AM205: Assignment 2 solutions[*]

Problem 1 – norms and Newton root finding

Part (a)

Write $b = (x, y)$. Then $\|b\|_2 = 1$ implies

$$x^2 + y^2 = 1 \tag{1}$$

and $\|Ab\|_2 = 1$ implies that

$$(4x - y)^2 + x^2 = 17x^2 - 8xy - y^2 = 1. \tag{2}$$

Subtracting Eq. 1 from Eq. 2 gives

$$0 = 16x^2 - 8xy = 8x(2x - y) \tag{3}$$

and hence either $x = 0$ or $x = \frac{y}{2}$. If $x = 0$, then $y = \pm 1$. If $x = \frac{y}{2}$, then substituting into Eq. 1 gives

$$\frac{y^2}{4} + y^2 = 1 \tag{4}$$

and hence $y = \pm\frac{2}{\sqrt{5}}$. The four solutions to $\|b\|_2 = 1$ and $\|Ab\|_2 = 1$ are therefore

$$(0, 1), \qquad (0, -1), \qquad (\tfrac{1}{\sqrt{5}}, \tfrac{2}{\sqrt{5}}), \qquad (-\tfrac{1}{\sqrt{5}}, -\tfrac{2}{\sqrt{5}}).$$

Figure 1(a) shows a plot of the loci of $\|b\|_2 = 1$ and $\|Ab\|_2 = 1$, with these four points marked on the plot.

Part (b)

Write $b = (x, y)$. Then $\|b\|_\infty = 1$ implies either

1. $|x| = 1$ and $|y| \le 1$,
2. $|x| \le 1$ and $|y| = 1$.

Consider case 1. Then

$$1 = \|Ab\|_\infty = \max\{|4x - y|, |x|\} = \max\{|4x - y|, 1\}. \tag{5}$$

For this to have solutions, it is necessary for $|4x - y| \le 1$. However, using the reverse triangle inequality shows that

$$|4x - y| \ge |4x| - |y| = 3 - |y| \ge 2$$

[*]Solutions written by Kevin Chen (TF, Fall 2014), Jordan Hoffmann (TF, Fall 2015), Raphaël Pestourie (TF, Fall 2015), and Chris H. Rycroft.

and hence there are no solutions. Now consider case 2, and suppose that $y = 1$. Then

$$1 = \|Ab\|_\infty = \max\{|4x - 1|, |x|\}, \tag{6}$$

which will be satisfied if $x = 0$ or $x = \frac{1}{2}$. If $y = -1$, then Eq. 6 will be satisfied if $x = 0$ or $x = -\frac{1}{2}$. The four solutions are therefore

$$(0, 1), \qquad (0, -1), \qquad (\tfrac{1}{2}, 1), \qquad (-\tfrac{1}{2}, -1).$$

Figure 1(b) shows a plot of the loci of $\|b\|_\infty = 1$ and $\|Ab\|_\infty = 1$, with these four points marked on the plot.

Part (c)

The program `newton.py` finds the solutions of $\|b\|_4 = 1$ and $\|Ab\|_4 = 1$ using the vector Newton–Raphson iteration. To find the four different solutions, the program uses the four solutions from part (a) as initial starting points. The solutions are

$$(0, 1), \qquad (0, -1), \qquad (0.4924791, 0.9849581), \qquad (-0.4924791, -0.9849581).$$

Figure 1(c) shows a plot of the loci of $\|b\|_4 = 1$ and $\|Ab\|_4 = 1$, with these four points marked on the plot.

Part (d)

The twelve points in Fig. 1 lie on the lines $x = 0$ and $x = \frac{y}{2}$. To justify this, note that for any p-norm for $p < \infty$, the unit circle $\|b\|_p = 1$ is given by an equation

$$f(x) + f(y) = 1 \tag{7}$$

where f is some arbitrary even function. The unit circle $\|Ab\|_p = 1$ is given by

$$f(4x - y) + f(x) = 1 \tag{8}$$

and subtracting Eq. 7 from Eq. 8 gives

$$f(4x - y) = f(y). \tag{9}$$

This equation will be satisfied if $4x - y = y$, which implies $x = \frac{y}{2}$. Since f is even, Eq. 9 will also be satisfied if $4x - y = -y$, which implies $x = 0$. As $p \to \infty$, the unit circle $\|b\|_p = 1$ approaches the unit circle of $\|b\|_\infty$, and thus the solutions for the infinity norm should also be colinear.

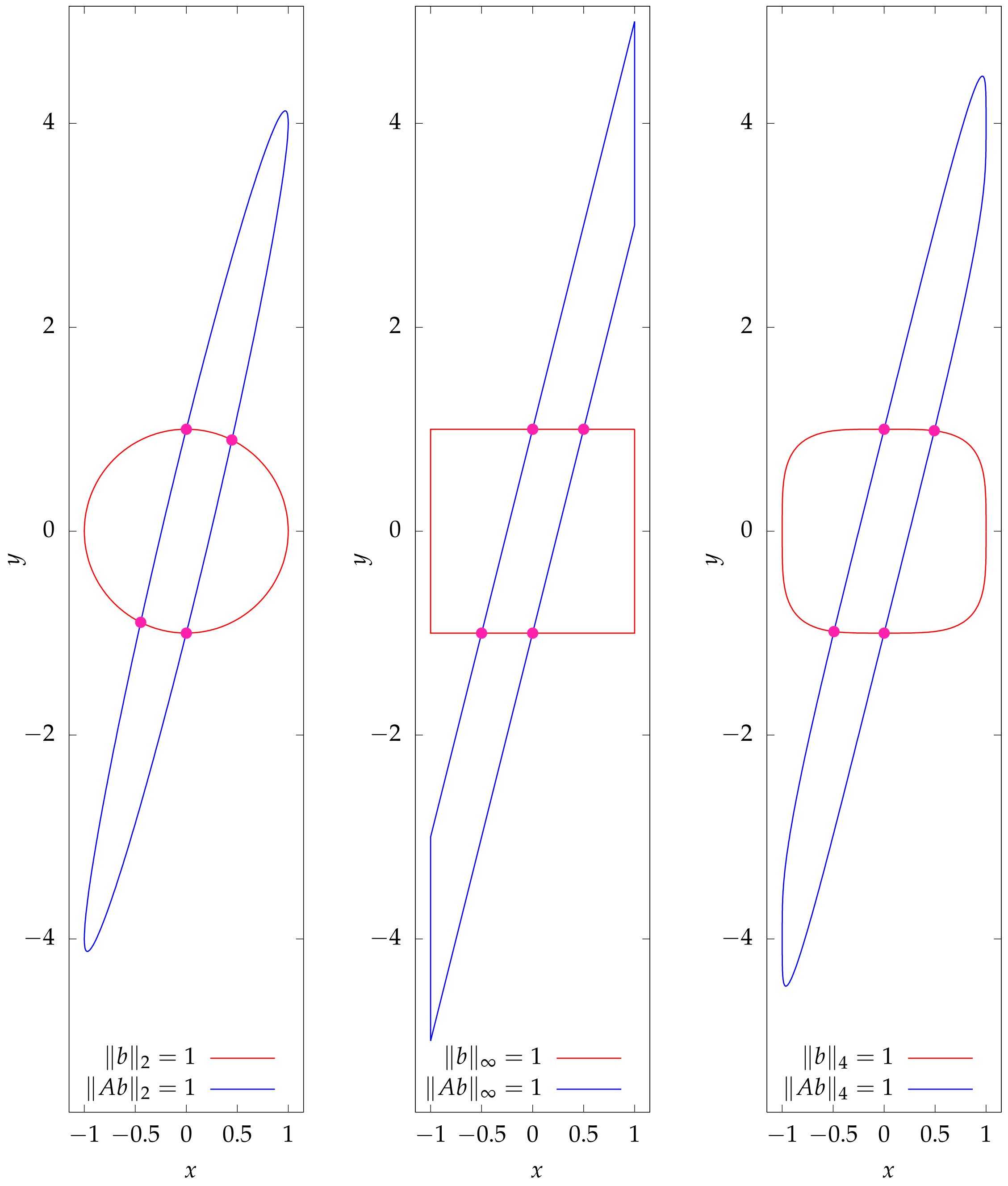

Figure 1: Loci of $\|x\|_p = 1$ and $\|Ax\|_p = 1$ for (a) $p = 2$, (b) $p = \infty$, and (c) $p = 4$. Intersection points on each panel are shown as magenta circles.

Problem 2 – binary LU factorization

Parts (a), (b), and (c)

This problem is very similiar to LU factorization on $\mathbb{R}$ except addition, subtraction, multiplication and division operations are altered. MATLAB and Python implementations of the forward substitution, backward substitution, and the LU factorization are provided, and follow the algorithms laid out on slides 7, 6, and 39 of the lecture 7 notes, respectively.

Part (d) – test cases

The test cases can be solved using the following steps:

1. Calculate matrices L, U, and P as the LU decomposition of A,
2. Solve the linear system $Ly = Pb$ using the forward substitution,
3. Solve the linear system $Ux = y$ using backward substitution.

The solution to the small test case is

$$x = [1,0,1,0,1,0,1,0,1,0,1,0,1,0]^\mathsf{T}. \tag{10}$$

The solution to the large test case is

$$\begin{gathered} x = [0,1,1,0,1,0,1,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0, \\ 1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,1,0,1,0, \\ 0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,1]^\mathsf{T}. \end{gathered} \tag{11}$$

This sequence has the interesting property that the component x_k is equal to one if k is prime and 0 otherwise.

Part (e) – probability of an $n \times n$ binary matrix being singular

We can think of this problem as forming n linearly independent column vectors. For a binary matrix of size n, the total number T of different matrices is given by

$$T = (2^n)^n = 2^{n^2}. \tag{12}$$

To construct a non-singular binary matrix, we consider building it up column by column, so that the columns are linearly independent. The first column can have any vector except the one with all zero entries, and hence the total number of available vectors is

$$C_1 = 2^n - 1. \tag{13}$$

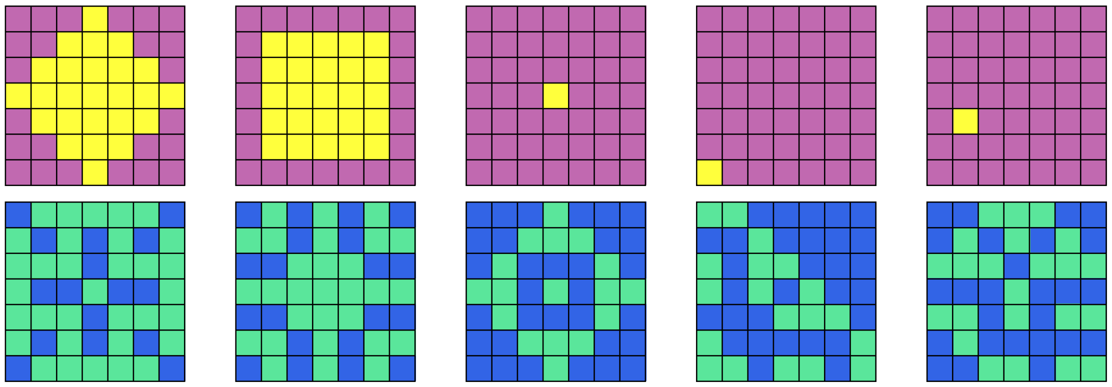
Figure 2: The five light patterns (top row) and the corresponding presses (bottom row) considered in question 3. Lit lights are shown in yellow. Pressed lights are shown in green.

The second column cannot be a linear combination of the first column, and hence the total number of available vectors for the second column is

$$C_2 = 2^n - 2. \tag{14}$$

This pattern continues and for the i^{th} column the total number of available vectors is

$$C_i = 2^n - 2^{i-1}. \tag{15}$$

Hence, the probability of a binary matrix being singular is

$$P = \frac{T - \prod_{i=1}^{n} C_i}{T} = 1 - \frac{(2^n-1)(2^n-2)...(2^n-2^{n-1})}{(2^n)^n}. \tag{16}$$

Problem 3 – the light game

The program `lights.py` takes a light pattern and calculates the presses required to make it using the binary LU solver introduced in problem 2. The presses required to make the five given patterns are shown in Fig. 2. Many interesting custom patterns were submitted, including smileys, a triangle, letters, and the π symbol. In addition, several students found eigenvectors of the matrix A, with eigenvalue 1, corresponding to cases where the button presses and light pattern match.

The program `lights2.py` calculates the dimension of the null space of the matrix A when constructed on on $m \times n$. Due to the symmetry, the program only considers cases where $n \leq m$. Table 1 shows the dimensions for all cases of $m, n \leq 10$, revealing a complicated pattern.

This problem is based on Lights Out, an electronic toy released in 1995, originally using a 5×5 grid. Detailed analyses of this game are available on the web and use more sophisticated mathematical techniques that allow the dimension of the null space to be calculated even for very large m and n.

m, n	1	2	3	4	5	6	7	8	9	10
1	·	1	·	·	1	·	·	1	·	·
2	1	·	2	·	1	·	2	·	1	·
3	·	2	·	·	3	·	·	2	·	·
4	·	·	·	4	·	·	·	·	4	·
5	1	1	3	·	2	·	4	1	1	·
6	·	·	·	·	·	·	·	6	·	·
7	·	2	·	·	4	·	·	2	·	·
8	1	·	2	·	1	6	2	·	1	·
9	·	1	·	4	1	·	·	1	8	·
10	·	·	·	·	·	·	·	·	·	·

Table 1: Dimensions of the null space of the matrix A representing the light game on an $m \times n$ grid. A dot represents a null space of zero dimension. Due to symmetry, the null space of an $m \times n$ grid is the same as for an $n \times m$ grid.

Problem 4 – difficult cases for LU factorization

A MATLAB code to generate G_n is provided. The MATLAB functions `tic` and `toc` were used to record the CPU time required to generate G_n. Figure 3 illustrates time used versus n on a log–log plot. Fitting this data to the function $t(n) = \alpha n^\beta$ gives

$$\alpha = 10^{-7.9}, \qquad \beta = 1.98.$$

We can interpret α as a constant overhead factor and β as the order of complexity. In this case $\beta \approx 2$, which implies the computation time grows proportional to n^2. This is expected, since the size of the matrix is n^2 and the number of negative entries is $\frac{n(n-1)}{2}$, both of which grow quadratically.

Part (c) – LU error

Performing the LU decomposition on G_n yields $G_n = L_n \times U_n$, where

$$L_n = \begin{pmatrix} 1 & 0 & .. & .. \\ -1 & 1 & .. & .. \\ .. & .. & .. & .. \\ -1 & -1 & -1 & 1 \end{pmatrix}, \qquad U_n = \begin{pmatrix} 1 & 0 & .. & 1 \\ 0 & 1 & .. & 2 \\ .. & .. & .. & .. \\ 0 & 0 & 0 & 2^{n-1} \end{pmatrix} \tag{17}$$

The permutation matrix P is simply an identity matrix. since $x = [1, 1, 1, 1, \ldots, 1]^\mathsf{T}$, we have $b = G_n \times x = [2, 1, 0, -1, -2, \ldots, -n+5, -n+4, -n+3, -n+2]^\mathsf{T}$. The forward substitution gives $Ly = b$ gives $y = [2, 3, 5, \ldots, 2^{n-2}+1, 2^{n-1}]$. Finally, the backward

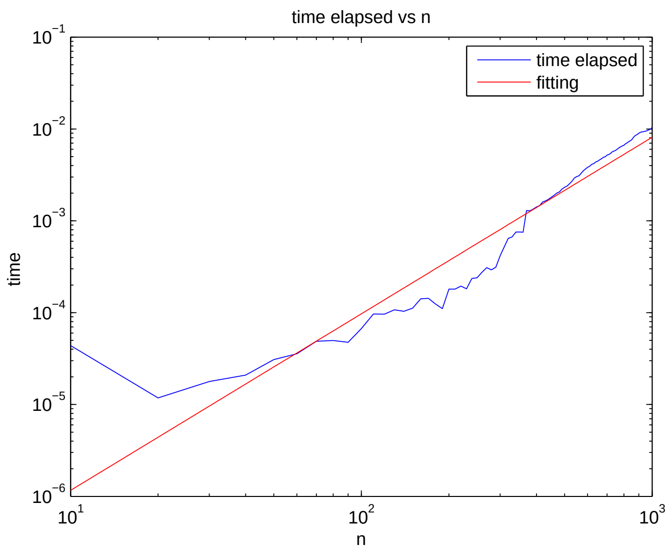

Figure 3: CPU time spent to generate G_n as a function of n.

substitution gives $\hat{x} = [1, 1, 1, 1, \ldots, 1]^\mathsf{T}$ if there is no rounding error. However, since the last column of U and y both scale with 2^{n-1}, rounding error becomes significant for large n. In backward substitution step, we have $\hat{x}_i = 2^{i-1} + 1 - 2^{i-1} = 1$ for all $i \neq n$. For $i = n$, there is no rounding error and $\hat{x}_n = 1$. However, due to rounding, if $i > 52$, we have $x_i = 2^{i-1} + 1 - 2^{i-1} = 0$. Hence, computationally, we have

$$\hat{x} = [1, 1, 1, 1, \ldots, 0, 0, 0, \ldots, 0, 1]^\mathsf{T}. \tag{18}$$

The first 52 entries of $\hat{x}$ are one, and the last entry is one, and all entries in the middle are zero. Hence to compute relative error, we have

$$\frac{\|x - \hat{x}\|}{\|\hat{x}\|_2} = 0 \tag{19}$$

for $n \leq 53$, and

$$\frac{\|x - \hat{x}\|}{\|\hat{x}\|_2} = \frac{\sqrt{n-53}}{\sqrt{53}} \tag{20}$$

for $n > 53$. Figure 4(a) shows the relative error and also shows that the inequality

$$\frac{\|x - \hat{x}\|_2}{\|\hat{x}\|_2} \leq \kappa(G_n, 2)) \frac{\|r(\hat{x})\|_2}{\|A\|_2 \|\hat{x}\|_2} \tag{21}$$

is satisfied, where $\kappa(G_n, 2)$ is the condition number G_n with respect to the 2-norm. Note that problem considered in this question is well-conditioned, since $\kappa(G_n, 2)$ grows slowly.

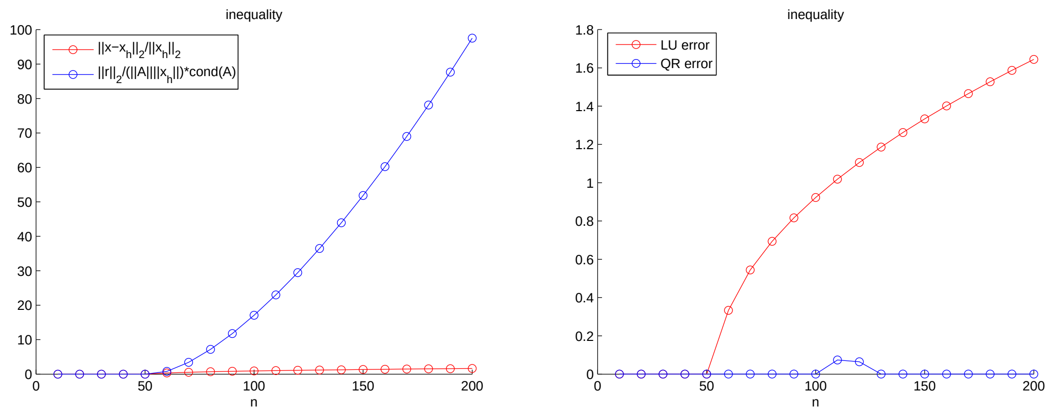

Figure 4: (a) Relative error using the LU decomposition. (b) Relative error comparison between the LU and QR factorizations.

Part (d) – comparison with QR

We can also solve this problem using the QR factorization. Since Q is an orthogonal matrix, the rounding error issue observed in (c) is mitigated. As a result, the relative error is much smaller. Figure 4b) shows the comparison.

Problem 5 – QR factorization using Givens rotations, applied to a bouncing ball

Part (a)

A Python program that performs Givens rotations can be found in `givens.py`. We are asked to test the program by performing 10 trials with random matrices. We are asked to calculate the Frobenius norm, $\|A - QR\|_F$. In doing so, we find:

```
The Frobenius norm is 1.71989412511e-15
The Frobenius norm is 2.4801876374e-15
The Frobenius norm is 1.86327735126e-15
The Frobenius norm is 2.11357387049e-15
The Frobenius norm is 3.02748958682e-15
The Frobenius norm is 1.54630262435e-15
The Frobenius norm is 2.03689632033e-15
The Frobenius norm is 1.5253773799e-15
The Frobenius norm is 2.53880321844e-15
The Frobenius norm is 1.86737483043e-15
```

It is reassuring to see errors of this magnitude. Because in principle, we should get an exact solution we only expect errors due to the inaccuracies of arithmetic. As a further

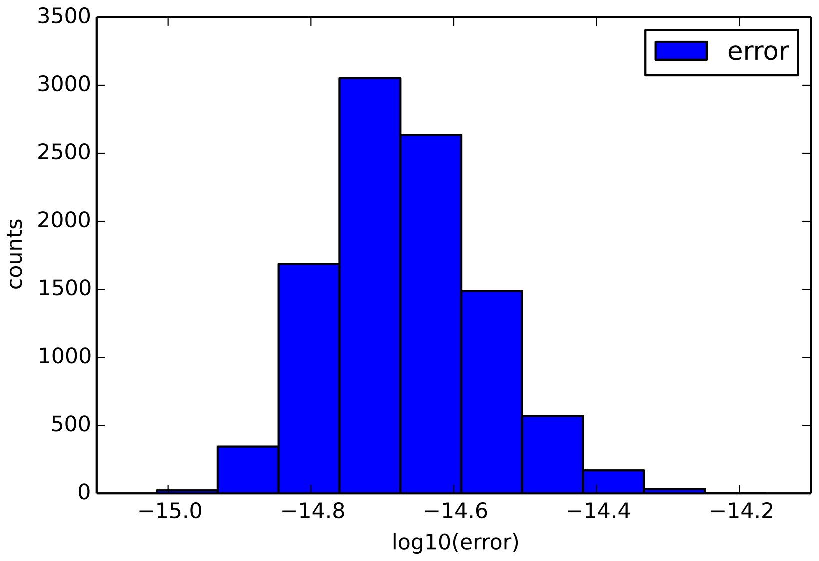

Figure 5: A histogram of the Frobenius Norm for Givens rotations where the decomposed matrix is made up of elements using Python's `np.random.rand()` function. The error is reported using $\log_{10}(\text{error})$.

sanity check, we can perform many simulations and consider $\log_{10}(\text{error})$. The results are shown in Fig. 5.

Part (b)(i)

We are asked to fit each of the three arcs to an equation of the form

$$y(f) = \alpha f^2 + \beta f + \gamma.$$

The code used to do this can be found in the associated files with the name `P5.py`. Note that due to the format of the data, the arcs face upward rather than downward. Rather than invert the data, we choose to leave the data as is and will remember our final value for g will be off by a factor of -1. This is because of the choice of coordinate system. The plot is shown in Figure 6. When calculating the different constants, depending on the data you use, you may get different values. In this case, we convert both the x and y axis to more useful units. We are told that each frame is $7/120$ seconds apart. To start at $t = 0$, we can write

$$\text{time [sec]} = (\text{frame} - 1)\frac{7}{120}.$$

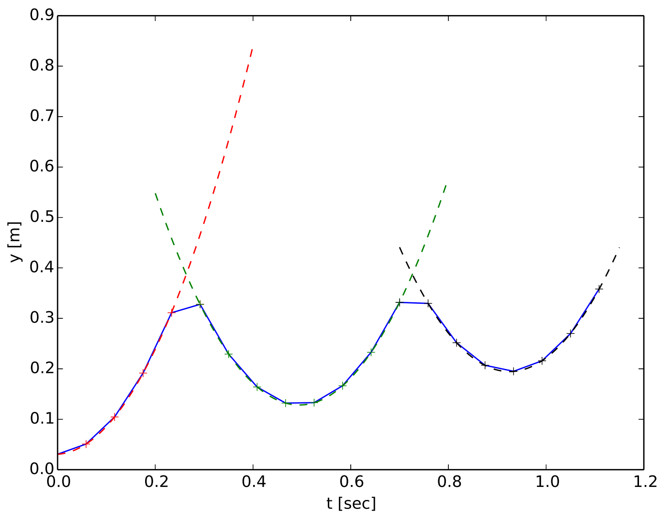

Figure 6: This plot shows the three different arcs and their associated parabolic fits.

To get the distance conversion, we are told that 43.5 pixels corresponds to 42.5 mm. We divide to find that each pixel correspond to 0.00097 meters.

	Arc 1	Arc 2	Arc 3
α	4.90935	4.83274	4.86478
β	0.0599324	-4.78178	-9.00079
γ	0.0306795	1.31106	4.35779

As expected, β and γ are not very constant over the different parabolas. However, α is quite similar for all three. This makes sense (and gives us useful information!) as we will see later.

Part (b)(ii)

Now, using our fits we are asked to calculate a few different quantities.

(A)

We want to calculate the gravitational acceleration. Recall from an introductory physics class that

$$a = \frac{d^2y}{dt^2},$$

where y gives us the position. In this case, we find that

$$\frac{d^2y}{dt^2} = 2\alpha.$$

Thus, we find that

$$2\alpha = \{9.8186, 9.6654, 9.7295\}.$$

Thus, we find (introducing the negative sign to make the value seem more familiar)

$$g = -9.7379 \pm 0.07694.$$

(B)

Now we want the height from which the ball was dropped. Notice that we don't have any data about exactly where the ball hits the table. Thus, we need to figure out where the table is. We can do this by finding where parabola 1 and 2 are equal as well as parabola 2 and 3. Therefore, we need to solve the equation

$$\begin{aligned}4.909349024546918t^2 + 0.05993241246429129t + 0.030679536970813857 = \\ = 4.8327406816693852t^2 - 4.7817810565214023t + 1.3110642433077426 \quad (22)\end{aligned}$$

for t. This can easier be done by hand, numerically, or by using the software package of your choice. In this case, we find that $t = 0.263351317$. Next, we want to evaluate the parabolas at this t to get the value of y corresponding to their intersection (and hopefully the table). We find that

$$y_1 = 0.386945397.$$

Doing an identical procedure for the second and third parabolas, we find that

$$y_2 = 0.387043488.$$

Now, we can take the average of the two values to get the approximate height of the table. We obtain

$$y_{\text{table}} \approx 0.386994.$$

Now, we need to get the position that the ball was originally released from. To do this, we will want to focus on the first arc. Recall that for the first arc, we have calculated

$$y(t) = 4.90935t^2 + 0.0599324t + 0.03067.$$

To find when the ball was released, we want to find when the first derivative (velocity) equals 0. Doing that calculation, we find that that the ball was released at

$$t_0 = -0.00610390626.$$

This makes sense: the ball starts almost exactly from rest. Plugging this time into the parabola, we find a modified release height of

$$y_{\text{start}} = 0.030496626.$$

By subtracting the two heights, we get that the ball was dropped

$$\text{height} \approx 0.356448 \text{ [m]}.$$

However, this is presumably from the middle of the ball. Thus, we need to subtract half the height of the ball to get the height from the bottom of the ball to the table. We get a final answer of

$$\text{height} \approx 0.3352 \text{ [m]}.$$

(C)

The coefficient of restitution e is given by

$$e = \frac{\text{Relative speed after collision}}{\text{Relative speed before collision}}.$$

To get this ratios, we can consider the derivative near the points of contact with the table. As does to compute where the table is located, we find that a collision will occur at $t = 0.263$ seconds. Then we can take the derivative:

$$y'(t) = v(t) = 2\alpha t + \beta.$$

We can compute this value for both sides using the corresponding α and β values. We find

$$e = \frac{2.23636 \text{ m/s}}{2.64570 \text{ m/s}} \approx 0.845282631.$$

Part (b)(iii)

Recall from physics that

$$x = v_0 t + \frac{1}{2} a t^2$$

where x in this case is the height of the drop, h and a is the acceleration due to gravity. Because the ball was dropped, we know that when it was released $v_0 = 0$. Therefore, we can rewrite this equation as

$$h = \frac{1}{2} g t^2.$$

Rearranging this equation, we get that the time for the first drop is given by

$$t = \sqrt{\frac{2h}{g}}.$$

From the conservation of energy we also know that $mgh = 1/2mv^2$. This gets us that

$$v = \sqrt{2gh}.$$

From the coefficient of restitution, we know that the velocity after the nth bounce is given by

$$v_n = e^n v_0 = e v_{n-1}.$$

The time between bounces will be given by

$$t_n = \frac{2v_n}{g}.$$

This can be rewritten using our expression for v_0 and v_n as

$$t_n = \frac{2e^n}{g}\sqrt{2gh}.$$

We wa want to sum up all of these terms. We have that

$$\text{total time} = \text{first fall} + \text{time for all bounces}.$$

This can be written as

$$\text{total time} = \sqrt{\frac{2h}{g}} + \sum_{n=1}^{\infty} \frac{2e^n}{g}\sqrt{2gh}.$$

Recall that for $|r| < 1$ we can write

$$\sum_{n=1}^{\infty} r^n = \frac{r}{1-r}.$$

Using this trick (and noting that $e < 1$) we can write our sum as

$$\text{total time} = \sqrt{\frac{2h}{g}} + \frac{2}{g}\sqrt{2gh}\sum_{n=1}^{\infty} e^n.$$

This gets us a final answer of

$$\text{total time} = \sqrt{\frac{2h}{g}} + \left(\frac{2}{g}\sqrt{2gh}\right)\left(\frac{e}{1-e}\right).$$

Plugging in our values we find

$$\text{total time} \approx 0.262 \text{ sec} + 2.975 \text{ sec} = 3.24 \text{ sec}.$$

Note that this value is quite sensitive to different e and h values (see 7). Using reasonable e and h values for the parameter regime we are interested in rest times tend to be between 3.1 and 3.5 seconds.

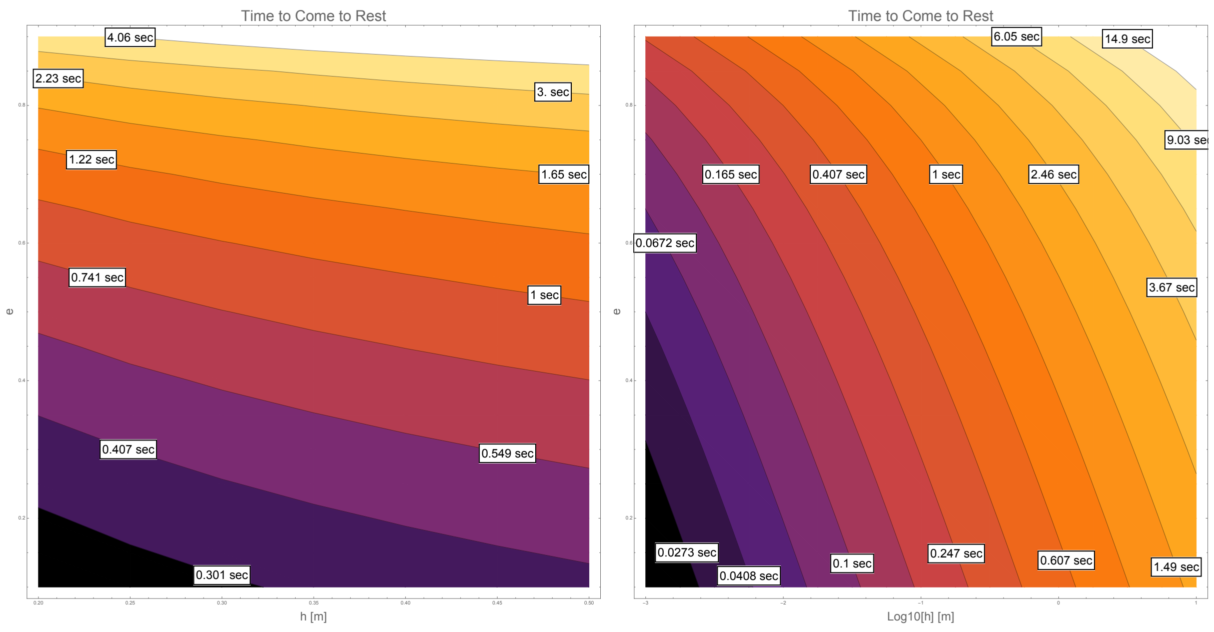

Figure 7: Contour/color plots showing the time for the bouncing ball with coefficient of restitution e to come to rest when dropped from a height h. The left plot shows a range of heights similar to those considered in problem 5. The right plot shows a larger range of heights, using a logarithmic axis. Logarithmic contour spacings are used to help them appear evenly spaced.

Problem 6 – New Hampshire leaf identification with the SVD

Throughout this question the differences in the results between the three resolutions of images, $(m, n) = (712, 560)$, $(m, n) = (538, 420)$, and $(m, n) = (356, 280)$ are negligible.

Part (a)

The program `gen_svd.py` reads in all of the 143 leaf images, and saves an image of the average leaf $\bar{\mathbf{s}}$ as shown in Fig. 8. The program also saves the average leaf vector components to a file, as well as the first sixteen left singular vectors $\mathbf{u}_j$ as described in the question. These files are then read in by subsequent programs for solving the later parts of the question.

Part (b)

The program `plot_svd_vecs.py` plots the positive and negative components to the first three left singular vectors, $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$. They are shown in Fig. 9. The form of $\mathbf{u}_1$ is reasonable, since the data set contains a large number of elliptical shaped leaves, plus a large number of maple leaves. For some constant $\lambda > 0$, the image of $\bar{\mathbf{S}} - \lambda \mathbf{u}_1$ will look like a maple leaf, while the image of $\bar{\mathbf{S}} + \lambda \mathbf{u}_1$ will look like a leaf with an elliptical shape. Hence this vector will likely explain a large amount of the variance in the data set.

Figure 8: The average leaf at the $(m, n) = (712, 560)$ resolution.

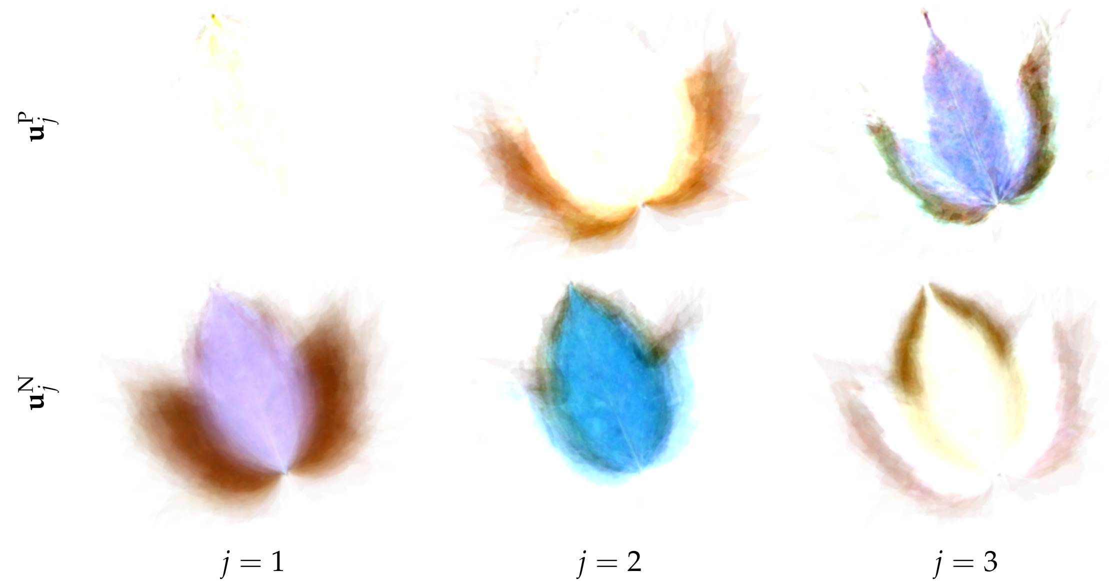

Figure 9: The positive ($\mathbf{u}_j^{\mathrm{P}}$) and negative ($\mathbf{u}_j^{\mathrm{N}}$) components of the three left singular vectors $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$. Note that the positive component of $\mathbf{u}_1$ is localized to a small point near the leaf tip.

Part (c)

The program `leaf_proj_image.py` can plot the projection $\mathbb{P}(\mathbf{T},k)$ of any leaf $\mathbf{T}$ onto the subspace centered on $\bar{\mathbf{S}}$ and spanned by the first k left singular vectors. Figure 10 shows the projections of leaves 14, 33, 87, and 140 using $k = 1,2,4,8,16$. As expected, the clarity of each leaf improves as k increases. By $k = 16$, all the leaf shapes and colors are matched reasonably well, although fine details in the leaf edges are not captured.

Of these four leaves, leaf 87 has the poorest match. Since the green–red color mix is rather unique to this leaf, the set of the first sixteen SVD vectors have a difficult time recovering this feature. The hole in leaf 140 is also not well-captured, although it becomes somewhat visible by $k = 16$.

Part (d)

The program `proj_sizes.py` computes the measure

$$\operatorname{dis}(\mathbf{S}_j) = \frac{1}{mn}\|\mathbf{S}_j - \mathbb{P}(\mathbf{S}_j,8)\|_2^2, \tag{23}$$

for each leaf j. The program determines that

$$\arg\min_j \operatorname{dis}(\mathbf{S}_j) = 57, \qquad \arg\max_j \operatorname{dis}(\mathbf{S}_j) = 120. \tag{24}$$

These results are true for all three resolutions. Specifically,

$$\operatorname{dis}(\mathbf{S}_{57}) = \begin{cases} 0.00967 & \text{for } (m,n) = (712,560), \\ 0.00956 & \text{for } (m,n) = (534,420), \\ 0.00937 & \text{for } (m,n) = (356,230) \end{cases} \tag{25}$$

and

$$\operatorname{dis}(\mathbf{S}_{120}) = \begin{cases} 0.07692 & \text{for } (m,n) = (712,560), \\ 0.07648 & \text{for } (m,n) = (534,420), \\ 0.07561 & \text{for } (m,n) = (356,230). \end{cases} \tag{26}$$

Figure 11 shows the original images $\mathbf{S}_j$ and their projections $\mathbb{P}(\mathbf{S}_j,8)$ for these two cases. Leaf 57 is very well matched by its projection. Leaf 120 is dark and has an unusual shape, and thus $\operatorname{dis}(\mathbf{S}_{120})$ is large, due to many pixels near its boundary not being captured well.

Part (e)

The program `proj_sizes.py` also computes the projection distance measure for the extra eight leaves, which are numbered from 143 to 150 in the provided data files. The values are shown in Table 2 for the three different resolutions. The three lowest projections correspond to leaves 143, 147, and 148.

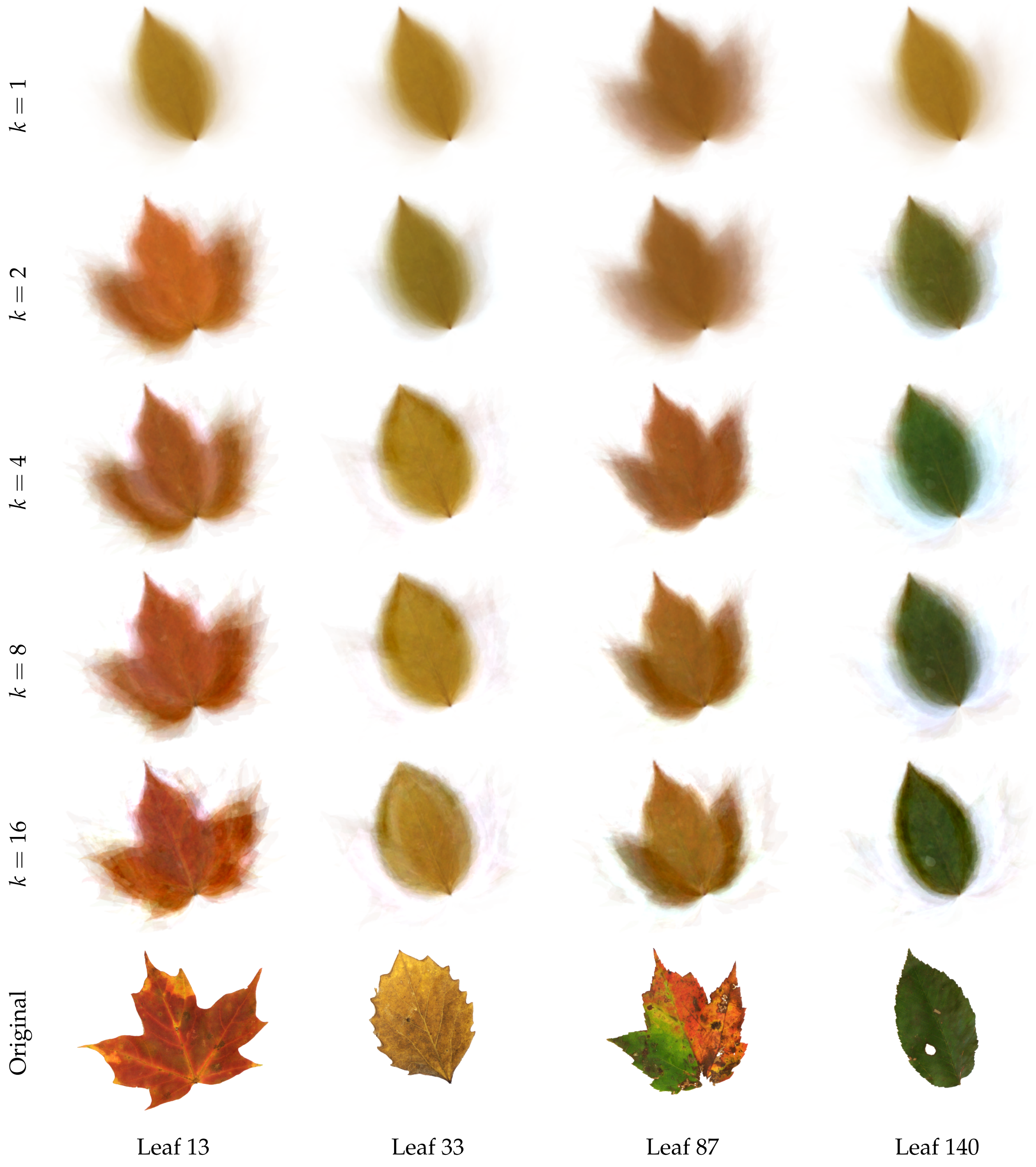

Figure 10: Example projections $\mathbb{P}(\mathbf{S}_j, k)$ for four different leaves and different values of k. The original leaf images are shown in the bottom row for comparison.

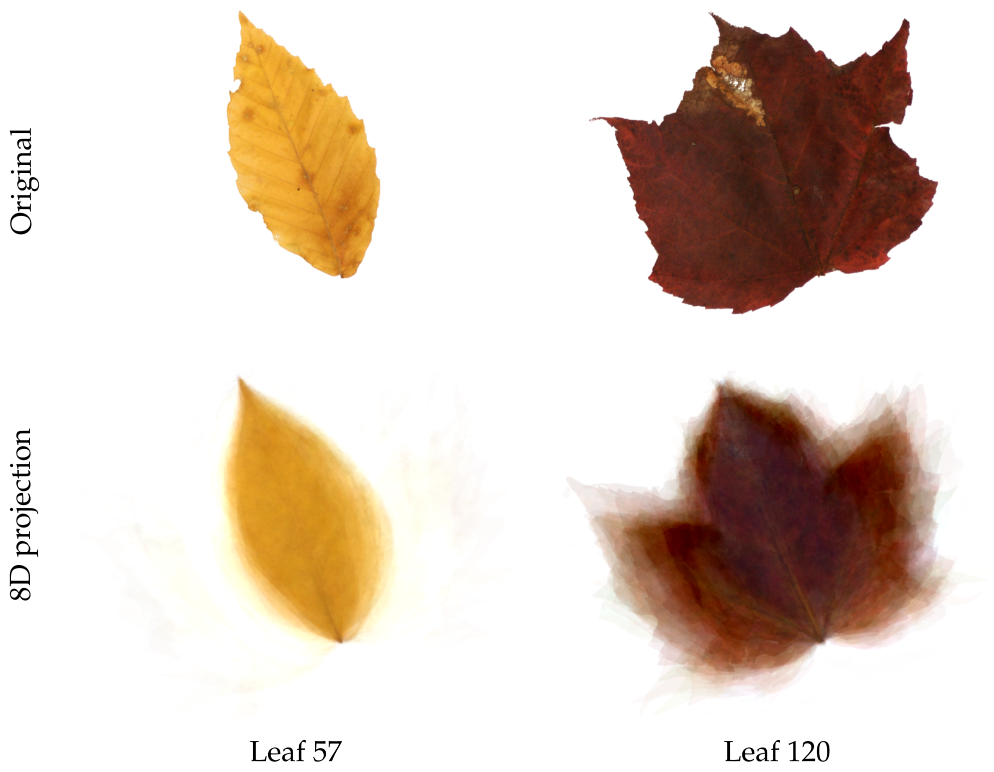

Figure 11: The original image $\mathbf{S}_j$ and its eight-dimensional projection for the case of the best represented leaf ($j = 57$) and the poorest represented leaf ($j = 120$), as measured by $\text{dis}(\mathbf{S}_j) = \frac{1}{mn}\|\mathbf{S}_j - \mathbb{P}(\mathbf{S}_j, 8)\|_2^2$.

In reality, leaves 144, 145, and 147 were collected in New Hampshire while the rest are from Harvard Yard, and thus the method only correctly identifies a single leaf. Note however that leaf 144 is ranked fourth and is close behind the third-placed leaf 148. It also looks reasonable that leaf 143 would be identified as a New Hampshire leaf, since it is very similar to several in the main data set.

It is surprising that leaf 145 is not identified, since it looks like a typical maple leaf and there are many similar ones in the main data set. There are two possible reasons for this. First, the boundary of this leaf is somewhat different to other maple leaves, and thus the projection can only do so well with matching the boundary, incurring a substantial penalty in $\text{dis}(\mathbf{S}_j)$ for boundary pixels that are not recovered well.

Second, it appears that the large values in Table 2 are predominantly weighted toward leaves that take up a large amount of the image. This suggests that normalizing $\text{dis}(\mathbf{S}_j)$ by $\frac{1}{mn}\|\mathbf{S}_j\|_2^2$ may be appropriate. If this is normalization is performed, then leaves 147, 145, and 143 have the three smallest values, and so two out of three of the New Hampshire leaves are identified.

(m,n)	143	144	145	146	147	148	149	150
$(712,560)$	0.01955	0.02799	0.03699	0.04009	0.02519	0.02724	0.04259	0.05453
$(534,420)$	0.01938	0.02767	0.03659	0.03975	0.02493	0.02695	0.04224	0.05416
$(356,280)$	0.01902	0.02702	0.03577	0.03915	0.02442	0.02636	0.04156	0.05339
Rank	1	4	5	6	2	3	7	8

Table 2: Projection distances $\mathrm{dis}(\mathbf{S}_j)$ for the eight extra leaves (numbered from 143 to 150), for the three different image resolutions. The ranking of the distances is also shown from lowest (1) to highest (8). The ranking is consistent for all three image resolutions.