In [ ]:
```
x - sum(x)/len(x)
```

Out[ ]:
```
array([-0.96666667,  1.13333333, -0.16666667])
```

Examples of functions that are not linear. The componentwise absolute value and the sort function are examples of nonlinear functions. These functions are easily computed by `abs` and `sorted`. By default, the `sorted` function sorts in increasing order, but this can be changed by adding an optional keyword argument.

In [ ]:
```
f = lambda x: abs(x) #componentwise absolute value
x = np.array([1,0])
y = np.array([0,1])
alpha = -1
beta = 2
f(alpha*x + beta*y)
```

Out[ ]:
```
array([1, 2])
```

In [ ]:
```
alpha*f(x) + beta*f(y)
```

Out[ ]:
```
array([-1,  2])
```

In [ ]:
```
f = lambda x: np.array(sorted(x, reverse = True))
f(alpha*x + beta*y)
```

Out[ ]:
```
array([ 2, -1])
```

In [ ]:
```
alpha*f(x) + beta*f(y)
```

Out[ ]:
```
array([1, 0])
```

8.2. Linear function models

Price elasticity of demand. Let's use a price elasticity of demand matrix to predict the demand for three products when the prices are changed a bit. Using this we can predict the change in total profit, given the manufacturing costs.

In [ ]:
```
p = np.array([10, 20, 15]) #Current prices
d = np.array([5.6, 1.5, 8.6]) #Current demand (say in thousands)
c = np.array([6.5, 11.2, 9.8]) #Cost to manufacture
profit = (p - c) @ d #Current total profit
```

```
print(profit)
```

```
77.51999999999998
```

In [ ]:
```
#Demand elesticity matrix
E =  np.array([[-0.3, 0.1, -0.1], [0.1, -0.5, 0.05], [-0.1, 0.05,
↪  -0.4]])
p_new = np.array([9,21,14]) #Proposed new prices
delta_p = (p_new - p)/p #Fractional change in prices
print(delta_p)
```

```
[-0.1         0.05       -0.06666667]
```

In [ ]:
```
delta_d = E @ delta_p # Predicted fractional change in demand
print(delta_d)
```

```
[ 0.04166667 -0.03833333  0.03916667]
```

In [ ]:
```
d_new = d * (1 + delta_d) # Predicted new demand
print(d_new)
```

```
[5.83333333 1.4425     8.93683333]
```

In [ ]:
```
profit_new = (p_new - c) @ d_new #Predicted new profit
print(profit_new)
```

```
66.25453333333333
```

If we trust the linear demand elasticity model, we should not make these price changes.

Taylor approximation. Consider the nonlinear function $f : \mathbf{R}^2 \to \mathbf{R}^2$ given by

$$f(x) = \begin{bmatrix} \|x-a\| \\ \|x-b\| \end{bmatrix} = \begin{bmatrix} \sqrt{(x_1-a_1)^2 + (x_2-a_2)^2} \\ \sqrt{(x_1-b_1)^2 + (x_2-b_2)^2} \end{bmatrix}.$$

The two components of f gives the distance of x to the points a and b. The function is differentiable, except when $x = a$ or $x = b$. Its derivative or Jacobian matrix is given by

$$Df(x) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1}(z) & \dfrac{\partial f_1}{\partial x_2}(z) \\ \dfrac{\partial f_2}{\partial x_1}(z) & \dfrac{\partial f_2}{\partial x_2}(z) \end{bmatrix} = \begin{bmatrix} \dfrac{z_1-a_1}{\|z-a\|} & \dfrac{z_2-a_2}{\|z-a\|} \\ \dfrac{z_1-b_1}{\|z-b\|} & \dfrac{z_2-b_2}{\|z-b\|} \end{bmatrix}.$$

Let's form the Taylor approximation of f for some specific values of a, b, and z, and then check it against the true value of f at a few points near z.

In [ ]:
```
f = lambda x: np.array([np.linalg.norm(x-a),
                        np.linalg.norm(x-b)])
Df = lambda z: np.array([(z-a)/np.linalg.norm(z-a),
                         (z-b)/np.linalg.norm(z-b)])
f_hat = lambda x: f(z) + Df(z)@(x - z)
a = np.array([1,0])
b = np.array([1,1])
z = np.array([0,0])
f(np.array([0.1,0.1]))
```

Out[ ]:
```
array([0.90553851, 1.27279221])
```

In [ ]:
```
f_hat(np.array([0.1,0.1]))
```

Out[ ]:
```
array([0.9       , 1.27279221])
```

In [ ]:
```
f(np.array([0.5,0.5]))
```

Out[ ]:
```
array([0.70710678, 0.70710678])
```

In [ ]:
```
f_hat(np.array([0.5,0.5]))
```

Out[ ]:
```
array([0.5       , 0.70710678])
```

Regression model. We revisit the regression model for the house sales data in Section 2.3. The model is

$$\hat{y} = x^T\beta + \nu = \beta_1 x_1 + \beta_2 x_2 + \nu,$$

where $\hat{y}$ is the predicted house sales price, x_1 is the house area in 1000 square feet, and x_2 is the number of bedrooms.

In the following code we construct the 2×774 data matrix X and vector of outcomes y^{d}, for the $N = 774$ examples, in the data set. We then calculate the regression model predictions $\hat{y}^{\mathrm{d}}$, the prediction error r^{d}, and the RMS prediction errors.

In [ ]:
```
# parameters in regression model
beta = [148.73, -18.85]
v = 54.40
D = house_sales_data()
```

```
yd = D['price'] # vector of outcomes
N = len(yd)
X = np.vstack((D['area'], D['beds']))
X.shape
```

Out[ ]:
```
(2, 774)
```

In [ ]:
```
ydhat = beta @ X + v; # vector of predicted outcomes
rd = yd - ydhat; # vector of predicted errors
np.sqrt(sum(rd**2)/len(rd)) # RMS prediction error
```

Out[ ]:
```
74.84571862623025
```

In [ ]:
```
# Compare with standard deviation of prices
np.std(yd)
```

Out[ ]:
```
112.78216159756509
```

8.3. Systems of linear equations

Balancing chemical reactions. We verify the linear balancing equation on page 155 of VMLS, for the simple example of electrolysis of water.

In [ ]:
```
R = np.array([2,1])
P = np.array([[2,0], [0,2]])
#Check balancing coefficients [2,2,1]
coeff = np.array([2,2,1])
coeff @ np.vstack((R, -P))
```

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