Section 7: Eigendecomposition and Principal Component Analysis with NumPy and SciPy

Eigendecomposition

Another ultra-powerful linear algebra tool in NumPy and SciPy is ***eigendecomposition***. Don't know what this is? Sure you've heard of it, because it is closely related to both SVD and to this magical thing called *principal component analysis* which we will get to in a second.

Summery version: A matrix **A** applied to column vector **x**, that is **Ax**, is a linear transformation of **x**.

There is a special transform in the following form: **Ax** = λ**x**

where **A** is nxn matrix, x is nx1 column vector ($X \neq 0$), and λ is some scalar. Any λ that satisfies the above equation is known as an *eigenvalue* of the matrix **A**, while the associated vector **x** is called an *eigenvector* corresponding to λ.

Not quite clear? Here's some more detail.

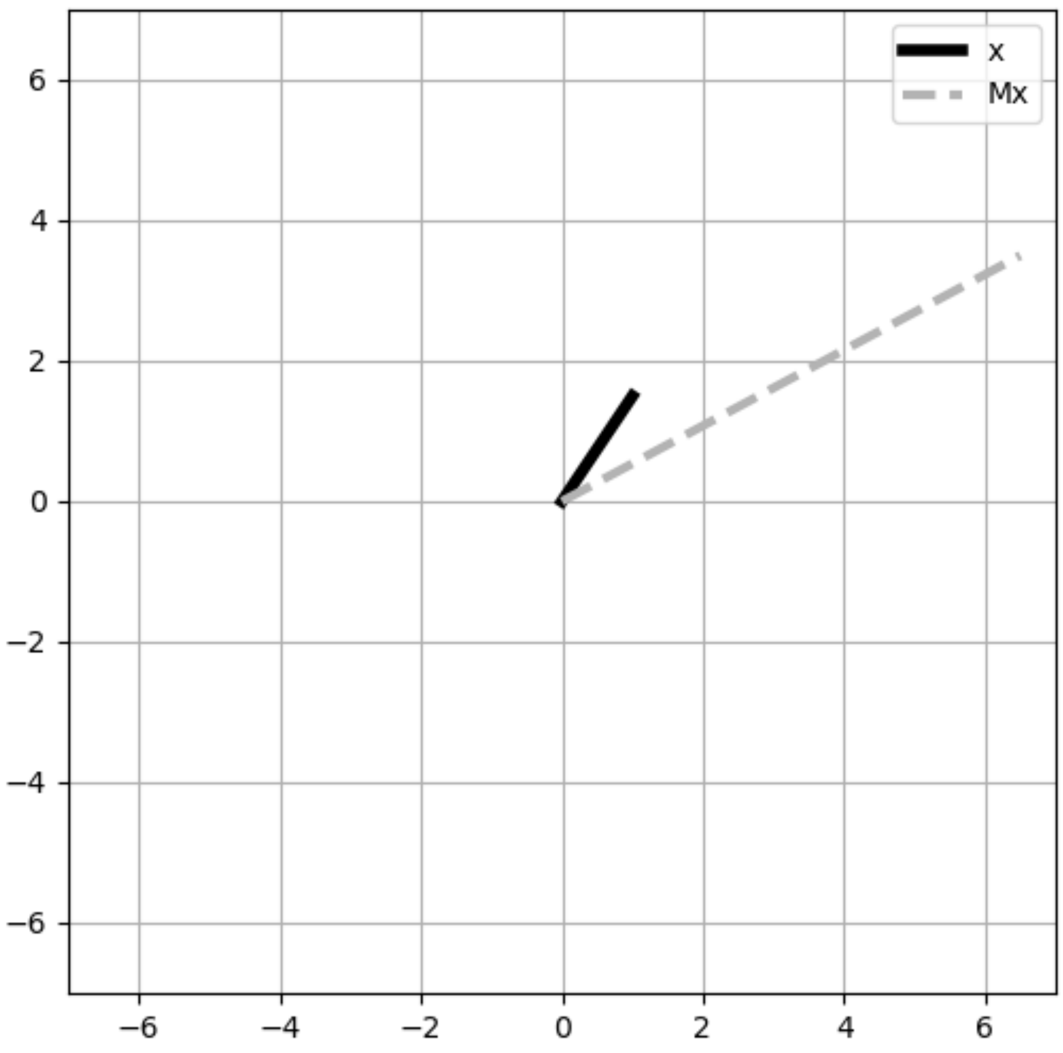
Motivation

When we think of a think of a vector, we think of a geometric line with some direction and magnitude.

We know that a vector x can be transformed to a different vector by multiplying by matrix A -> Ax. The effect of the transformation represents a scale of the length of the vector and/or the rotate of the vector.

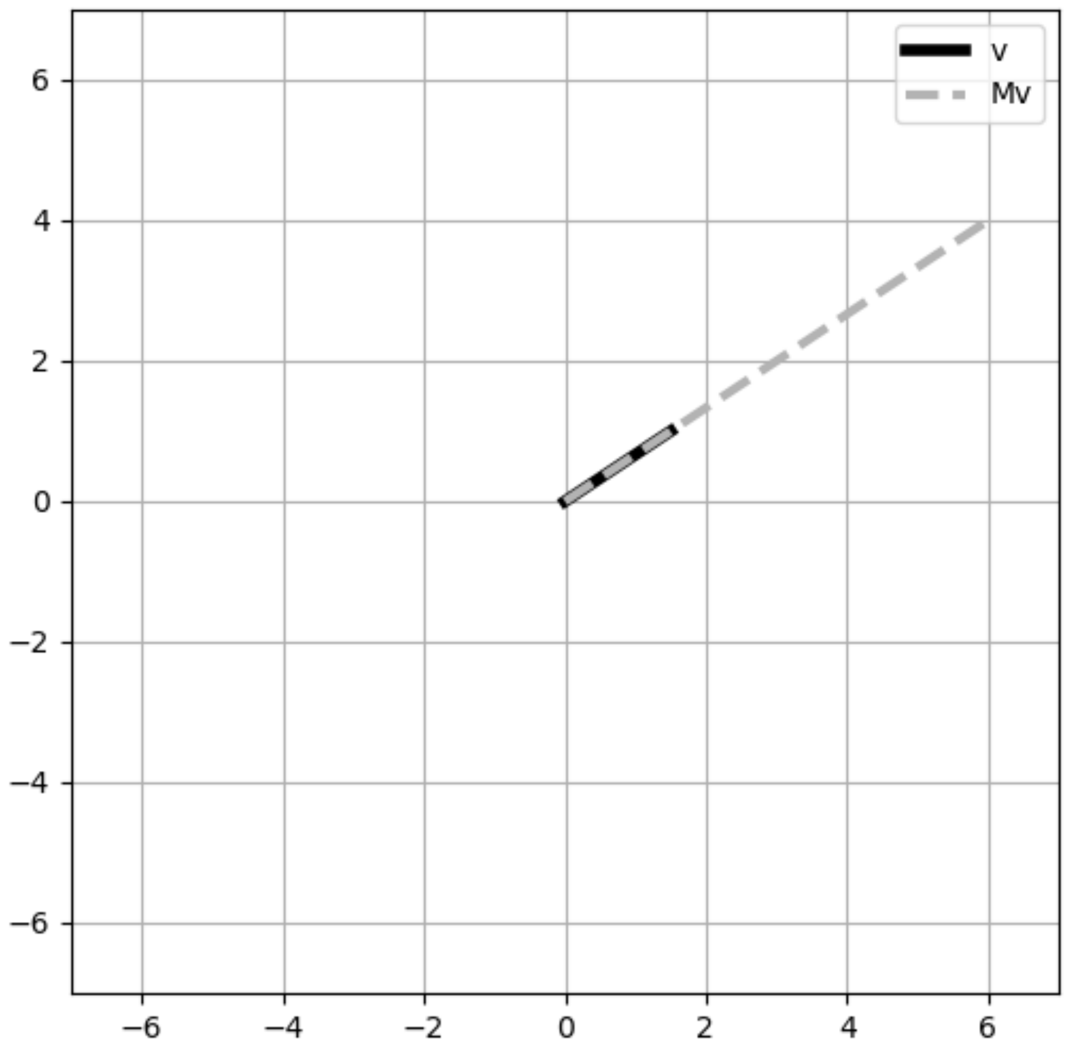
Matrix-vector multiplication can be thought of then as a way of rotating and scaling that vector. For some vectors, the effect of transformation of Ax is only scale (stretching, compressing, and flipping). Here's a simple way of visualizing:

```
import numpy as np
import matplotlib.pyplot as plt

# some matrix
M  = np.array([ [2,3],[2,1] ])
x  = np.array([ [1,1.5] ]).T # transposed into a column vector!
Mx = M.dot(x)


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

plt.plot([0,x[0,0]],[0,x[1,0]],'k',linewidth=4,label='x')
plt.plot([0,Mx[0,0]],[0,Mx[1,0]],'--',linewidth=3,color=[.7,.7,.7],label=
'Mx')
plt.xlim([-7,7])
plt.ylim([-7,7])
plt.legend()
plt.grid()
plt.show()
```



Now, let's try a different vector with the same matrix: e.g. just swapping the x and y positions:

```
M  = np.array([ [2,3],[2,1] ])
v  = np.array([ [1.5,1] ]).T # transposed into a column vector!
Mv = M.dot(v)


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

plt.plot([0,v[0,0]],[0,v[1,0]],'k',linewidth=4,label='v')
plt.plot([0,Mv[0,0]],[0,Mv[1,0]],'--',linewidth=3,color=[.7,.7,.7],label=
'Mv')
plt.xlim([-7,7])
plt.ylim([-7,7])
plt.legend()
plt.grid()
plt.show()
```



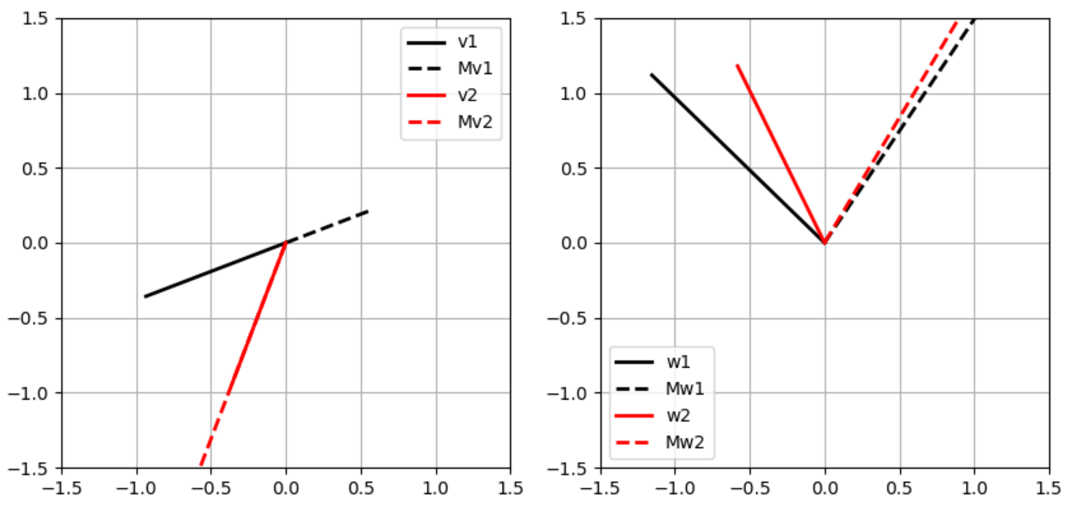
The above examples are from `def ex1` in `eigendecomp.py`.

Note what is different. The matrix-vector product is no longer rotated. The matrix is still scaled it has the same direction as the input vector v. I.e. this matrix-vector multiplication acts *as if* it were a scalar-vector multiplication (taking a vector and just multiplying the entries). In this case, v is then an *eigenvector* of the matrix M. This is our introduction to *eigendecomposition*.

Basics of Eigendecomposition

Okay, so what is eigendecomposition? The above example gives a geometric definition: an *eigenvector* of a matrix is one where a matrix *stretches* but does not rotate a vector. The amount of stretching done is then the *eigenvalue*.

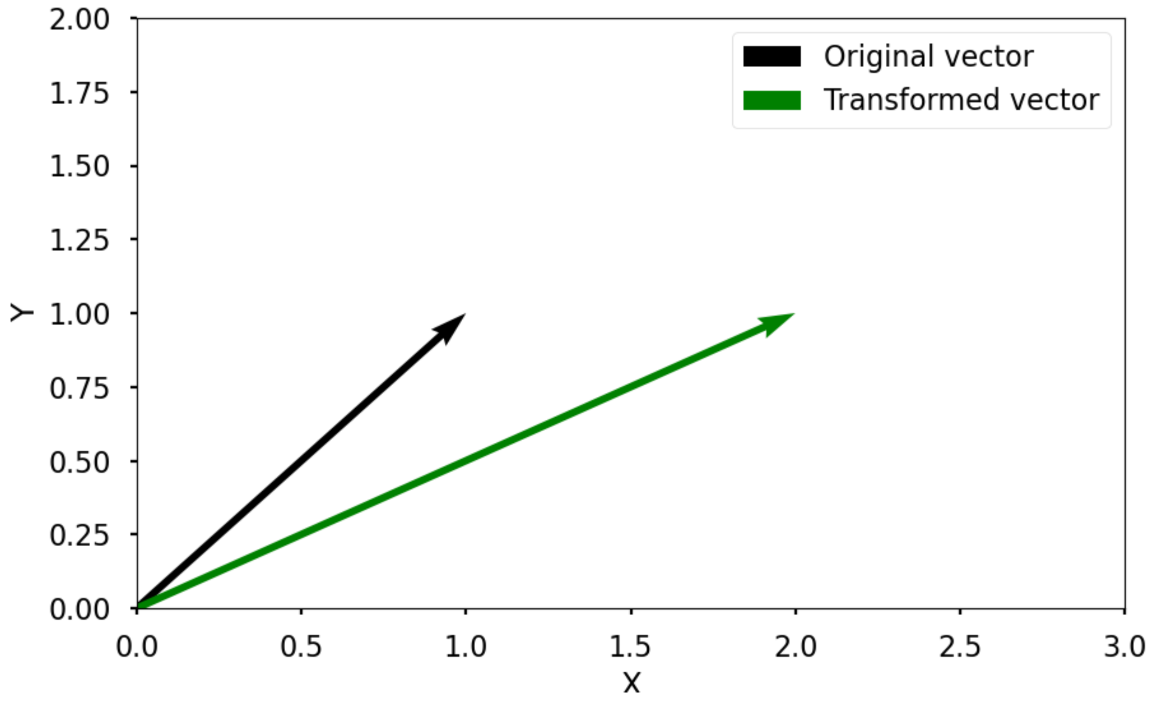
The figures below show vectors before and after multiplication by a 2x2 matrix from the source code in `def ex2()`. The two vectors on the left are eigenvectors whereas the two vectors in the right plot are not eigenvectors. The eigenvectors point in the same direction before and after being multiplied by the matrix.



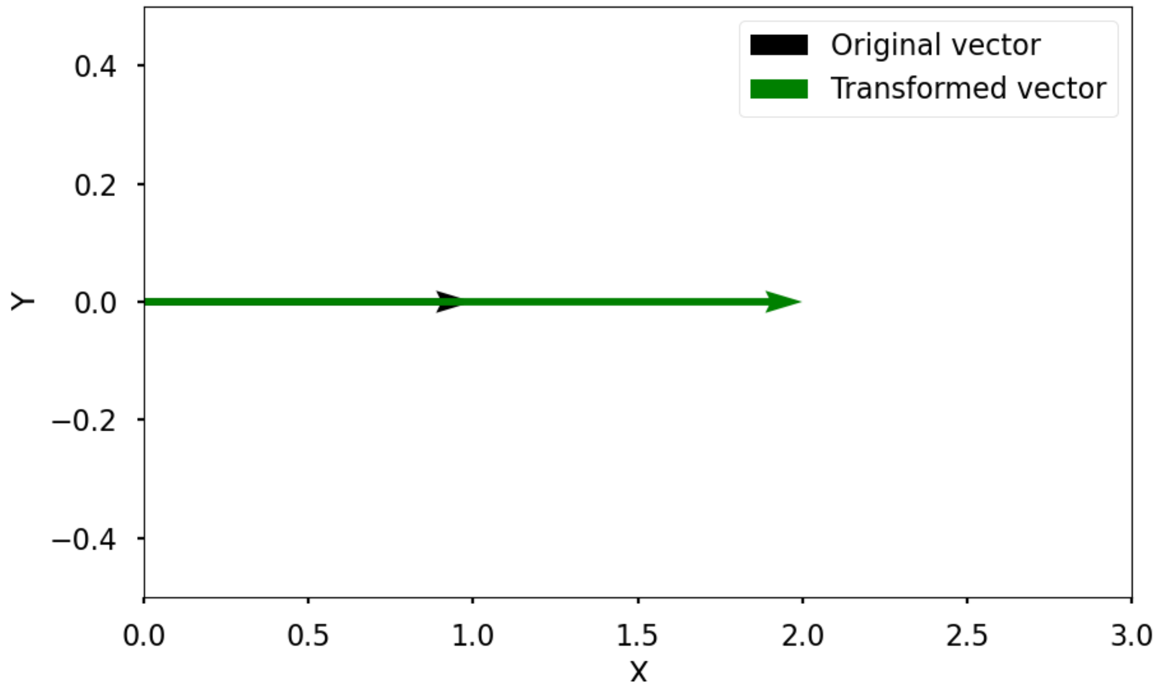
Here's another example from `def ex3()`

```
def ex3():
 plt.style.use('seaborn-poster')

 A = np.array([[2, 0],[0, 1]])

 x = np.array([[1],[1]])
 b = np.dot(A, x)
 plot_vect(x,b,(0,3),(0,2))

#different version
 x = np.array([[1], [0]])
 b = np.dot(A, x)

 plot_vect(x,b,(0,3),(-0.5,0.5))

def plot_vect(x, b, xlim, ylim):
    '''
    function to plot two vectors,
    x - the original vector
    b - the transformed vector
    xlim - the limit for x
    ylim - the limit for y
    '''
    plt.figure(figsize = (10, 6))
    plt.quiver(0,0,x[0],x[1],\
        color='k',angles='xy',\
        scale_units='xy',scale=1,\
        label='Original vector')
    plt.quiver(0,0,b[0],b[1],\
        color='g',angles='xy',\
        scale_units='xy',scale=1,\
        label ='Transformed vector')
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.show()
```

We get the following:



We can see from the generated figure that the original vector x is rotated and stretched longer after transformed by A. The vector [[1], [1]] is transformed to [[2], [1]]. Let's try to do the same exercise with a different vector [[1], [0]].



with this new vector, the only thing changed after the transformation is the length of the vector, it is stretched. The new vector is [[2], [0]], therefore, the transform is

$$Ax=2x$$

Eigendecomposition works for square matrices. Like SVD, it often crops up with correlation or covariance matrices (more on that later). Every square matrix of dimensions MxM has M eigenvalues (scalars) and M corresponding eigenvectors. Eigendecomposition then computes these scalar-vector pairs.

The main eigenvalue equation is very simple:

$$\mathbf{A}\mathbf{v}=\lambda\mathbf{v}.$$

To be clear this is not saying that the matrix **A** is the same thing as a scalar λ. Rather, *the effect or consequence of the matrix operating on the vector* is the same as the effect of the scalar operating on that same vector.

Finding Eigenvalues and Eigenvectors

So how can we find eigenvalues and eigenvectors? Thankfully, NumPy makes this very easy with `np.linalg.eig`. E.g.

```
matrix=np.array([[1,2],[3,4]])
evals,evecs=np.linalg.eig(matrix)

evals
#array([-0.37228132,  5.37228132])

evecs
#array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Here's another example:

```
a = np.array([[2, 2, 4],
              [1, 3, 5],
              [2, 3, 4]])

w,v=np.linalg.eig(a)

print('E-value:', w)
#E-value: [ 8.80916362  0.92620912 -0.73537273]

print('E-vector', v)
#E-vector [[-0.52799324 -0.77557092 -0.36272811]
 [-0.604391    0.62277013 -0.7103262 ]
 [-0.59660259 -0.10318482  0.60321224]]
```

And we can do the same thing with SciPy

```
from scipy import linalg
matrix=np.array([[1,2],[3,4]])
evals,evecs=linalg.eig(matrix)

evals
#array([-0.37228132,  5.37228132])

evecs
#array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Okay fine, we can compute eigenvalues and eigenvectors. So? We gain a bit more insight if we rearrange the eigenvalue equation:

$\mathbf{A}\mathbf{v}=\lambda\mathbf{v}$

$\mathbf{A}\mathbf{v}-\lambda\mathbf{v}=0$

$(\mathbf{A}-\lambda\mathbf{I})\mathbf{v}=0$

Note the imposition of the identity matrix $\mathbf{I}$ (A matrix minus a scalar is not a thing/wrong dimensionality: we have to operate element-wise).

And then ...

$\tilde{\mathbf{A}} = \mathbf{A}-\lambda\mathbf{I}$, so

$\tilde{\mathbf{A}}\mathbf{v} = 0$

In other words, the eigenvector is in the nullspace of the matrix shifted by its eigenvalue. A matrix shifted by its eigenvalue is singular, because only singular matrices have non-trivial (e.g. v=0) null space. Because singular matrices have determinants of zero, $|\mathbf{A}$-$\lambda\mathbf{I}|$ =0. I.e. we shift a matrix by its (unknown) eigenvalue λ, set its determinant to zero and solve for λ. For the case of a simple 2x2 matrix with elements $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, shifting by eigenvalues λ leads to the equation:

$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ - λ $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, or $\lambda^2 - (a+d)\lambda + (ad - bc)$ = 0. I.e. this is a second order polynomial equation with thus two solutions. A 3x3 matrix? The leading term is λ^3 and so on. In other words, the characteristic polynomial of an MxM matrix will have λ^M term and a MxM matrix will have M eigenvalues.

Okay, great but where do the eigenvectors actually come from?

Here's a numerical example. The following is a matrix and its eigenvalues:

$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$, λ_1 = 3, λ_2 = -1.

Take the case of λ_1. To reveal its eigenvector, shift the matrix by λ_1 and find a vector in its null space:

$\begin{bmatrix} 1-3 & 2 \\ 2 & 1-3 \end{bmatrix}$ = $\begin{bmatrix} -2 & 2 \\ 2 & -2 \end{bmatrix}$ $\begin{bmatrix} x1 \\ x2 \end{bmatrix}$ = $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$

We are left with $\begin{bmatrix} -2 * x1 + 2 * x2 \\ 2 * x1 - 2 * x2 \end{bmatrix}$ = $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Here, x1=1 and x2=1 work just fine.

This means that [1 1] is an eigenvector of the matrix associated with an eigenvalue of 3. Now this was a really trivial example. In practice, these vectors are found by using Gauss-Jordan elimination to solve a system of equations. But we don't need to worry about that for this class.

Diagonalizing a Square Matrix

In the example above, the eigenvalue equation lists one eigenvalue and one corresponding eigenvector. But we can do a lot better than that. Specifically, it follows that an M x M matrix as M eigenvalue equations:

$\mathbf{A}\mathbf{v}_1 = \lambda_1 \mathbf{v}_1$

...

$\mathbf{A}\mathbf{v}_M = \lambda_M \mathbf{v}_1$

Now, this series of equations is a bit ugly. So we can rewrite it in a more succinct manner. Instead of storing hte eigenvalues in a vector, we stor the eigenvalues in teh diagonal of a matrix Λ

I.e. the righthand side of the previous series of equations can be summarized as ...

$$\begin{bmatrix} v11 & v12 & v13 \\ v12 & v22 & v23 \\ v13 & v32 & v33 \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} = \begin{bmatrix} \lambda_1 v11 & \lambda_2 v12 & \lambda_3 v13 \\ \lambda_1 v12 & \lambda_2 v22 & \lambda_3 v23 \\ \lambda_1 v13 & \lambda_2 v32 & \lambda_3 v33 \end{bmatrix}$$

More generally, then, the matrix-eigenvalue equation can be written as

$\mathbf{AV} = \mathbf{V}\Lambda.$

This result then has the following consequences (these are equivalent statements produced by inverting matrices or multiplying both sides by the same matrix):

$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$

$\Lambda = \mathbf{V}\mathbf{A}\mathbf{V}^{-1}$

The first equation shows that **A** becomes diagonal inside the space of **V** (i.e. **V** moves us into the "diagonal space", and then V^{-1} gets us back out). This can be interpreted in the context of basis vectors: **A** is dense in the standard basis, but then we apply a set of transformations (**V**) to rotate the matrix into a new set of basis vectors (i.e. the eigenvectors) in which the information is sparse and represented by a diagonal matrix.

Implementing Eigendecomposition in Python ... and Reconstruction

The above equations describing the relationships between eigenvalues, eigenvectors, and the original matrix are key.

As an example of implementing eigendecomposition in Python and then reconstructing the original matrix **A** again, see below.

In this example we start with a 4x4 matrix of random numbers: `M=np.random.rand(4,4)`.

Then perform eigendecomposition:

```
Lambda, V = np.linalg.eig(M)

Lambda
#array([ 2.23181445, -0.45111139, -0.03815004,  0.29505027])

V
#array([[-0.68399481, -0.74663466, -0.37373889, -0.37896774],
       [-0.56713117,  0.65054355, -0.4304368 , -0.56779075],
       [-0.40996887,  0.09567061,  0.80682488,  0.13311561],
       [-0.20600692,  0.10088063, -0.15516769,  0.71852443]])
```

To calculate all the components of the eigendecomposition, we need Λ and V^{-1}.

First, for the eigenvalues Λ we just have to transform "Lambda" into a diagonal matrix:

```
bLambda=np.diag(Lambda)

bLambda
#array([[ 2.23181445,  0.        ,  0.        ,  0.        ],
       [ 0.        , -0.45111139,  0.        ,  0.        ],
       [ 0.        ,  0.        , -0.03815004,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.29505027]])
```

Second, we use `np.linalg.inv` to compute V^{-1}:

```
Vinv=np.linalg.inv(V)

Vinv
#array([[-0.55271075, -0.46090615, -0.60642142, -0.54338254],
       [-0.6905315 ,  0.70422395,  0.08961782,  0.17568387],
       [-0.18232051, -0.26996913,  0.91869023, -0.47969362],
       [-0.10088904, -0.28931918,  0.01194578,  1.10769148]])
```

Now, we have all the ingredients we need to reconstruct **A**. And we do that exactly in the same way we reconstructed matrices from SVD:

```
Mrecon = np.dot(U,np.dot(bLambda, Vinv))


M-Mrecon

#array([[ 3.33066907e-16,  6.66133815e-16,  1.11022302e-15,
         1.44328993e-15],
       [ 3.33066907e-16, -8.32667268e-16,  0.00000000e+00,
         4.99600361e-16],
       [ 0.00000000e+00, -2.22044605e-16,  1.11022302e-16,
         3.33066907e-16],
       [ 5.55111512e-17, -2.22044605e-16,  0.00000000e+00,
         1.66533454e-16]])
```

The tiny difference between **M** and **Mrecon** (∼ machine precision level) confirms that we have successfully reconstructed **M**.

Eigendecomposition of Singular Matrices

We have said before that if you have a singular matrix you are in trouble because you cannot invert that matrix. However, you can eigendecompose a singular matrix.

Here's an example:

```
a=np.array([[1,4,7],[2,5,8],[3,6,9]])
L,V=np.linalg.eig(a)

print('Rank = {0} \n'.format(np.linalg.matrix_rank(a)))
print('Eigenvalues: {0}'.format( L.round(2)))
print('Eigenvectors: {0}'.format(V.round(2)))

#Rank = 2
#Eigenvalues: [16.12 -1.12 -0.  ]
#Eigenvectors: [[-0.46 -0.88  0.41]
 [-0.57 -0.24 -0.82]
 [-0.68  0.4   0.41]]
```

This matrix has a rank of 2. It has one zero-valued eigenvalu with a nonzero eigenvector.

For singular matrices, at least one eigenvalue is guaranteed to be zero. This doesn'tm ean that the number of nonzero eigenvalues equals the rank of the matrix -- that's true for singular values (the scalar values from SVD) but not for eigenvalues. But if the matrix is singular, then at least one eigenvalue equals zero.

The converse is true: every full-rank matrix has zero zero-valued eigenvalues.

The point of this section: 1) eigendecomposition is valid for reduced rank matrices and 2) the presence of at least one zero-valued eigenvalue indicates a reduced-rank matrix.

A few other notes ...

Now, before we get to the fun stuff there are a few other things to mention.

- Eigenvectors are stored in the Columns, not Rows. --- Now, sometimes if you forget this, you will try to do some linear algebra operation that triggers a broadcasting error ... but not always. So keep this in mind: columns not rows.
- Sign and Scale Indeterminancy of Eigenvectors -- Go back to our example eigenvector determination of [1 1]. Sure that works. But so does [2 2] or [-10.7 10.7] etc. Any *scaled* version of the vector [1 1] is a basis for that the null space. In other words, if **v** is an eigenvector of a matrix, then so is $\alpha\mathbf{v}$, where α is a non-zero scalar.
- Eigendecomposition works with square matrices only -- Just thought I would repeat this point for emphasis.

Now, so far this seems a bit like SVD. We have decomposed a matrix and get back something (eigenvalues and eigenvectors, in this case). SVD and eigendecomposition are powerful because we can make Python *truncate* these matrices to filter out noise (or in more jargony language, filter out the components that do not explain much of the variance we see). And that leads us to *principle component analysis*.