=np.min(dataX)
    # compute bin nr for each data point, based on x coordinate
    binNr=np.clip(((dataX-vMin)/(vMax-vMin+1E-10)*nBins).astype(np.
 ↪int32),0,nBins-1)
    # generate list of bin midpoints
    midpoints=((np.arange(nBins)+0.5)/nBins)*(vMax-vMin)+vMin
    # create binned YLists, based on binNr array:
    binnedYLists=[[] for i in range(nBins)]
    for b,y in zip(binNr,dataY):
        binnedYLists[b].append(y)
    return midpoints,binnedYLists
```

[47]:
```
# try a low number of bins now, can also experiment with higher numbers if␣
 ↪desired
nBinsX=10
midpointsX,binnedYLists=binData(dataX,dataY,nBinsX)

# compute histograms over y for each x-bin

nBinsY=10

# range of y-data
rng=[np.min(dataY),np.max(dataY)]
histList=[np.histogram(binnedYList,range=rng,bins=nBinsY) for binnedYList in␣
 ↪binnedYLists]
# each entry of this list is of form hist, bin_edges where hist contains the␣
 ↪counts, bin_edges contains
# information about bin boundaries
```

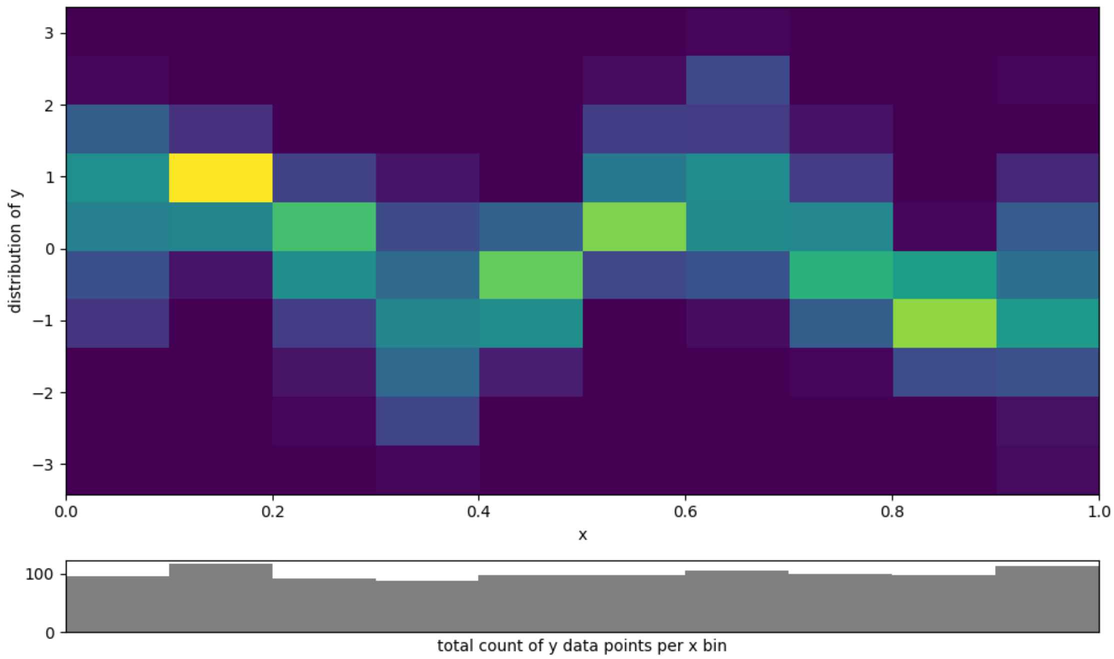
### 1.2.4 For a first, local impression, plot a few individual histograms

[48]:
```
# use transparency to display multiple histograms as overlapping bar charts

# width and midpoints of bins along Y-axis
deltaX=(rng[1]-rng[0])/nBinsY
midpointsY=deltaX*(np.arange(nBinsY)+0.5)

# colorList
colorList=[(1,0,0,0.5),(0,0,1,0.5)]

for i,iX in enumerate([2,6]):
    plt.
 ↪bar(midpointsY,histList[iX][0],width=deltaX,color=colorList[i],label="x={:.
 ↪02f}".format(midpointsX[iX]))
plt.legend()
plt.show()
```



### 1.2.5 Try to visualize all histograms for better global impression

[49]:
```
# show all histograms as color coded image
# * we will normalize each xBin to unit mass
# * show absolute count of data points per xBin below in separate plot
# * during generation of the image be careful about axis order and alignments!␣
 ↪(more details later)

xcounts=[len(binnedYList) for binnedYList  in binnedYLists]

img=np.zeros((nBinsY,nBinsX),dtype=np.double)
for iX in range(nBinsX):
    # write each y-histogram into corresponding column of image
    img[:,iX]=histList[iX][0]
    # normalize this colum
    img[:,iX]=img[:,iX]/np.sum(img[:,iX])


fig=plt.figure(figsize=(10,6))

gs = fig.add_gridspec(5, 1)

fig.add_subplot(gs[:-1,:])
plt.imshow(img,extent=(0,1,rng[0],rng[1]),aspect="auto",origin="lower")
plt.xlabel("x")
plt.ylabel("distribution of y")

fig.add_subplot(gs[-1,:])
plt.bar((np.arange(nBinsX)+0.5)/nBinsX,xcounts,width=1/nBinsX,color="#808080")
plt.xlim([0,1])
plt.xticks([])
plt.xlabel("total count of y data points per x bin")
plt.tight_layout()
plt.show()
```




### 1.2.6 Interlude

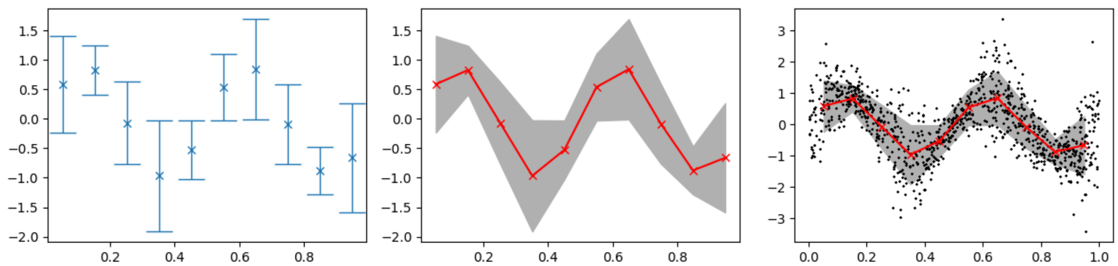
- this encodes most of the information contained in the data
- but it is complicated and color coding is not very quantitative
- in many cases a simpler, but more quantitative representation of the histograms can be shown instead
- now go through a few examples for this

### 1.2.7 Mean and standard deviation

- simplest version: for each bin show mean and standard deviation
- represent in figure by markers with error bars

[50]:
```
# compute mean and standard variation of y in each bin
ymeans=np.array([np.mean(y) for y in binnedYLists])
ystd=np.array([np.std(y) for y in binnedYLists])

fig=plt.figure(figsize=(12,3))
fig.add_subplot(1,3,1)
# display this as scatter plot with error bars
plt.
 ↪errorbar(midpointsX,ymeans,yerr=ystd,capsize=10,c=colors[0],marker="x",lw=0,elinewidth=1)

# alternative visualization
```

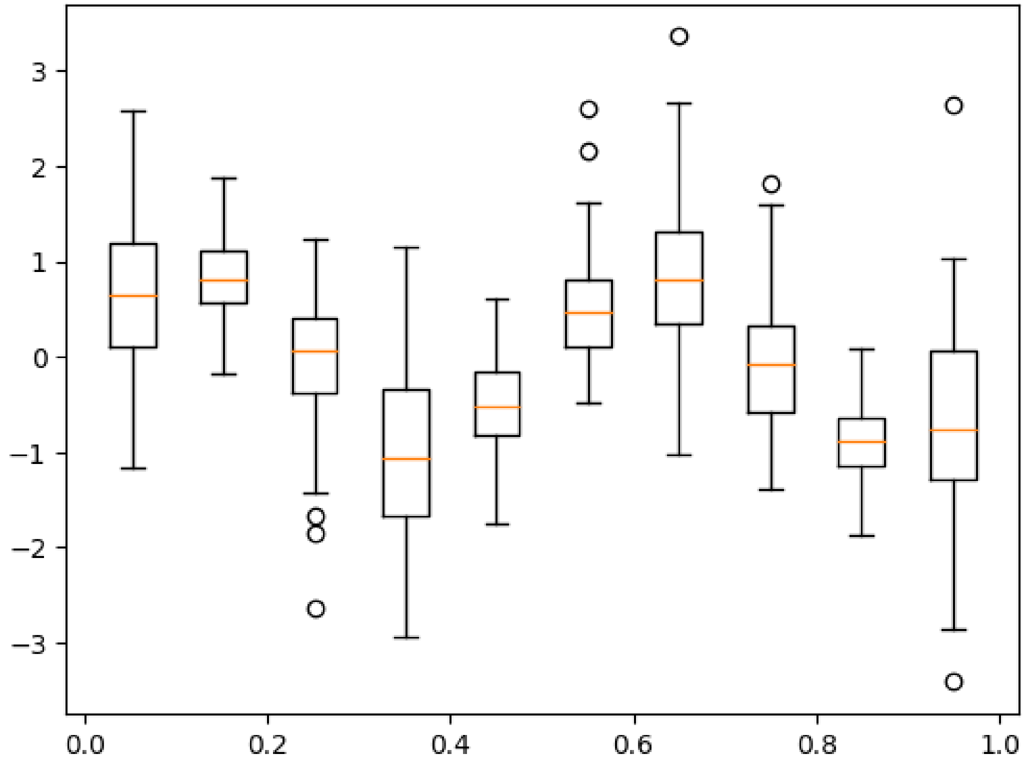
```
fig.add_subplot(1,3,2)
plt.fill_between(midpointsX,ymeans-ystd,ymeans+ystd,color="#b0b0b0",zorder=-1)
plt.plot(midpointsX,ymeans,c="r",marker="x")

# another alternative: add points
fig.add_subplot(1,3,3)
plt.fill_between(midpointsX,ymeans-ystd,ymeans+ystd,color="#b0b0b0",zorder=-1)
plt.plot(midpointsX,ymeans,marker="x",c="r")
plt.scatter(dataX,dataY,marker="x",s=1,c="k")

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



- mean and standard deviation are really only good representations if the distribution is Gaussian
- this is often true, but definitely not always

### 1.3 Box plot

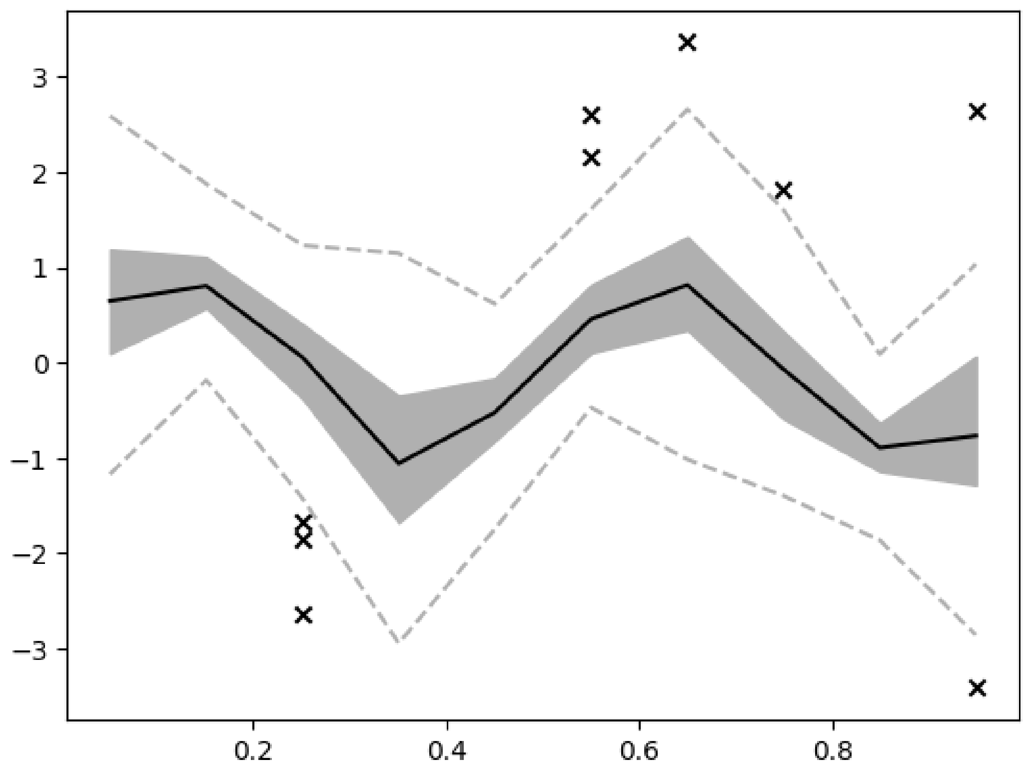
[51]:
```
plt.boxplot(binnedYLists,positions=midpointsX,manage_ticks=False,widths=0.5/
    ↪nBinsX)
plt.show()
```




[52]:
```
# manual computation of boxplot data

# compute 25,50 and 75 percentiles of each y-distribution
q=np.array([25,50,75])
qDat=np.array([np.percentile(y,q) for y in binnedYLists])
# interquartile range
IQR=qDat[:,2]-qDat[:,0]

# whisker positions
# preliminary position, based on IQR:
wlo=qDat[:,0]-1.5*IQR
whi=qDat[:,2]+1.5*IQR
# now find closest actual data points (from within the whiskers)
for i in range(len(binnedYLists)):
    y=np.array(binnedYLists[i])
    wlo[i]=np.min(y[y>wlo[i]])
    whi[i]=np.max(y[y<whi[i]])


# determine outliers
outliers=[]
for i in range(len(binnedYLists)):
```

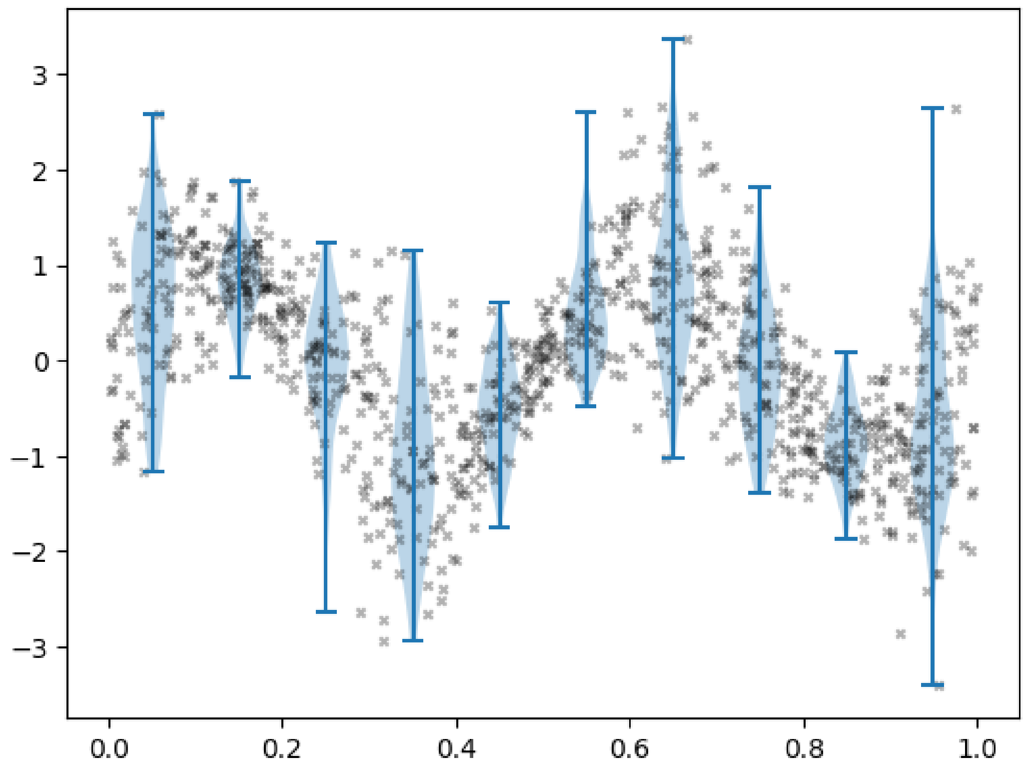
```python
    y=np.array(binnedYLists[i])
    dist=np.maximum(y-whi[i],wlo[i]-y)
    y=y[dist>0]
    for z in y:
        outliers.append([midpointsX[i],z])
outliers=np.array(outliers)
```

[53]:
```python
# this allows alternative visualization
# which might work better in case of many bins

# median line
plt.plot(midpointsX,qDat[:,1],c="k")
# IQR as filled region
plt.fill_between(midpointsX,qDat[:,2],qDat[:,0],color="#b0b0b0")
# whiskers
plt.plot(midpointsX,wlo,c="#b0b0b0",ls="dashed")
plt.plot(midpointsX,whi,c="#b0b0b0",ls="dashed")
# outliers
if len(outliers)>0:
    plt.scatter(outliers[:,0],outliers[:,1],c="k",marker="x")

plt.show()
```



### 1.4 Violin plot

[54]:
```
plt.violinplot(binnedYLists,positions=midpointsX,widths=0.5/
 ↪nBinsX,showmedians=False)
plt.scatter(dataX,dataY,marker="x",c="k",s=10,alpha=0.3)
plt.show()
```

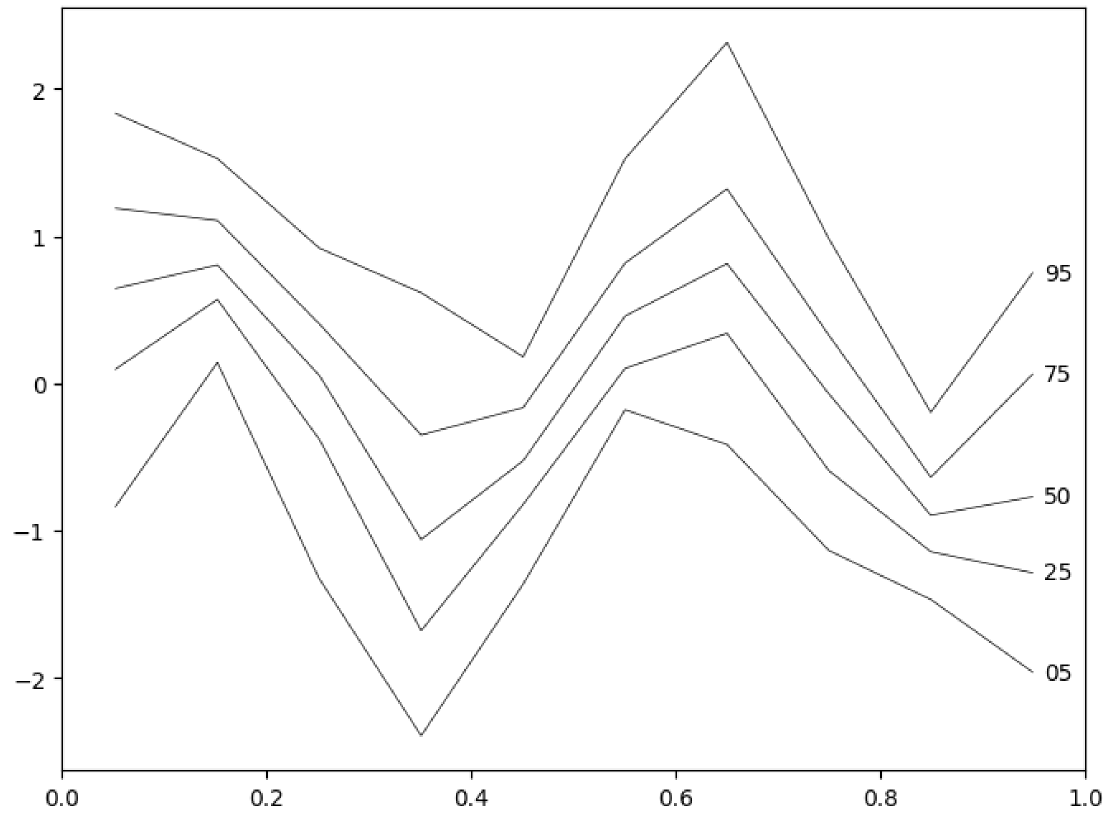



[55]:
```
# hint: check WorldPopulation Project example for 2-sided violin plot
```

### 1.5 Experiment: plot percentiles

[56]:
```
q=[5,25,50,75,95]
nQ=len(q)
qList=[np.percentile(y,q) for y in binnedYLists]
qList=np.array(qList)

fig=plt.figure(figsize=(8,6))
ax=fig.add_subplot()

for i in range(nQ):
```

```
    ax.plot(midpointsX,qList[:,i],c="k",lw=0.5)
    ax.text(midpointsX[-1]+0.01,qList[-1,i],"{:02d}".
  ↪format(int(q[i])),va="center",ha="left")
plt.xlim([0,1])
plt.show()
```




### 1.6 More example data

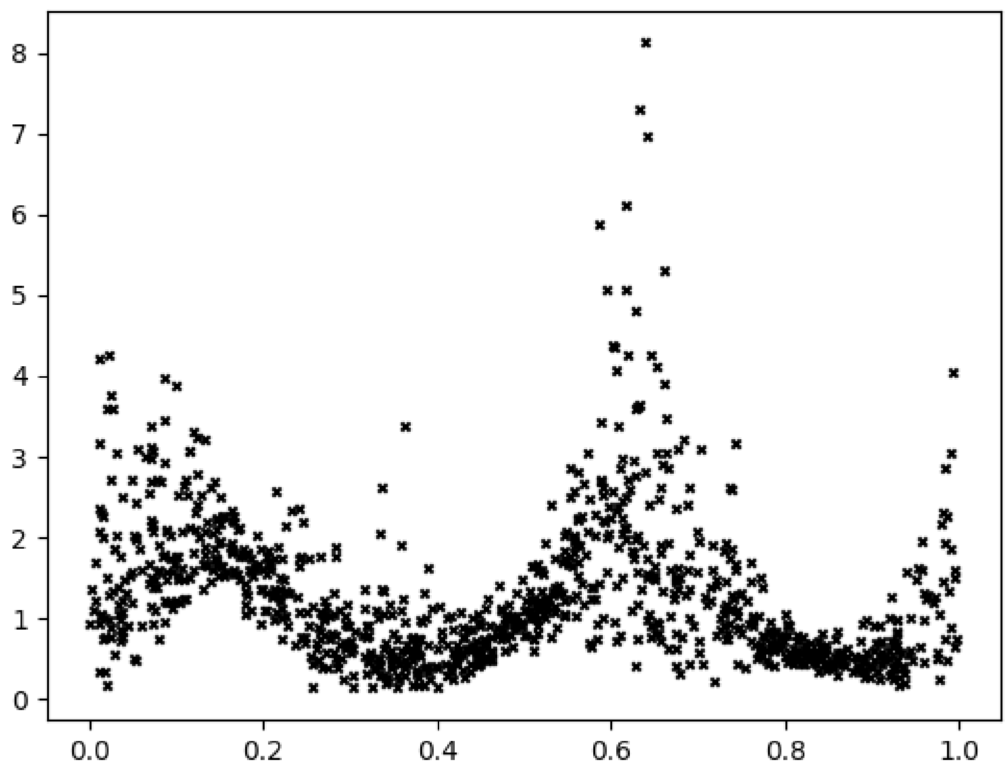
- the two examples below provide more challenging example data for which some simpler visualization methods may fail / be in appropriate
- run the corresponding cells below and then run through the cells in the Visualization section above again to see the effects

#### 1.6.1 Data example 2: Non-linearly distorted Gaussian

[57]:
```
# first generate same data as above

# nr of points
nP=1000
```

```
# X: uniform sampling from 0 to 1
dataX=np.random.random(size=(nP,))
# Y
dataMean=np.sin(2*np.pi*dataX*2)
dataStd=np.sin(2*np.pi*(dataX+0.5)*1.5)**2*0.7+0.3
dataY=dataMean+dataStd*np.random.normal(size=nP)

# now apply non-linear map, this will make distribution asymmetric
dataY=np.exp(0.7*dataY)
```

[58]:
```
plt.scatter(dataX,dataY,marker="x",c="k",s=10)
plt.show()
```

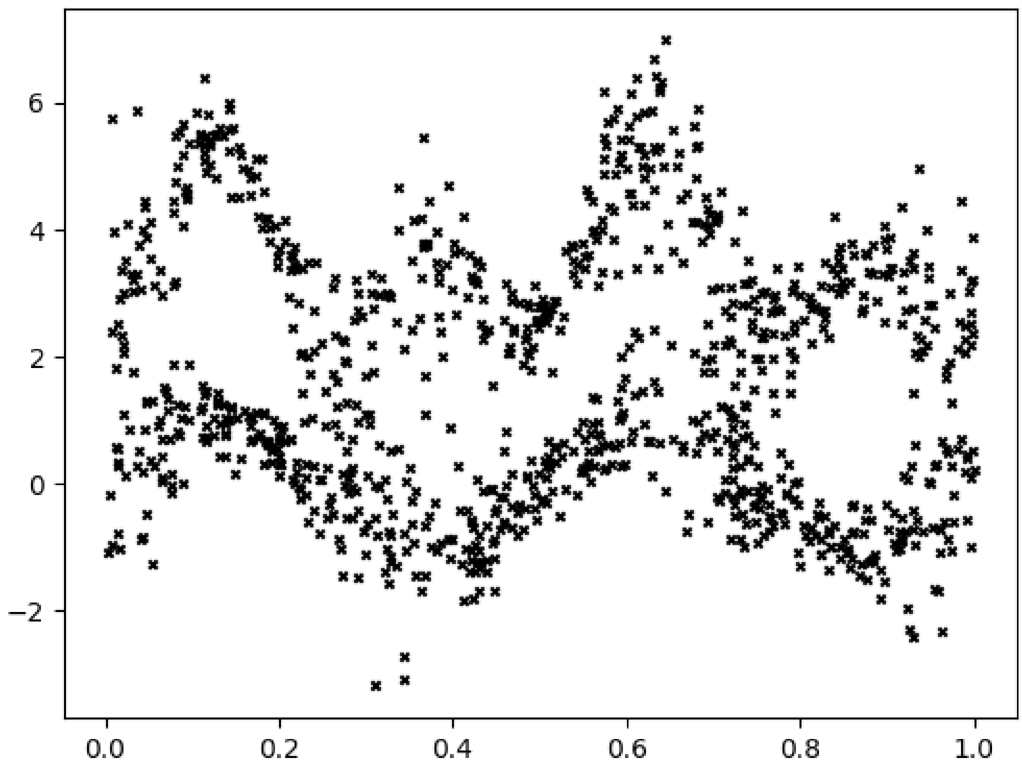



#### 1.6.2 Data example 3: Bimodal distribution

[31]:
```
# first generate same data as above

# nr of points
nP=1000
```

```python
# X: uniform sampling from 0 to 1
dataX=np.random.random(size=(nP,))
# Y
dataMean=np.sin(2*np.pi*dataX*2)
dataStd=np.sin(2*np.pi*(dataX+0.5)*1.5)**2*0.7+0.3
dataY=dataMean+dataStd*np.random.normal(size=nP)

# now add random discrete noise, i.e. the data will become bimodal
dataFlips=np.random.randint(0,2,size=(nP,))
dataFlipsSize=np.sin(2*np.pi*(dataX+0.25)*2.)**2*2.+2.5

dataY+=dataFlips*dataFlipsSize
```

[32]:
```python
plt.scatter(dataX,dataY,marker="x",c="k",s=10)
plt.show()
```




[ ]:
```python

```