# Statistics-and-PCA

September 7, 2017

```
In [1]: using PyPlot
```

## 1 Mean and variance

Suppose we have a black box (a **distribution**) that generates data points $x_k$ (**samples**) $k = 1, \ldots,$. If we have $n$ data points, the **sample mean m** is simply the average:

$$m = \frac{1}{n} \sum_{k=1}^{\infty} x_k$$

In the limit $n \to \infty$, we get the mean $\mu$ of the underlying distribution from which the samples are generated.

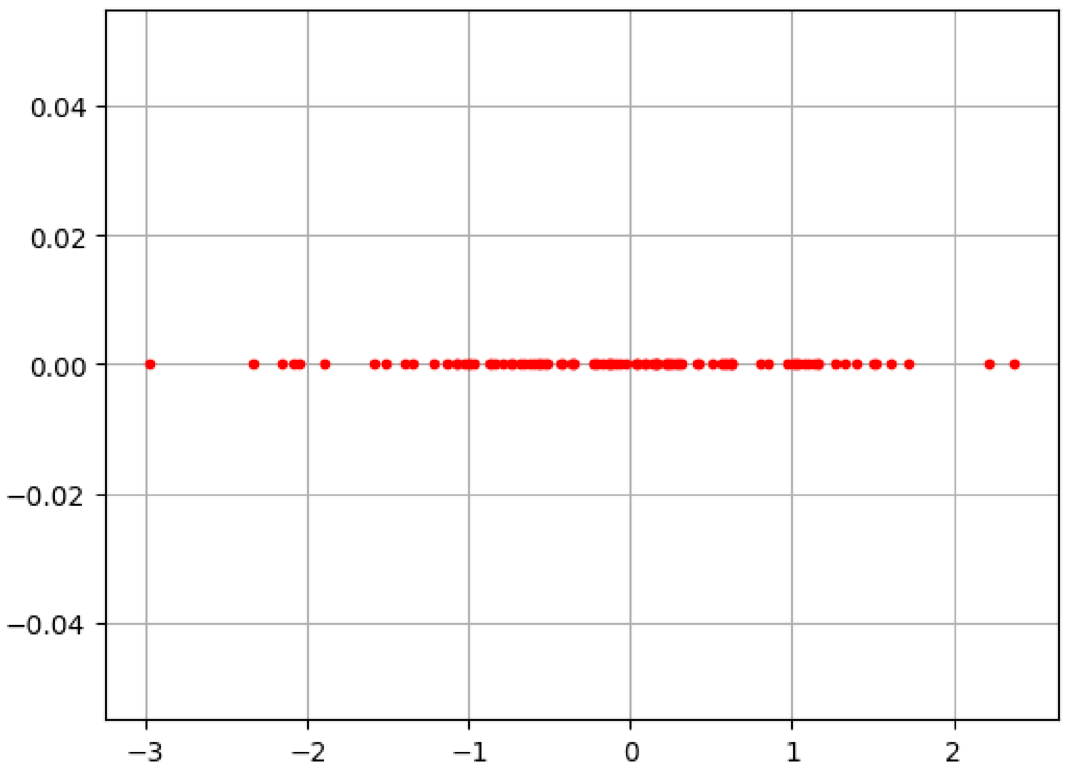
The **sample variance S**$^2$ is the mean-square deviation from the mean:

$$\text{Var}(x) = S^2 = \frac{1}{n-1} \sum_{k=1}^{\infty} (x_k - m)^2$$

where the denominator $n-1$ is Bessel's correction. The limit $n \to \infty$ of the sample variance gives $\sigma^2$, the variance of the underlying distribution, and by using $n-1$ instead of $n$ in the denominator it turns out that we get a better estimate of $\sigma^2$ when $n$ is not huge.

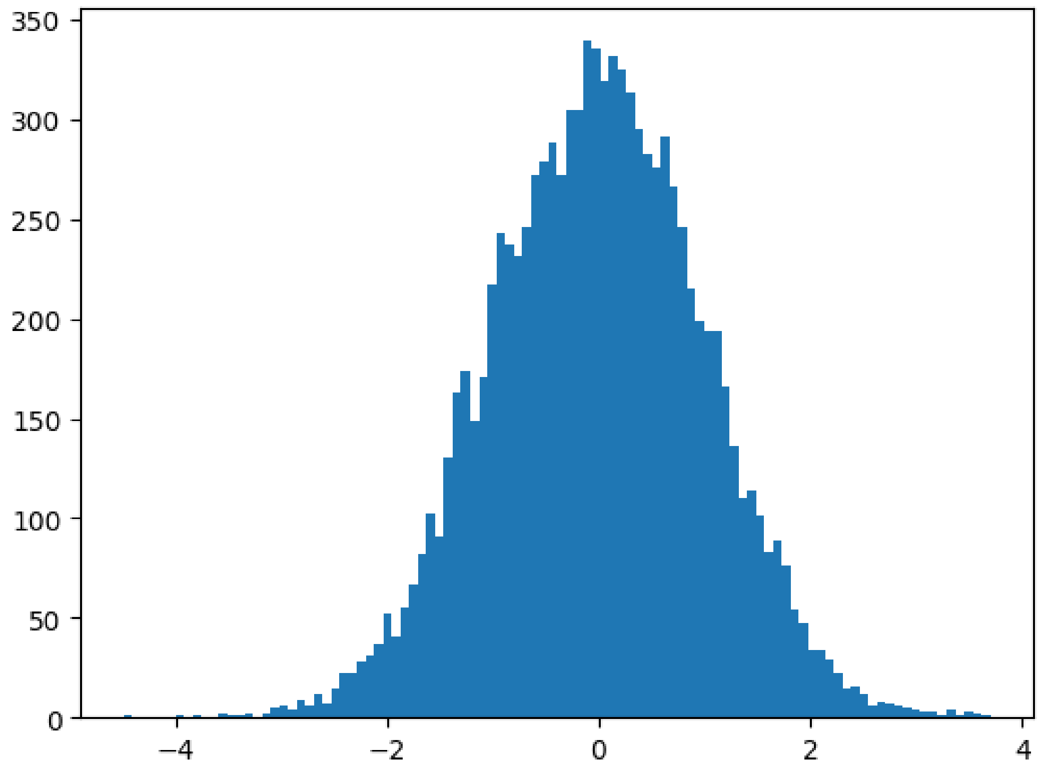
For example, the `randn()` function in Julia draws samples from a normal distribution: a Gaussian or "bell curve" with mean zero and variance 1:

```
In [2]: x = randn(100) # 100 gaussian random numbers:
        plot(x, zeros(x),"r.")
        grid()
```



It is more informative to plot a histogram:

```
In [3]: x = randn(10000)
        plt[:hist](x, bins=100);
```



The mean is the **center** of this peak and the square root $S$ of the variance is a measure of the **width** of this peak.

The mean of those 10000 samples is a pretty good estimate for the true mean (= 0) of the underlying normal distribution:

```
In [4]: mean(x)
```

```
Out[4]: -0.011081208688453203
```

The sample variance is:

```
In [5]: sum((x-mean(x)).^2)/(length(x)-1)
```

```
Out[5]: 1.0040402892748792
```

Or (equivalently but more efficient) the built-in function `var`:

```
In [6]: var(x)
```

```
Out[6]: 1.0040402892748792
```

which is a pretty good estimate for the true variance (= 1).

If we looked at more points, we would get better estimates:

```
In [7]: xbig = randn(10^7)
        mean(xbig), var(xbig)
```

```
Out[7]: (6.320131515417326e-5,0.9991775483644516)
```

### 1.1 Mean and variance in linear algebra

If we define the vector $o = (1, 1, \ldots)$ to be the vector of $n$ 1's, with $o^T o = n$, then the mean of $x$ is:

$$m = \frac{o^T x}{o^T o}$$

which is simply the **projection of x onto o**. And the sample variance is

$$\text{Var}(x) = \frac{\|x - mo\|^2}{n-1} = \frac{\left\|\left(I - \frac{oo^T}{o^T o}\right) x\right\|^2}{n-1}$$

is the **length$^2$ of the projection of x [U+27C2] to o** divided by $n - 1$.

In fact, the $n - 1$ denominator is closely related to the fact that this orthogonal projection "lives" in an $n-1$ dimensional space: after you subtract off the mean, there are only $n-1$ degrees of freedom left. (This is just a handwaving argument; for more careful derivations of this denominator, see e.g. Bessel's correction on Wikipedia.)

## 2 Covariance and Correlation

A key question in statistics is whether/how two sets of data are **correlated**. If you have two variables $x$ and $y$, do they tend to "move together"?

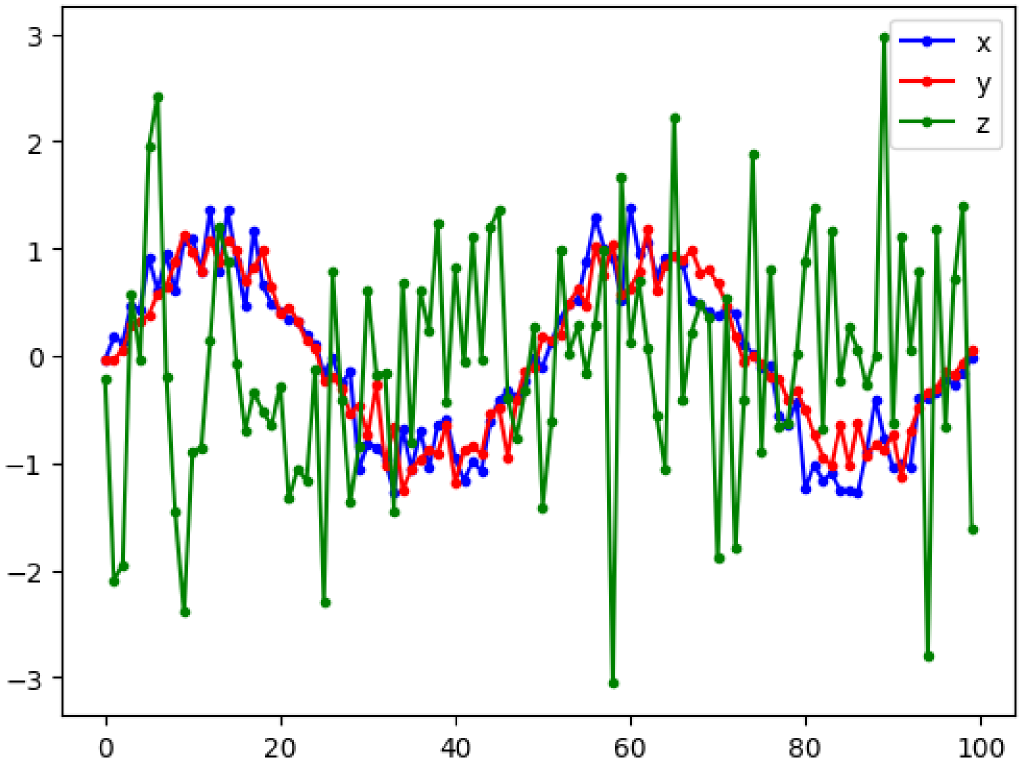
An intuitive measure for this is: **when x is greater/less than its mean, is y *also* greater/less than its mean?** Translated into math, this leads to the **covariance**:

$$\text{Covar}(x, y) = \frac{1}{n-1} \sum_{k=1}^{n} (x_k - \text{mean}(x))(y_k - \text{mean}(y)) = \frac{(Px)^T (Py)}{n-1} = \frac{x^T P y}{n-1}$$

where $P = I - \frac{oo^T}{o^T o}$ is the projection operator from above that subtracts the mean from a vector (i.e. it projects vectors onto the subspace of vectors with zero mean). (In the last step we used the facts that $P^T = P$ and $P^2 = P$.)

For example, here are plots of two very correlated vectors x and y of data and a third data set z that is just independent random numbrers

```
In [8]: x = sin(linspace(0,4π,100) + randn(100)*0.1) .* (1 + 0.3*randn(100))
        y = sin(linspace(0,4π,100) + randn(100)*0.1) .* (1 + 0.3*randn(100))
        z = randn(100)
        plot(x, "b.-")
        plot(y, "r.-")
        plot(z, "g.-")
        legend(["x","y","z"])
```



```
Out[8]: PyObject <matplotlib.legend.Legend object at 0x3290bc690>
```

All three have mean nearly zero:

```
In [9]: mean(x),mean(y),mean(z)
```

```
Out[9]: (-0.020601421221630608,-0.0011893930679909397,-0.06415236608580241)
```

But the covariance of x and y is totally different from the covariance of x and z:

```
In [10]: # A simple covariance function.  See https://github.com/JuliaStats/StatsBase.jl for
         # better statistical functions in Julia.
         covar(x,y) = dot(x - mean(x), y - mean(y)) / (length(x) - 1)
```

```
Out[10]: covar (generic function with 1 method)
```

```
In [11]: covar(x,y), covar(x,z)
```

```
Out[11]: (0.4940689175964578,-0.07312995881233886)
```

The variance and covariance have the units of the data squared. I can make the covariance of x and y smaller simply by dividing y by 10, which doesn't see like a good measure of how correlated they are.

Often, it is nicer to work with a dimensionless quantity *independent* of the vector lengths, the **correlation**:

$$\text{Cor}(x, y) = \frac{\text{Covar}(x, y)}{\sqrt{\text{Var}(x) \times \text{Var}(y)}} = \frac{(Px)^T(Py)}{\|Px\| \times \|Py\|}$$

This is just the dot product of the vectors (after subtracting their means) divided by their lengths.

It turns out that Julia has a built-in function `cor` that compute precisely this:

```
In [12]: covar(x,y) / sqrt(covar(x,x) * covar(y,y)) # correlation, manually computed
```

```
Out[12]: 0.935143359379067
```

```
In [13]: cor(x,y)
```

```
Out[13]: 0.9351433593790669
```

```
In [14]: cor(x,z)
```

```
Out[14]: -0.08605379904360413
```

```
In [15]: abs(cor(x,y)/cor(x,z))
```

```
Out[15]: 10.866961944413662
```

Now that we've scaled out the overall length of the vectors, we can sensibly compare the correlation of x,y with the correlation of x,z, and we see that the former are more than 10x the correlation of the latter in this sample

## 3 The covariance and correlation matrices

If we have a bunch of data sets, we might want the covariance or correlation of *every* pair of data sets. Since these are basically dot products, asking for *all* of the dot products is the same as asking for a **matrix multiplication**.

In particular, suppose that $X$ is the $m \times n$ matrix whose **rows** are $n$ different datasets of length $n$. First, we need to subtract off the means of each row to form a new matrix $A$:

$$A = XP = XP^T = (PX^T)^T$$

where $P$ is the projection matrix from above that subtracts the mean. Multiplying by $P^T = P$ on the *right* corresponds to projecting each *row* of $X$.

Given $A$, we can compute **all** of the covariances simply by computing the **covariance matrix S**

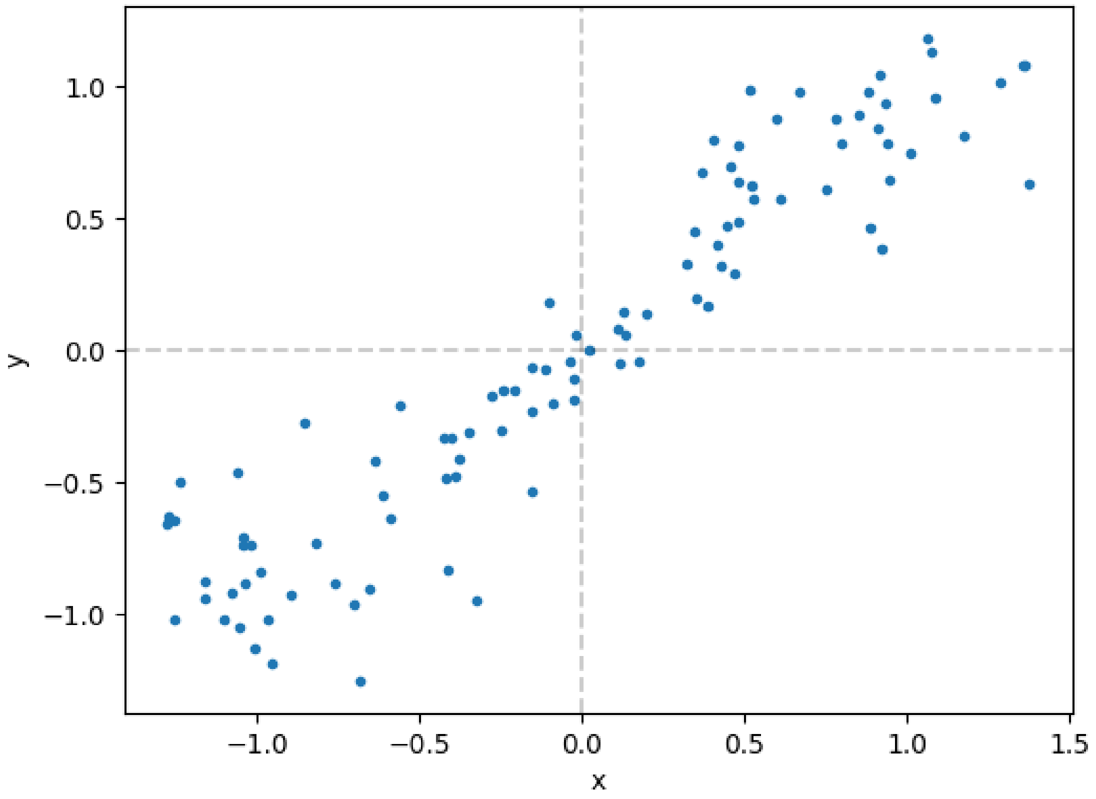
$$S = \frac{AA^T}{n-1}$$

since $AA^T$ computes all of the dot products of all of the rows of $A$. The diagonal entries of $S$ are the variances of each dataset, and the off-diagonal elements are the covariances.

Alternatively, we can compute the **correlation matrix** $C = \hat{A}\hat{A}^T$, where $\hat{A}$ is simply the matrix $A$ scaled so that each row has unit length. i.e. $\hat{A} = DA$, where $D$ is a diagonal matrix whose entries are the inverse of the length of each row, i.e. $D$ is the inverse of the diagonal entries of $AA^T$.

Let's look in more detail at the two correlated vectors $x$ and $y$ from above:

```
In [16]: plot(x,y, ".")
         xlabel("x")
         ylabel("y")
         axhline(0, linestyle="--", color="k", alpha=0.2)
         axvline(0, linestyle="--", color="k", alpha=0.2)
```



```
Out[16]: PyObject <matplotlib.lines.Line2D object at 0x329302910>
```

The correlation matrix is:

```
In [17]: A = [x' - mean(x); y' - mean(y)] # rows are x and y with means subtracted
```

```
Out[17]: 2×100 Array{Float64,2}:
          -0.0136799   0.194767   0.153322   …  -0.255841  -0.131478   0.00542467
          -0.0379629  -0.0412615  0.0632018     -0.169123  -0.0637059  0.0591737
```

```
In [18]: S = A * A' / (length(x)-1)
```

```
Out[18]: 2×2 Array{Float64,2}:
          0.579931  0.494069
          0.494069  0.481329
```

In this case, since there are only two datasets, $S$ is just a $2 \times 2$ matrix.

## 4 PCA: diagonalizing the covariance matrix

A key question in analyizing data analysis is to figure out **which variables are responsible for most of the variation in the data**. These may not be the variables you measured, but may instead be some **linear combination of the measured variables**!

Mathematically, this corresponds to **diagonalizing the covariance (or correlation) matrix**:

- $S$ is real-symmetric and positive-definite (or at least semidefinite), so diagonalization $S = Q\Lambda Q^T$ finds real, positive eigenvalues $\lambda_k = \sigma_k^2 \geq 0$ and an **orthonormal basis Q of eigenvectors**.
- The eigenvectors form a **coordinate system** in which the covariance matrix **S becomes diagonal**, i.e. a **coordinate system in which the variables are uncorrelated**.
- The diagonal entries in this coordinate system, the eigenvalues, are the **variances of these uncorrelated components**.

This process of diagonalizing the covariance matrix is called principal component analysis, or **PCA**. Let's try it:

```
In [19]: σ², Q = eig(S)
```

```
Out[19]: ([0.0341076,1.02715],
          [0.671084 -0.741381; -0.741381 -0.671084])
```

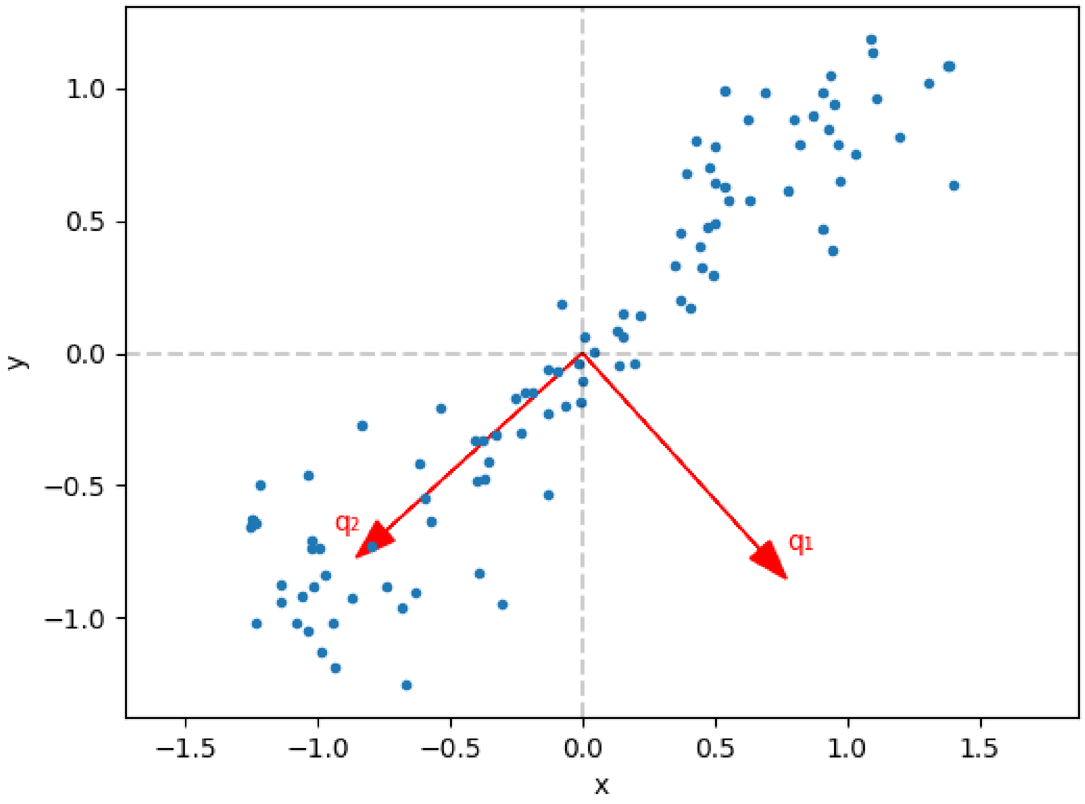
We see that the second eigenvector is responsible for almost all of the variation in the data, because its eigenvalue (the variance $\sigma^2$) is much larger:

```
In [20]: σ²
```

```
Out[20]: 2-element Array{Float64,1}:
          0.0341076
          1.02715
```

Let's plot these two eigenvectors on top of our data:

```
In [21]: plot(x-mean(x),y-mean(y), ".")
         xlabel("x")
         ylabel("y")
         axhline(0, linestyle="--", color="k", alpha=0.2)
         axvline(0, linestyle="--", color="k", alpha=0.2)
         arrow(0,0, Q[:,1]..., head_width=0.1, color="r")
         arrow(0,0, Q[:,2]..., head_width=0.1, color="r")
         text(Q[1,1]+0.1,Q[2,1], "q₁", color="r")
         text(Q[1,2]-0.2,Q[2,2], "q₂", color="r")
         axis("equal")
```



```
Out[21]: (-1.3857420714936417,1.5314737916078676,-1.371642469431078,1.3080133686711006)
```

The $q_2$ direction, corresponding to the biggest eigenvalue of the covariance matrix, is indeed the direction with the biggest variation in the data!

In this case, it is along the (1,1) direction because $x$ and $y$ tend to move together.

The other direction $q_1$ is the other uncorrelated (= orthogonal) direction of variation in the data. Not much is going on in that direction.

(In fact, this $q_2$ can be viewed as a kind of "best fit" line which the Strang book calls perpendicular least squares, and is also called Deming regression.)

## 5 PCA and the SVD

Instead of forming $AA^T$ and diagonalizing that, we can equivalently (in the absence of roundoff errors) us the singular value decomposition (SVD) of $A/\sqrt{n-1}$. Recall the SVD

$$\frac{A}{\sqrt{n-1}} = U\Sigma V^T$$

where $U$ and $V$ are orthogonal matrices and $\Sigma = \begin{pmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \ddots \end{pmatrix}$ is a diagonal $m \times n$ matrix of the singular values $\sigma_k$.

Then, if we compute the covariance matrix , we get:

$$S = \frac{AA^T}{n-1} = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma V^T V\Sigma^T U^T = U\Sigma\Sigma^T U^T$$

Since $\Sigma\Sigma^T$ is a diagonal matrix of the *squares* $\sigma_k^2$ of the singular values $\sigma_k$, we find:

- The squares $\sigma_k^2$ of the singular values are the variances of the uncorrelated components of the data (the eigenvalues of $S$).
- The left singular vectors $U$ are **precisely** the orthonormal eigenvectors of $AA^T$, i.e. the uncorrelated components of the data.

In practice, PCA typically uses the SVD directly rather than explicitly forming the covariance matrix $S$. (It turns out that computing $AA^T$ explicitly exacerbates sensitivity to rounding errors and other errors in $A$.)

```
In [22]: U, σ, V = svd(A / sqrt(length(x)-1))
```

```
Out[22]: (
         [-0.741381 -0.671084; -0.671084 0.741381],

         [1.01349,0.184682],
         [0.00353214 -0.0103205; -0.0115734 -0.0877767; ... ; 0.0139059 0.0223135; -0.00433678 0.021893
```

```
In [23]: σ.^2
```

```
Out[23]: 2-element Array{Float64,1}:
          1.02715
          0.0341076
```

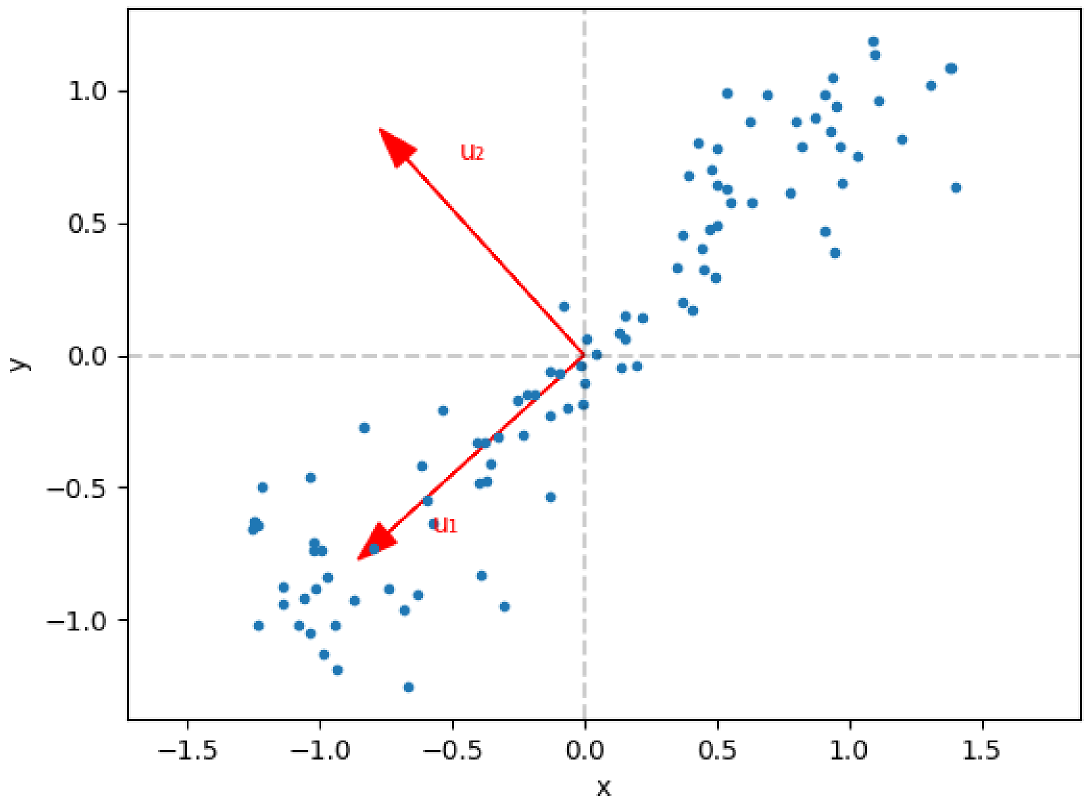
As promised, this is the same as the eigenvalues of $S$ from above.

Conveniently, the convention for the SVD is to sort the singular values in descending order $\sigma_1 \geq \sigma_2 \geq \cdots$. So, the **first** singular value/vector represents *most* of the variation in the data, and so on.

```
In [24]: U
```

```
Out[24]: 2×2 Array{Float64,2}:
          -0.741381  -0.671084
          -0.671084   0.741381
```

```
In [25]: plot(x-mean(x),y-mean(y), ".")
         xlabel("x")
         ylabel("y")
         axhline(0, linestyle="--", color="k", alpha=0.2)
         axvline(0, linestyle="--", color="k", alpha=0.2)
         arrow(0,0, U[:,1]..., head_width=0.1, color="r")
         arrow(0,0, U[:,2]..., head_width=0.1, color="r")
         text(U[1,2]+0.1,U[2,1], "u₁", color="r")
         text(U[1,2]+0.2,U[2,2], "u₂", color="r")
         axis("equal")
```



```
Out[25]: (-1.3857420714936417,1.5314737916078676,-1.371642469431078,1.3080133686711006)
```

There is an irrelevant sign flip from before (the signs of the eigenvectors and singular vectors are arbitrary), but otherwise it is the same.