Lab 1

Invertible Affine Transformations and Linear Systems

Lab Objective: *Apply affine transformations to a set of vectors in $\mathbb{R}^2$ and solve linear systems.*

Linear transformations in $\mathbb{R}^2$

Dilations

A *dilation* of the vector space rescales the vectors. Graphically, a dilation stretches or compresses the space. A linear transformation is a dilation if and only if its matrix representation is diagonal, so in particular all Type II elementary matrices are dilations. The matrix $\begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$ corresponds to the dilation in Figure 1.1.

Problem 1. Write a function that accepts an array of points and an array giving the stretching factors in each direction. Your function should return the dilated points.

Included with this lab is a dataset of points saved as a numpy array. To import this array, use `np.load("pi.npy")`. This function will return an array of size $(2, 368)$. This array represents 368 points in $\mathbb{R}^2$ stored as columns.

To check your work, plot the original points and their images under the transformation using the function `plot_transform()` defined below.

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

def plot_transform(original, new):
    """Display a plot of points before and after a transform.

    Inputs:
        original (array) - Array of size (2,n) containing points in R2
            as columns.
        new (array) - Array of size (2,n) containing points in R2
```

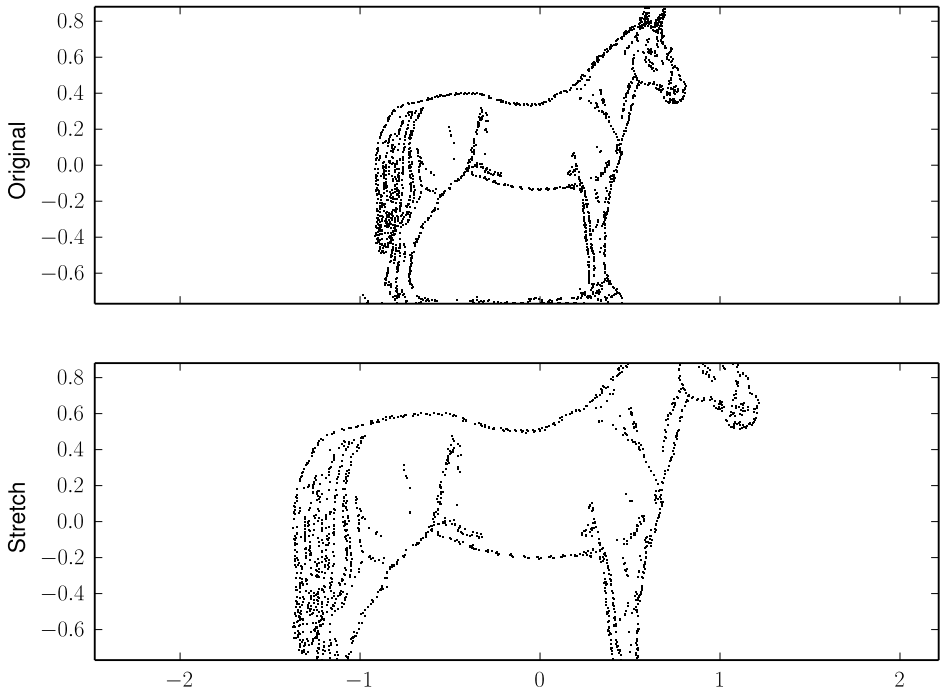

Figure 1.1: An example of a dilation. The top image was stretched by a factor of 1.5 in all directions, producing the bottom image.

```
        as columns.
    """
    v = [-5,5,-5,5]
    plt.subplot(1, 2, 1)
    plt.title('Before')
    plt.gca().set_aspect('equal')
    plt.scatter(original[0], original[1])
    plt.axis(v)
    plt.subplot(1, 2, 2)
    plt.title('After')
    plt.gca().set_aspect('equal')
    plt.scatter(new[0], new[1])
    plt.axis(v)
    plt.show()
```

Rotations

A second type of linear transformation is to rotate vectors around the origin. A rotation of θ radians counterclockwise corresponds to the matrix $\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$. When $\theta = \pi/3$ we get the rotation matrix illustrated in Figure 1.2.

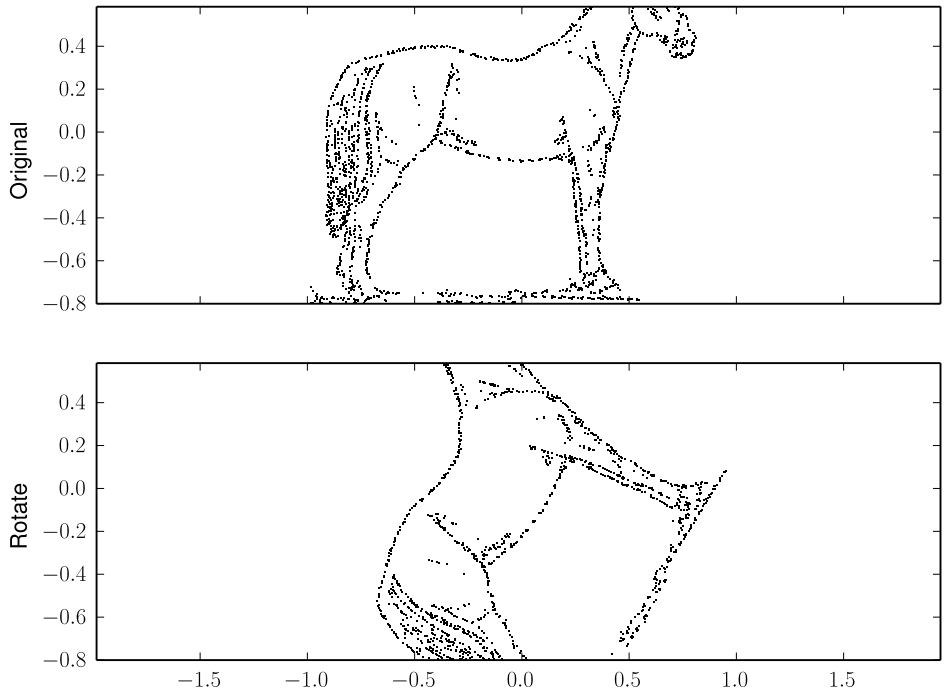

Figure 1.2: An example of a rotation. The top image was rotated by $\pi/3$, producing the bottom image.

Problem 2. Write a function that accepts an array of points and the angle of rotation (in radians). Your function should return the rotated points.

To check your work, plot the original points and their images under the transformation using the function `plot_transform()` defined in Problem 1.

Shears

A third type of linear transformation is a *shear*, which "slants" a set of vectors. The corresponding matrix is a Type III elementary matrix. A horizontal shear has the form $\begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}$ and a vertical skew has the form $\begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix}$. Notice that horizontal skews fix the y-coordinate of a vector while vertical skews fix the x-coordinate. The horizontal shear in Figure 1.3 corresponds to the matrix $\begin{pmatrix} 1 & 1.02 \\ 0 & 1 \end{pmatrix}$.

Reflections

A fourth type of linear transformation is reflections about a line, also called *Householder transformations*. Reflecting about a line spanned by (l_1, l_2) corresponds to

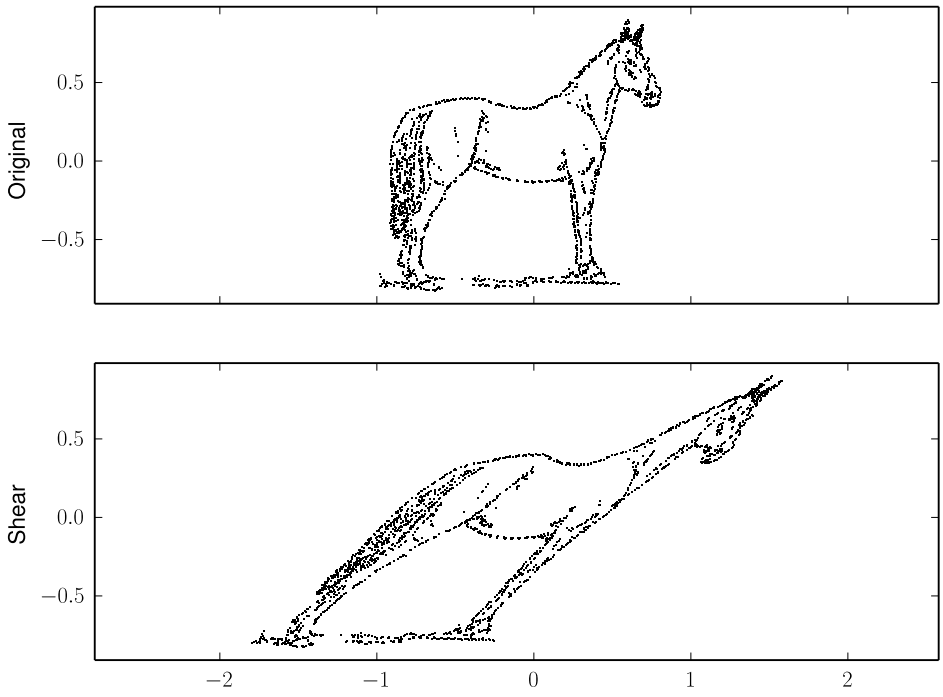

Figure 1.3: An example of a shear. The top image was sheared horizontally to produce the bottom image.

the matrix

$$\frac{1}{l_1^2 + l_2^2} \begin{pmatrix} l_1^2 - l_2^2 & 2l_1 l_2 \\ 2l_1 l_2 & l_2^2 - l_1^2 \end{pmatrix}.$$

For example, the line $y = x$ is spanned by $(1, 1)$. In this case the corresponding matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is a Type I elementary matrix, in fact the only one of size 2×2. As another example, the reflection in Figure **??** about the line $y = (1/\sqrt{3})x$ corresponds to the matrix $\frac{1}{4}\begin{pmatrix} 2 & 2\sqrt{3} \\ 2\sqrt{3} & -2 \end{pmatrix}$.

Composition of linear transformations

Recall that composition of linear transformations corresponds to matrix multiplication. For example, if S is a matrix representing a shear and R is a matrix representing a rotation, then RS represents a shear followed by a rotation.

In fact, any linear transformation of $\mathbb{R}^2$ is a composition of the transformations discussed in this lab. This is because reflections, dilations, and shears provide us with all the elementary matrices, and every matrix is a product of elementary matrices.

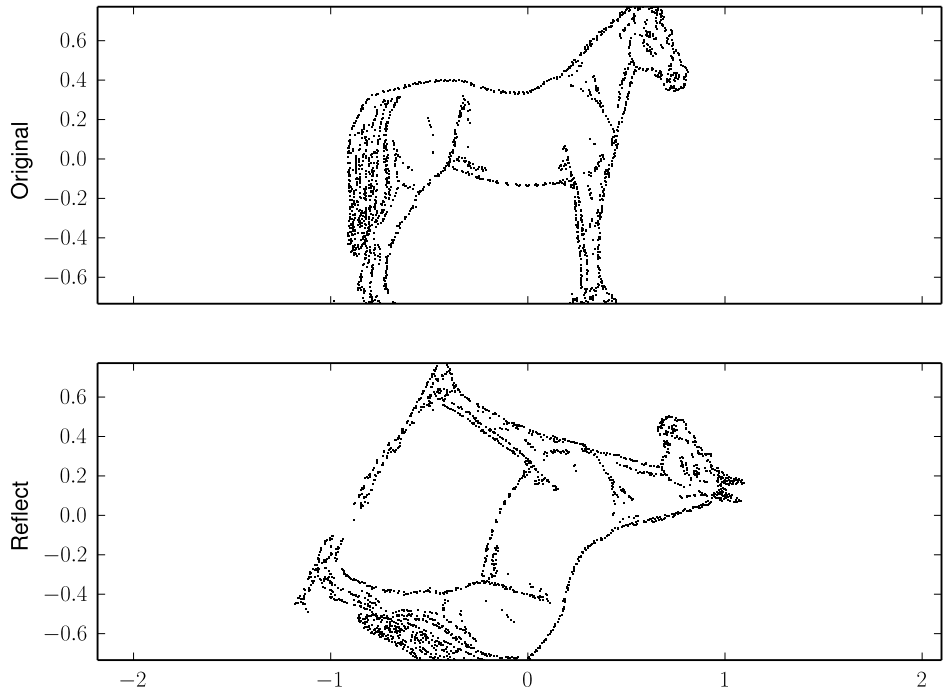

Figure 1.4: An example of a reflection. The top image was reflected about the line $y = (1/\sqrt{3})x$, producing the bottom image.

Affine transformations

Translations

A translation is a map $T : \mathbb{R}^2 \to \mathbb{R}^2$ defined by $T(\mathbf{x}) = \mathbf{x} + \mathbf{b}$ where $\mathbf{b} \in \mathbb{R}^2$. For example, if $\mathbf{b} = (2, 0)^\mathsf{T}$, then applying T to an image will shift it left by 2. This translation is illustrated in Figure 1.5.

Translations are usually NOT linear maps. Therefore, they cannot be represented as matrix multiplication.

Problem 3. Write a function that accepts an array of points and an array indicating how much to shift them in each direction. The function should return the translated points. Hint: You can construct a 2×1 array using the syntax `np.vstack([a,b])`. This may be more convenient for broadcasting. Another hint: To check your work, plot the original points and their images under the transformation using the function `plot_transform()` defined in Problem 1.

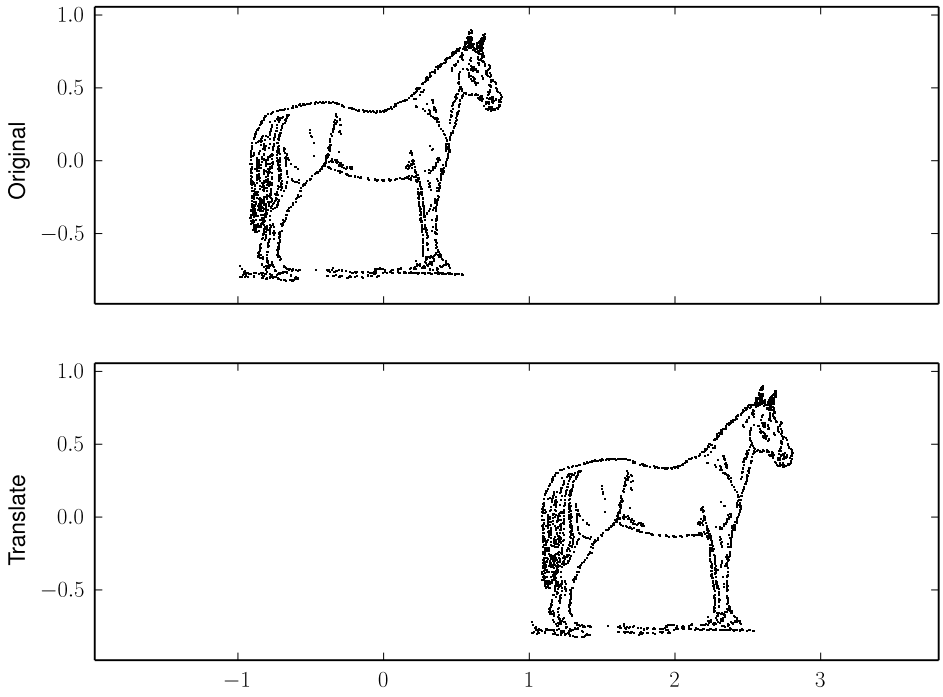

Figure 1.5: An example of a translation. The top image was translated by the vector $(2, 0)^\mathsf{T}$ to produce the bottom image.

Affine transformations

Translations, together with linear transformations, make up the broader class of transformations called "affine transformations." These are transformations of the form $T : \mathbb{R}^2 \to \mathbb{R}^2$, $T(X) = AX + b$ where A is an $n \times n$ matrix and $b \in \mathbb{R}^n$. Affine transformations include all compositions of scalings, rotations, dilations, reflections, and translations. For example, if S represents a shear and R a rotation, and if $\mathbf{b}$ is a vector in $\mathbb{R}^2$, then $T(\mathbf{x}) = RS\mathbf{x} + \mathbf{b}$ first shears $\mathbf{x}$, then rotates it, and finally translates it by $\mathbf{b}$.

Problem 4. Imagine a particle p_1 rotating around a second particle p_2 which is moving through $\mathbb{R}^2$ in a straight line. Suppose p_2 begins at the origin and p_1 begins at $(1, 0)$. We can compute the trajectory of p_1 using affine transformations.

1. Write a function that returns the position of p_1 at a time t. Your function should accept a time t, an angular velocity ω, a direction vector $\mathbf{v}$, and a speed s. Assume p_1 rotates with angular velocity ω and p_2 moves in the direction of $\mathbf{v}$ with speed s. The location of p_1 at time t can be computed as follows:

- Calculate the position of p_2 at time t with the formula $(st/\|\mathbf{v}\|)\mathbf{v}$.
- Calculate the position of p_1 as follows:
  - Rotate p_1 by $t\omega$ radians.
  - Translate the resulting vector by the vector equal to the position of p_2 at time t.

Plot the trajectory of p_1 on the time interval $(0, 10)$ assuming $\omega = \pi$, $v = (1, 1)$, and $s = 2$. Your graph should look something like Figure 1.6.

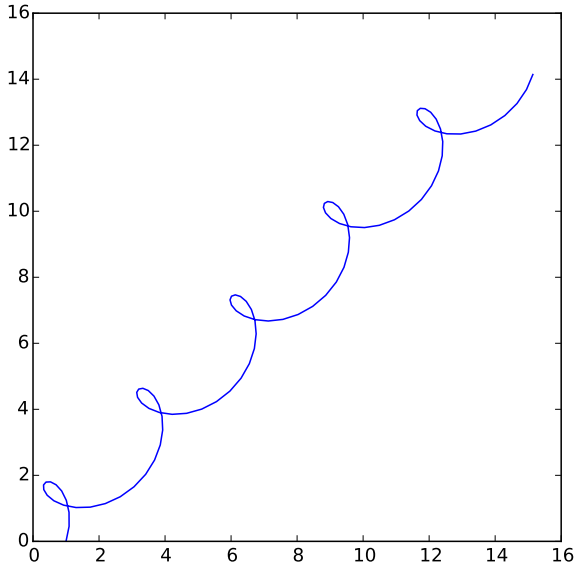

Figure 1.6: Solution to Problem 4.

Linear systems

This section describes efficient algorithms for solving systems of linear equations.

Programming elementry row operations

When you perform a row operation on a matrix, you are really left-multiplying by some elementary matrix. However, matrix multiplication is not the most efficient way to implement row operations. It is much faster to perform row operations by modifying the array in place, and only changing those entries that are affected by the operation. The following code implements the three elementary row operations by modifying the array in place.

```
1 def type_I(A, i, j):
```

```
    """Swap the i-th and j-th rows of A."""
    A[i], A[j] = np.copy(A[j]), np.copy(A[i])

def type_II(A, i, const):
    """Multiply the i-th row of A by const."""
    A[i] *= const

def type_III(A, i, j, const):
    """Add a constant of the j-th row of A to the i-th row."""
    A[i] += const*A[j]
```

row_opers.py

Programming row reduction

When you solve a linear system by reducing the matrix with row operations, it is most efficient to reduce only to row echelon form (REF) [1] and then use back substitution. Here is some code that reduces a matrix to REF.

```
>>> A = np.array([[4., 5., 6., 3.],[2., 4., 6., 4.],[7., 8., 0., 5.]])
array([[ 4.,  5.,  6.,  3.],
       [ 2.,  4.,  6.,  4.],
       [ 7.,  8.,  0.,  5.]])
>>> A[1] -= (A[1,0]/A[0,0]) * A[0]
>>> A[2] -= (A[2,0]/A[0,0]) * A[0]
>>> A[2,1:] -= (A[2,1]/A[1,1]) * A[1,1:]
>>> A
array([[ 4. ,  5. ,  6. ,  3. ],
       [ 0. ,  1.5,  3. ,  2.5],
       [ 0. ,  0. , -9. ,  1. ]])
```

In the third row operation modified only a part of the third row because we knew that the first value would still be 0. Modifying only those entries of the array that are affected by a row operation will save a lot of time when you are row reducing a large matrix.

Beware that round-off error in row reduction can cause serious problems. Suppose we wish to row reduce a matrix A as follows.

```
>>> A = np.array([[4., 5., 6., 3.],[2., 2.5, 6., 4.],[7., 8., 0., 5.]])
array([[ 4.,  5.,  6.,  3.],
       [ 2.,  2.5,  6.,  4.],
       [ 7.,  8.,  0.,  5.]])
>>> A[1] -= (A[1,0]/A[0,0]) * A[0]
>>> A[2] -= (A[2,0]/A[0,0]) * A[0]
```

If we work this out by hand, at this point we have

$$A = \begin{pmatrix} 4 & 5 & 6 & 3 \\ 0 & 0 & 3 & 2.5 \\ 0 & -7.5 & -10.5 & -.25 \end{pmatrix}. \tag{1.1}$$

If we swap the second and third rows, then the matrix is in row echelon form. However, suppose that due to round-off error, the machine instead computes

[1] We do not require leading coefficients to be 1 in the REF.

$$A = \begin{pmatrix} 4 & 5 & 6 & 3 \\ 0 & 10^{-15} & 3 & 2.5 \\ 0 & -7.5 & -10.5 & -.25 \end{pmatrix}.$$

The algorithm would then attempt to pivot on the `A[1,1]` entry as follows.

```
>>> A[2,1:] -= (A[2,1]/A[1,1]) * A[1,1:]
>>> A
array([[ 4. ,  5.0e+00 , 6.00e+00 ,  3.000e-00 ],
       [ 0. ,  1.0e-14 , 3.00e+00 ,  2.500e-00 ],
       [ 0. ,  0.0e+00 , 2.25e+14 ,  1.875e+14 ]])
```

The round-off error in the `A[1,1]` entry has affected the third row, and the matrix is now much different than the correct answer in (1.1). In larger matrices, round-off error can propagate through many steps in a calculation, resulting in garbage output.

Some of NumPy's matrix algorithms use row reduction. NumPy's methods use several clever tricks to minimize the impact of round-off errors. However, these methods may still give you garbage output due to round-off error, especially if the matrix is *ill-conditioned*. You should always be aware of this possibility.

Problem 5. Write a function which reduces a square matrix to REF. You may make the following assumptions:

1. The matrix is invertible.
2. During your row reduction, a zero will never appear on the main diagonal.
3. All round-off errors may be ignored.

During a row operation, do not modify any entries that you know will be zero before and after the operation.

The LU decomposition

The LU decomposition of a square matrix A is a factorization $A = LU$ where U is an upper triangular and L is a lower triangular matrix. In fact, U is the REF of A and L is a product of Type III elementary matrices whose inverses reduce A to U. Thus, the LU decomposition is a way of storing the REF and "how we got there."

The LU factorization of A exists only when A can be reduced to REF using only Type III elementary matrices (no row swaps). However, we can always permute the rows of A to obtain a matrix that has an LU decomposition. If P encodes the row swaps, then a decomposition $PA = LU$ always exists.

If A has an LU decomposition (not requiring row swaps), then we can find it as follows. Reduce A to REF with k row operations, corresponding to matrices $E_1, \dots, E_k$. Then $U = E_k \dots E_2 E_1 A$, where U is the REF of A. Because there were

no row swaps, each E_i is a lower triangular Type III matrix. Then E_i^{-1} is also lower triangular. Thus, $L = (E_k \dots E_2 E_1)^{-1}$ is lower triangular, and $LU = A$.

Because $L = (E_k \dots E_2 E_1)^{-1} = I E_1^{-1} E_2^{-1} \dots E_k^{-1}$, we can compute L by right-multiplying the identity by the matrices we used to reduce U. In fact, in this special situation, each right-multiplication will change only one entry of L. We have the following algorithm for the LU decomposition, assuming it exists.

Algorithm 1.1 The algorithm for LU decomposition of a matrix A. This algorithm returns lower triangular L and upper triangular U such that $A = LU$.

```
1: procedure LU Decomposition(A)
2:     m, n ← shape(A)
3:     U ← copy(A)
4:     L ← Id(n)
5:     for i = 1 ... n − 1 do
6:         for j = 0 ... i − 1 do
7:             L[i, j] ← U[i, j]/U[j, j]
8:             U[i, j :] ← U[i, j :] − L[i, j]U[j, j :]
9:     return L, U
```

Problem 6. Write a function that finds the LU decomposition of a square matrix. You may assume that the matrix has an LU decomposition.

The LU decomposition can be performed in-place by storing U on and above the main diagonal of the array and storing L below it. The main diagonal of L does not need to be stored since all its entries are ones.

Applications of the LU decomposition

The LU decomposition is a more efficient way to solve linear systems than row reduction, and it can also be used to quickly compute inverses and determinants. SciPy implements these methods in the `linalg` module, specifically in the functions `linalg.lu_factor`, `linalg.solve`, `linalg.inv`, and `linalg.det`.

Let us see how to use the LU decomposition to solve matrix equations. Suppose that after row swaps, A has the decomposition $PA = LU$. Then $Ax = b$ is equivalent to $LUx = Pb$. We can solve this system by first solving $Ly = Pb$ and then $Ux = y$. Since L and U are triangular, these systems can be solved with backward and forward substitution, which is faster than row reduction. Thus, we can perform row reduction once to compute the LU factorization of A, and then we can use substitution to solve $Ax = b$ many different values of b. This technique is significantly faster than storing A^{-1} and then computing $A^{-1}b$ (see Problem 7).

Problem 7. In this problem you will solve the system $Ax = b$ for fixed A and many different values of b. You will do this in two ways. For a random

1000×1000 array A and a random 1000×500 array B do the following:

1. Time `la.lu_factor(A)`.
2. Time `la.inv(A)`.
3. Store the output of `la.lu_factor()`. Time `la.lu_solve()` on this stored output and B.
4. Store the inverse of A. Time how long it takes to multiply A^{-1} by B. Print each of these times. What can you conclude about the more efficient way to solve linear systems?

Our technique for solving linear equations can also be used to invert matrices. We can compute A^{-1} by solving $LUx_i = P_i$ for every P_i that is a column of P. Then A^{-1} is the matrix with columns $x_1, x_2, \ldots, x_n$.

Finally, the LU decomposition also gives us an efficient way to compute determinants. For if $PA = LU$, then $\det(A) = [\det(P)]^{-1} \det(L) \det(U)$. But the determinant of a triangular matrix is the product of its diagonal entries, and every diagonal entry of L is 1. Also, $\det(P)$ is the number of row swaps we applied to A to put it in the appropriate form. So if U has diagonal entries u_{ii} for $i = 1, \ldots, n$, then

$$\det(A) = (-1)^S \left(\prod_{i=1}^{n} u_{ii} \right),$$

where S is the number of row-swaps.