Lab Manual

for

Linear Algebra

by

Jim Hefferon

Cover: my Chocolate Lab, Suzy.

Contents

Singular Value Decomposition

Recall that a line through the origin in $\mathbb{R}^n$ is $\{r \cdot \vec{v} \,\big|\, r \in \mathbb{R}\}$. One of the defining properties of a linear map is that $h(r \cdot \vec{v}) = r \cdot h(\vec{v})$. So the action of h on any line through the origin is determined by the action of h on any nonzero vector in that line.

For instance consider the line $y = 2x$ in the plane.

$$\{r \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} \,\big|\, r \in \mathbb{R}\}$$

If $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ is represented by the matrix

$$\mathrm{Rep}_{E_2,E_2}(t) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

then here is the effect of t on one vector in the line.

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 7 \\ 10 \end{pmatrix}$$

And here is the effect of t on other members of the line.

$$\begin{pmatrix} 2 \\ 4 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 14 \\ 20 \end{pmatrix} \qquad \begin{pmatrix} -3 \\ -6 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} -21 \\ -30 \end{pmatrix} \qquad \begin{pmatrix} r \\ 2r \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 7r \\ 10r \end{pmatrix}$$

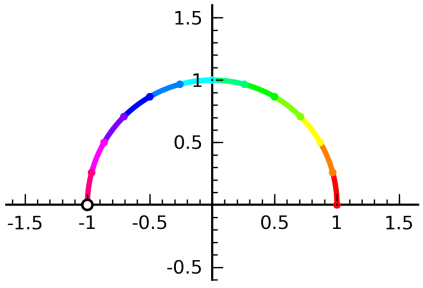
So this condition in the definition of linear map imposes a simple uniformity on its action on lines through the origin.

Unit circle

Consider transformations of the plane $\mathbb{R}^2$. Because of $t(r \cdot \vec{v}) = r \cdot t(\vec{v})$, one way to describe a transformation's action is to pick a set containing one nonzero vector from each line through the origin, describe where the transformation maps those elements.

A natural set that contains one nonzero element from each line through the origin is the upper half unit circle.

$$U = \{\begin{pmatrix} x \\ y \end{pmatrix} \,\big|\, x = \cos(t),\ y = \sin(t),\ 0 \leqslant t < \pi\}$$



The above graph came from using a routine that draws the effect of a transformation on the unit circle and then specifying the identity transformation.

```
sage: load "plot_action.sage"
sage: p = plot_circle_action(1,0,0,1)  # identity matrix
sage: p.set_axes_range(-1.5, 1.5, -0.5, 1.5)
sage: p.save("graphics/svd000.png")
```

Its source code is at the end of this chapter but in short `plot_circle_action(a,b,c,d)` multiplies points on the unit circle by this matrix.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

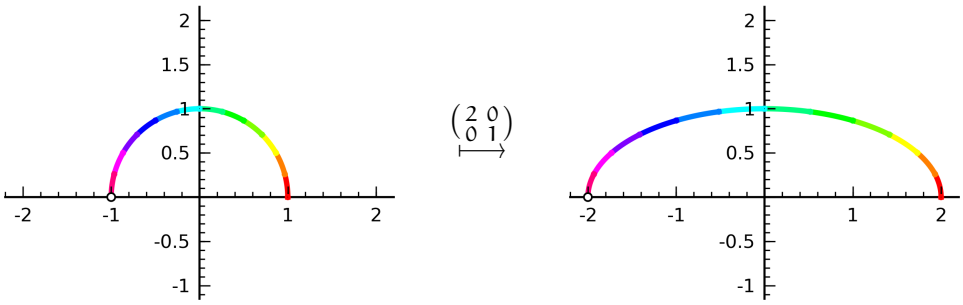
The colors are for the before and after pictures below.

Here is the effect of the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} 2x \\ y \end{pmatrix}$$

that doubles the x component of input vectors. It shows before and after, the upper half circle domain and the output codomain.

```
sage: p = plot_circle_action(2,0,0,1)
```

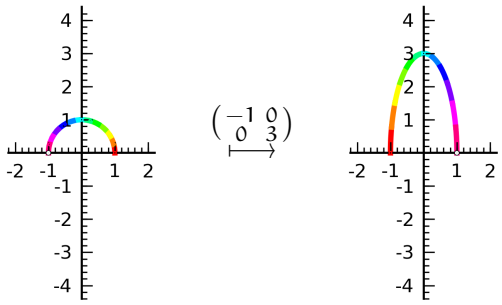


The next before and after picture shows the effect of the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} -x \\ 3y \end{pmatrix}$$

that triples the y component and multiplies the x component by -1.

```
sage: p = plot_circle_action(-1,0,0,3)
```



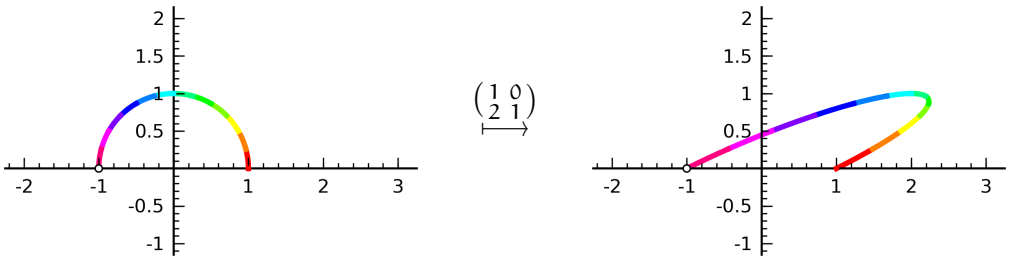
Now the colors come in. The input circle moves counterclockwise from red to orange, then to green, blue, indigo, and finally to violet. But the output does the opposite: to move from red to violet you move clockwise. This transformation changes the *orientation* (or *sense*) of the curve.

The next transformation is a skew.

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x+2y \\ y \end{pmatrix}$$

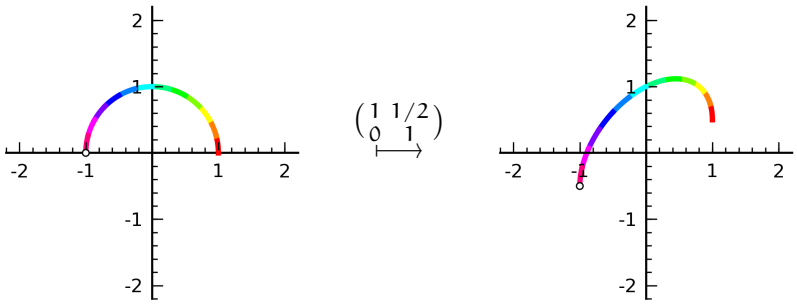
The output's first component is affected by the input vector's distance from the y-axis.

```
sage: p = plot_circle_action(1,0,2,1)
```



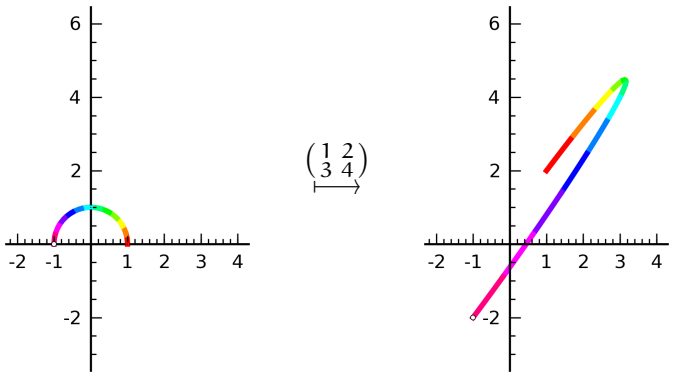
Here is another skew. In this case the output's second component is affected by the input's distance from the x-axis.

```
sage: p = plot_circle_action(1,1/2,0,1)
```



And here is a generic transformation. It changes orientation also.

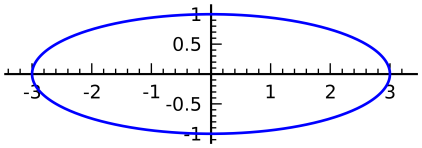
```
sage: p = plot_circle_action(1,2,3,4)
```



SVD

The above pictures show the unit circle mapping to ellipses. Recall that in $\mathbb{R}^2$ an ellipse has a *major axis*, the longer one, and a *minor axis*.[1] Write σ_1 for the length of the semi-major axis, the distance from the center to the furthest-away point on the ellipse, and write σ_2 for the length of the semi-minor axis.

```
sage: sigma_1=3
sage: sigma_2=1
sage: E = ellipse((0,0), sigma_1, sigma_2)
sage: E.save("graphics/svd100.png")
```

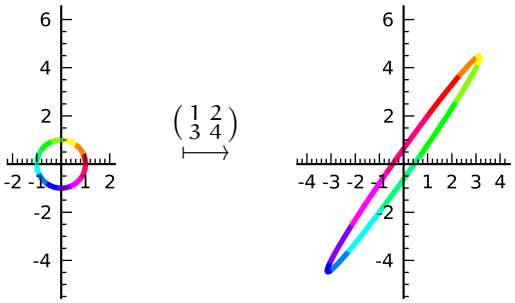


In an ellipse the two axes are orthogonal. In the above graph the major axis lies along the x-axis while the minor axis lies along the y-axis.

Under any linear map $t\colon \mathbb{R}^n \to \mathbb{R}^m$ the unit sphere maps to a hyperellipse. This is a version of the *Singular Value Decomposition* of matrices: for any linear map $t\colon \mathbb{R}^m \to \mathbb{R}^n$ there are bases $B = \langle \vec{\beta}_1, \ldots, \vec{\beta}_m \rangle$ for the domain and $D = \langle \vec{\delta}_1, \ldots, \vec{\delta}_n \rangle$ for the codomain such that $t(\vec{\beta}_i) = \sigma_i \vec{\delta}_i$, where the *singular values* σ_i are scalars. The next section sketches a proof but we first illustrate this result by using an example matrix. Sage will find the two bases B and D and will picture how the vectors $\vec{\beta}_i$ are mapped to the $\sigma_i \vec{\text{delta}}_i$.

So consider again the generic matrix. Here is its action again, this time shown on a full circle.

```
sage: p = plot_circle_action(1,2,3,4,full_circle=True)
```



(*)

Sage will find the SVD of this example matrix.

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.404553584834  -0.914514295677]
[-0.914514295677   0.404553584834]
sage: Sigma
```

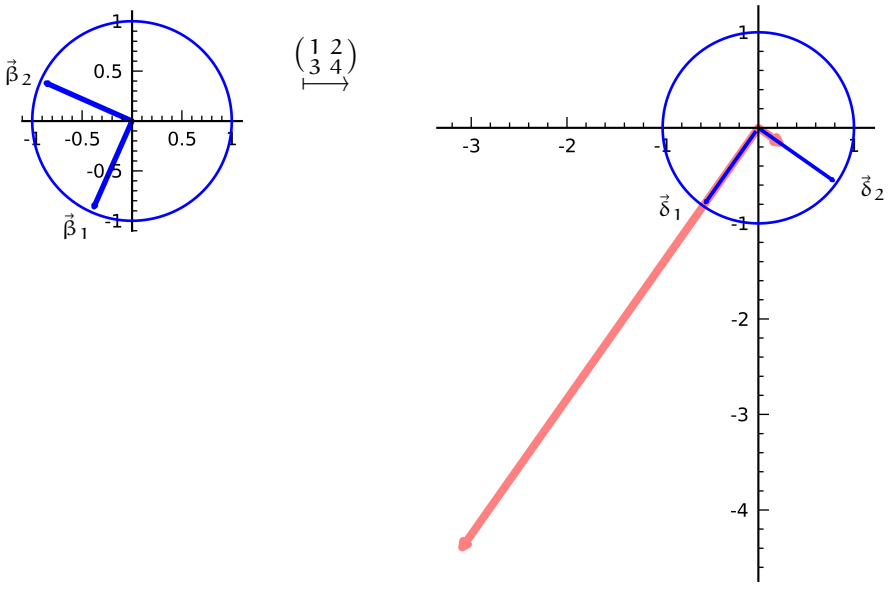
[1] If the two axes have the same length then the ellipse is a circle. If one axis has length zero then the ellipse is a line segment and if both have length zero then it is a point.

```
[ 5.46498570422              0.0]
[            0.0  0.365966190626]
sage: V
[-0.576048436766   0.81741556047]
[ -0.81741556047 -0.576048436766]
sage: U*Sigma*(V.transpose())
[1.0  2.0]
[3.0  4.0]
```

The Singular Value Decomposition has M as the product of three matrices, $U\Sigma V^T$. The basis vectors $\vec{\beta}_1$, $\vec{\beta}_2$, $\vec{\delta}_1$, and $\vec{\delta}_2$ are the columns of U and V. The singular values are the diagonal entries of Σ. Sage will plots the effect of the transformation on the basis vectors for the domain so we can compare those with the basis vectors for the codomain.

```
sage: beta_1 = vector(RDF, [U[0][0], U[1][0]])
sage: beta_2 = vector(RDF, [U[0][1], U[1][1]])
sage: delta_1 = vector(RDF, [V[0][0], V[1][0]])
sage: delta_2 = vector(RDF, [V[0][1], V[1][1]])
sage: C = circle((0,0), 1)
sage: P = C + plot(beta_1) + plot(beta_2)
sage: P.save("graphics/svd102a.png")
sage: image_color=Color(1,0.5,0.5)    # color for t(beta_1), t(beta_2)
sage: Q = C + plot(beta_1*M, width=3, color=image_color)
sage: Q = Q + plot(delta_1,width=1.4, color='blue')
sage: Q = Q + plot(beta_2*M,width=3, color=image_color)
sage: Q = Q + plot(delta_2,width=1.4, color='blue')
sage: Q.save("graphics/svd102b.png")
```

In the picture below the domain's blue $\vec{\beta}$'s on the left map to the codomain's light red $t(\vec{\beta})$'s on the right. Also on the right, in blue, are the $\vec{\delta}$'s. The red $t(\vec{\beta}_1)$ does look to be about 5.5 times $\vec{\delta}_1$, and $t(\vec{\beta}_2)$ does look something like 0.4 times $\vec{\delta}_2$.



(**)

Note also that the two bases are *orthonormal*—the unit circles help us see that the bases are comprised of unit vectors and further, the two members of each basis are orthogonal.

Compare this diagram to the one before it labelled $(*)$, which shows the effect of the matrix on the unit circle. We used the whole circle in $(*)$ to spotlight the ellipse and to make clearer that in $(**)$ the longest red vector is a semi-major axis of that ellipse and the shortest red vector is a semi-minor axis.

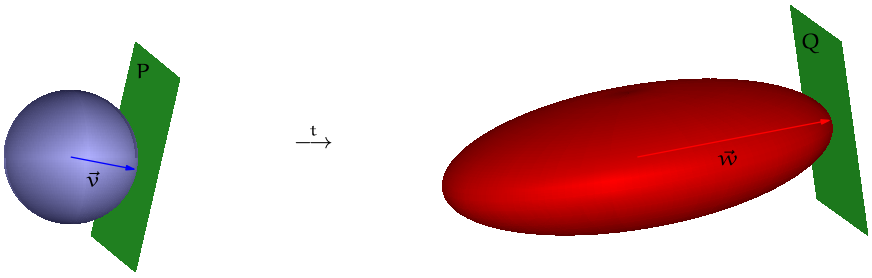
Proof sketch

This argument, adapted from Blank et al. [1989], is a sketch because it uses results that a typical reader has only seen in a less general version and because it relies on material from the book that is optional. In addition, we'll consider only the case of a nonsingular matrix and map; it shows the main idea, which is the point of a sketch.

Consider an $n \times n$ matrix T that is nonsingular, and the nonsingular transformation $t: \mathbb{R}^n \to \mathbb{R}^n$ represented by T with respect to the standard bases.

Recall the Extreme Value Theorem from Calculus I: for a continuous function f, if $D \subset \mathbb{R}$ is a closed and bounded set then the image $f(D)$ is also a closed and bounded set (see Wikipedia [2012a]). A generalization of that result gives that because the unit sphere in $\mathbb{R}^n$ is closed and bounded then its image under t is closed and bounded. Although we won't prove this, the image is in fact an ellipsoid so we will call it that.

Because the ellipsoid is closed and bounded it has a point furthest from the origin. Let $\vec{w}$ be a vector extending from the origin to that furthest point. Let $\vec{v}$ be the member of the unit sphere that maps to $\vec{w}$. Let P be the plane that touches the sphere only at the endpoint of $\vec{v}$. Let Q be the image of P under t. Since t is one-to-one, Q touches the ellipsoid only at $\vec{w}$.



The tangent plane P has the property that the set of vectors whose bodies lie in P are the set of vectors perpendicular to $\vec{v}$. That is, if in the picture above we slide P along the vector $\vec{v}$ to the origin then we have the subspace of $\mathbb{R}^n$ of vectors perpendicular to $\vec{v}$. This subspace has dimension $n-1$. We will argue below that if we similarly slide Q to the origin then we have the set of vectors perpendicular to $\vec{w}$. With that we will have an argument by induction: we start constructing the bases B and D by taking $\vec{\beta}_1$ to be $\vec{v}$, taking σ_1 to be the length $\|\vec{w}\|$, and taking $\vec{\delta}_1$ to be $\vec{w}/\|\vec{w}\|$. Then we proceed by considering the restriction of t to P.

So consider Q. First observe that there is a unique plane that touches the ellipsoid only at $\vec{w}$ since if there were another $\hat{Q}$ then $t^{-1}(\hat{Q})$ would be a second plane, besides P, that touches the sphere only at $\vec{v}$, which is impossible. To see that Q is perpendicular to $\vec{w}$ consider a sphere centered at the origin whose radius is $\|\vec{w}\|$. This sphere has a plane tangent at the endpoint of $\vec{w}$, which is perpendicular to $\vec{w}$. Because $\vec{w}$ ends at a point on the ellipsoid furthest from the origin, the ellipsoid is entirely contained in the sphere, so this plane touches the ellipsoid only at $\vec{w}$. Therefore, from the second sentence of this paragraph, this plane is Q. That ends the argument.

Matrix factorization

We can express those geometric ideas in an algebraic form (for a proof see Trefethen and Bau [1997]).

The *singular value decomposition* of an $m \times n$ matrix A is a factorization $A = U\Sigma V^T$. The $m \times n$ matrix Σ is is all zeroes except for diagonal entries, the singular values, $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r > 0$ where r is the rank of A. The $m \times m$ matrix U and the $n \times n$ matrix V are unitary, meaning that their columns form an orthogonal basis of unit vectors, the left and right *singular vectors* for A, respectively.

```
sage: M = matrix(RDF, [[0, 1, 2], [3, 4, 5]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.274721127897 -0.961523947641]
[-0.961523947641  0.274721127897]
sage: Sigma
[7.34846922835            0.0            0.0]
[           0.0            1.0            0.0]
sage: V
[-0.392540507864  0.824163383692  0.408248290464]
[-0.560772154092  0.137360563949 -0.816496580928]
[ -0.72900380032 -0.549442255795  0.408248290464]
```

The product $U\Sigma V^T$ simplifies. To see how, consider the case where all three matrices are 2×2. Write $\vec{u}_1$, $\vec{u}_2$ for the columns of U and $\vec{v}_1$, $\vec{v}_2$ for the columns of V, so that the rows of V^T are $\vec{v}_1^T$ and $\vec{v}_2^T$.

$$\begin{aligned} U\Sigma V^T &= \begin{pmatrix} \vec{u}_1 & \vec{u}_2 \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} \vec{v}_1^T \\ \vec{v}_2^T \end{pmatrix} \\ &= \begin{pmatrix} \vec{u}_1 & \vec{u}_2 \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \vec{v}_1^T \\ \vec{v}_2^T \end{pmatrix} + \begin{pmatrix} \vec{u}_1 & \vec{u}_2 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} \vec{v}_1^T \\ \vec{v}_2^T \end{pmatrix} \\ &= \sigma_1 \cdot \begin{pmatrix} \vec{u}_1 & \vec{u}_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \vec{v}_1^T \\ \vec{v}_2^T \end{pmatrix} + \sigma_2 \cdot \begin{pmatrix} \vec{u}_1 & \vec{u}_2 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \vec{v}_1^T \\ \vec{v}_2^T \end{pmatrix} \end{aligned} \qquad (***)$$

In the first term, right multiplication by the $1,1$ unit matrix picks out the first column of U, and left multiplication by the $1,1$ unit matrix picks out first row of V so those are the only parts that remain after the product. In short, we get this.

$$\begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} \\ v_{1,2} & v_{2,2} \end{pmatrix} = \begin{pmatrix} u_{1,1}v_{1,1} & u_{1,1}v_{2,1} \\ u_{2,1}v_{1,1} & u_{2,1}v_{2,1} \end{pmatrix} = \vec{u}_1\vec{v}_1^T$$

Thus, equation $(***)$ simplifies to $U\Sigma V^T = \sigma_1 \cdot \vec{u}_1\vec{v}_1^T + \sigma_2 \cdot \vec{u}_2\vec{v}_2^T$. Cases other than 2×2 simplify in the same way.

Application: data compression

We can write any matrix as a sum $M = \sigma_1 \cdot \vec{u}_1\vec{v}_1^T + \sigma_2 \cdot \vec{u}_2\vec{v}_2^T + \cdots$ where the vectors have unit size and the σ_i's decrease in size.

Suppose that the matrix is $n \times n$. To store or transmit it we need to work with n^2 real numbers. For instance, if $n = 500$ then we have $50^2 = 250\,000$ reals. To express the matrix as a sum, each term requires n real numbers for $\vec{u}_i$, another n reals for $\vec{v}_i$, and one more real for σ_i. So keeping all the terms would require $500 * (2 * 500 + 1) = 500\,500$ reals, which is twice the data of the original matrix. But if you keep only some of the terms, say the first 50 of them, then you would save: $50 \cdot (2n+1) = 50\,050$, which is about 20% of the 500^2 size of the full matrix. Thus, if you have data as a matrix then you can hope to compress it with that summation formula by dropping terms with small σ's. The issue is whether you lose too much of the the information by only retaining the information associated with some of the singular values.

To ilustrate that you can succeed in retaining at least some aspects of data we will do image compression. Meet Lenna. This top third of a pinup is a standard test image Wikipedia [2012b].

The code we will use is in the `img_squeeze` routine listed at the end of this chapter. The code breaks the picture into three matrices, for the red data, the green data, and the blue data.

In that code we have a peek at the eight largest singular values in the red matrix: 93233.7882139, 11193.1186308, 8660.12549856, 7054.38662024, 5916.89456484, 5742.14618029, 4125.16604075, and 3879.15303685. We retain the terms for the largest 10% of the singular values. The code reports that the singular value where we make that cutoff is 606.389279688. It also gives the eight smallest: 0.547288023466, 0.470266870056, 0.1988515109, with five more that are so small they are essentially zero.[1] The large singular values are much larger than the small singular values, even setting aside the ones at the end that are zero except for numerical issues.

We want to see how badly the image degrades for various cutoffs. The code below sets it at 10%. This image is 512×512 so that will sum the terms associated with the first 51 singular values.

```
sage: load "img_squeeze.sage"
sage: img_squeeze("Lenna.png", "Lenna_squeezed.png", 0.10)
```

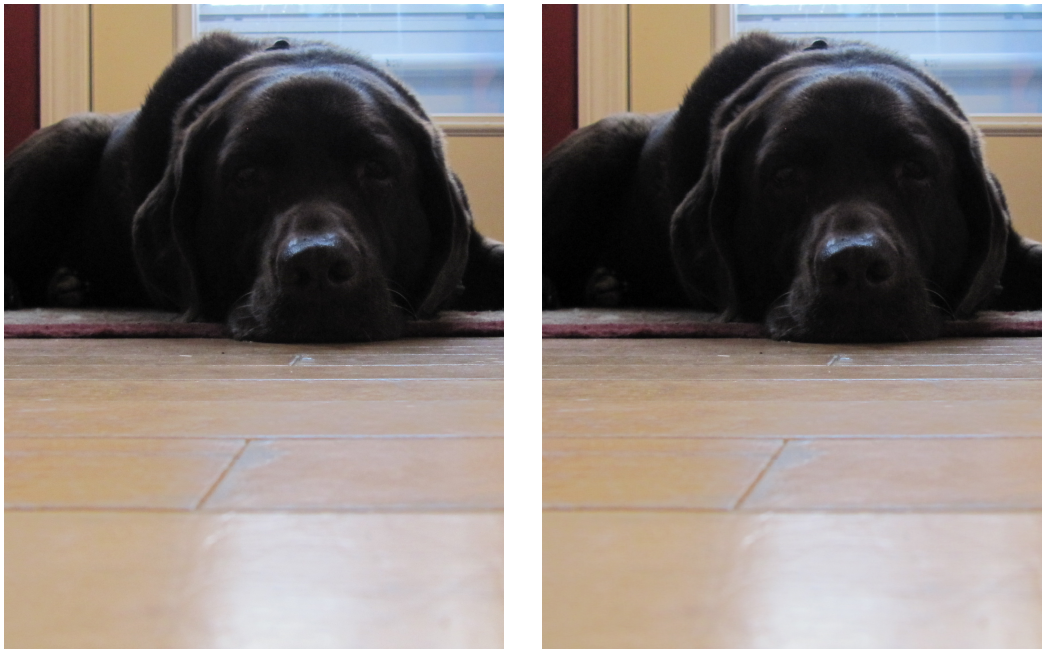
Below is the squeezed image.

[1] These are four values of $1.024\,723\,452\,83 \times 10^{-11}$ and one of $2.325\,339\,864\,91 \times 10^{-12}$.

It definitely shows loss. The colors are not as good, the edges are not sharp, and there are a few artifacts, including a horizontal line across the top of Lenna's forehead and another halfway up her hat. But certainly the image is entirely recognizable.

Where the image is $n \times n$, every additional term in the summation adds to the storage and transmission requirements by about $2n$ reals. The original image requires n^2 reals so we want to choose percentage values less than 0.50. The tradeoff is that higher values give better fidelity. So selecting an optimal percentage parameter is an engineering decision. The above image has the advantage is that it needs only 20% of the storage and transmission requirements of the original image.

Experimenting with percentage value shows that setting the parameter to 20% is enough to make the output image hard to tell from the original. For a final contrast, below is the picture of Suzy from this manual's cover, on the left as the camera took it and on the right squeezed with a cutoff of 0.20.



Source of plot_action.sage

The `plot_circle_action` routine that we call above takes the four entries of the 2×2 matrix and returns a list of graphics. Other parameters are the number of colors, and a flag giving whether to plot a full circle or whether to plot just the top half circle (this is the default).

This routine determines the list of colors and calls a helper `color_circle_list`, given below, which returns a list of graphics. Finally, this routine plots those graphics.

```
def plot_circle_action(a, b, c, d, n = 12, full_circle = False):
    """Show the action of the matrix with entries a, b, c, d on half
    of the unit circle, broken into a number of colors.
     a, b, c, d  reals  Entries are upper left, ur, ll, lr.
     n = 12  positive integer  Number of colors.
     full_circle=False  boolean  Show whole circle, or top half
    """
```

```
    colors = rainbow(n)
    G = Graphics()  # holds graph parts until they are to be shown
    for g_part in color_circle_list(a,b,c,d,colors,full_circle):
        G += g_part
```

The helper does the heavy lifting. It produces a parametrized curve $(x(t), y(t))$, and uses Sage's `parametric_plot` function to get the resulting graphic.

```
DOT_SIZE = .02
CIRCLE_THICKNESS = 2
def color_circle_list(a, b, c, d, colors, full_circle=False):
    """Return list of graph instances for the action of a 2x2 matrix on
    half of the unit circle.  That circle is broken into chunks each
    colored a different color.
      a, b, c, d  reals  entries of the matrix ul, ur, ll, lr
      colors  list of rgb tuples; len of this list is how many chunks
      full_circle=False  Show a full circle instead of a half circle.
    """
    r = []
    if full_circle:
        p = 2*pi
    else:
        p = pi
    t = var('t')
    n = len(colors)
    for i in range(n):
        color = colors[i]
        x(t) = a*cos(t)+c*sin(t)
        y(t) = b*cos(t)+d*sin(t)
        g = parametric_plot((x(t), y(t)),
                            (t, p*i/n, p*(i+1)/n),
                            color = color, thickness=CIRCLE_THICKNESS)
        r.append(g)
        r.append(circle((x(p*i/n), y(p*i/n)), DOT_SIZE, color=color))
    if not(full_circle):    # show (x,y)=(-1,0) is omitted
        r.append(circle((x(pi), y(pi)), 2*DOT_SIZE, color='black',
                        fill = 'true'))
        r.append(circle((x(pi), y(pi)), DOT_SIZE, color='white',
                        fill = 'true'))
    return r
```

If this routine is plotting the upper half circle then it adds a small empty circle at the end to show that the image of $(-1, 0)$ is not part of the graph.

The global variable `CIRCLE_THICKNESS` sets the thickness of the plotted curve, in points (a printer's unit, here 1/72 inch). Similarly `DOT_SIZE` sets the size of the small empty circle.

Source of img_squeeze.sage

We will use the Python Image Library for help in reading the figure, getting the color value at each pixel, etc. The function `img_squeeze` takes three arguments, the names of the input and output

functions, and a real number between 0 and 1 which determines the percentage of the singular values to include in the sum before the cutoff.

```
# Compress the image
import Image

def img_squeeze(fn_in, fn_out, percent):
    """Squeeze an image using Singular Value Decomposition.
        fn_in, fn_out  string  name of file
        percent  real in 0..1  Fraction of singular values to use
    """
```

This function first brings the input data to a format where each pixel is a triple (red, green, blue) of integers that range from 0 to 255. It uses those numbers to gradually build three Python arrays `rd`, `gr`, and `bl`, which then initialize the three Sage matrices `RD`, `GR`, and `BL`.

```
    img = Image.open(fn_in)
    img = img.convert("RGB")
    rows, cols = img.size
    cutoff = int(round(percent*min(rows,cols),0))
    # Gather data into three arrays, then give to Sage's matrix()
    rd, gr, bl = [], [], []
    for row in range(rows):
        for a in [rd, gr, bl]:
            a.append([])
        for col in range(cols):
            r, g, b = img.getpixel((int(row), int(col)))
            rd[row].append(r)
            gr[row].append(g)
            bl[row].append(b)
    RD, GR, BL = matrix(RDF, rd), matrix(RDF, gr), matrix(RDF, bl)
```

The next step finds the Singular Value Decomposition of those three. Out of curiosity, we have a peek at the eight largest singular values in the red matrix, the singular value where we make the cutoff, and the eight smallest.

```
    # Get the SVDs
    print "about to get the svd"
    U_RD,Sigma_RD,V_RD = RD.SVD()
    U_GR,Sigma_GR,V_GR = GR.SVD()
    U_BL,Sigma_BL,V_BL = BL.SVD()
    # Have a look
    for i in range(8):
        print "sigma_RD",i,"=",Sigma_RD[i][i]
    print "  :"
    print "sigma_RD",cutoff,"=",Sigma_RD[cutoff][cutoff]
    print "  :"
    for i in range(Sigma_RD.nrows()-8,Sigma_RD.nrows()):
        print "sigma_RD",i,"=",Sigma_RD[i][i]
```

Finally, for each matrix we compute the sum $\sigma_1 \cdot \vec{u}_1\vec{v}_1^{\mathsf{T}} + \sigma_2 \cdot \vec{u}_2\vec{v}_2^{\mathsf{T}} + \cdots$ up through the cutoff index.

```
    # Compute sigma_1 u_1 v_1^trans+ ..
```

```
    a=[]
    for i in range(rows):
        a.append([])
        for j in range(cols):
            a[i].append(0)
    A_RD, A_GR, A_BL = matrix(RDF, a), matrix(RDF, a), matrix(RDF, a)
    for i in range(cutoff):
        sigma_i = Sigma_RD[i][i]
        u_i = matrix(RDF, U_RD.column(i).transpose())
        v_i = matrix(RDF, V_RD.column(i))
        A_RD = copy(A_RD)+sigma_i*u_i*v_i
        sigma_i = Sigma_GR[i][i]
        u_i = matrix(RDF, U_GR.column(i).transpose())
        v_i = matrix(RDF, V_GR.column(i))
        A_GR = copy(A_GR)+sigma_i*u_i*v_i
        sigma_i = Sigma_BL[i][i]
        u_i = matrix(RDF, U_BL.column(i).transpose())
        v_i = matrix(RDF, V_BL.column(i))
        A_BL = copy(A_BL)+sigma_i*u_i*v_i
```

The code asks for the transpose of the `U_RD.column(i)`, etc., which may seem to be a mistake. Remember that Sage favors rows for vectors, so this is how we make the $\vec{u}$ vector a column, while $\vec{v}$ is already a row. Because of this operation on the vector, the first time you run this code in a Sage session you may get a depreciation warning.

To finish, we put the data in the `.PNG` format and save it to disk.

```
    # Make a new image
    img_squoze = Image.new("RGB", img.size)
    for row in range(rows):
        print "transferring over row=", row
        for col in range(cols):
            p = (int(A_RD[row][col]),
                 int(A_GR[row][col]),
                 int(A_BL[row][col]))
            img_squoze.putpixel((int(row), int(col)), p)
    img_squoze.save(fn_out)
```

This part of the routine function takes a very long time to run, in part because the code is intended to be easy to read rather than fast. Consequently, in the function source there are some status lines that help convince a person executing the code that it is still working; those busy-work lines are left out of some of the earlier output listings.

Bibliography

Robert A. Beezer. Sage for Linear Algebra. http://linear.ups.edu/download/fcla-2.22-sage-4.7.1-preview.pdf, 2011.

S. J. Blank, Nishan Krikorian, and David Spring. A geometrically inspired proof of the singular value decomposition. *The American Mathematics Monthly*, pages 238–239, March 1989.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

Jim Hefferon. Linear Algebra. http://joshua.smcvt.edu/linearalgebra, 2012.

David Joyner and William Stein. Open source mathematical software. *Notices of the AMS*, page 1279, November 2007.

Python Team. Floating point arithmetic: issues and limitations, 2012a. URL http://docs.python.org/2/tutorial/floatingpoint.html. [Online; accessed 17-Dec-2012].

Python Team. The python tutorial, 2012b. URL http://docs.python.org/2/tutorial/index.html. [Online; accessed 17-Dec-2012].

Ron Brandt. *Powerful Learning*. Association for Supervision and Curriculum Development, 1998.

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012a.

Sage Development Team. Sage reference manual 5.3. http://www.sagemath.org/pdf/reference.pdf, 2012b.

Shunryu Suzuki. *Zen Mind, Beginners Mind*. Shambhala, 2006.

Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

Wikipedia. Extreme value theorem, 2012a. URL http://en.wikipedia.org/wiki/Extreme_value_theorem. [Online; accessed 28-Nov-2012].

Wikipedia. Lenna, 2012b. URL http://en.wikipedia.org/wiki/Lenna. [Online; accessed 29-Nov-2012].