

Lab 5

Linear Systems

Lab Objective: *The fundamental problem of linear algebra is solving the linear system $A\mathbf{x} = \mathbf{b}$, if it is even possible. There are many approaches to solving this problem, each with different pros and cons. In this lab we implement the LU decomposition and use it to solve square linear systems. We also introduce SciPy, together with its libraries for linear algebra and working with sparse matrices.*

Gaussian Elimination

The standard approach for solving the linear system $A\mathbf{x} = \mathbf{b}$ on paper is reducing the augmented matrix $[A \mid \mathbf{b}]$ to row-echelon form (REF) via *Gaussian elimination*, then using back substitution. The matrix is in REF when the leading non-zero term in each row is the diagonal term, so the matrix is upper triangular.

At each step of the process, there are three possible operations: swapping two rows, multiplying one row by a scalar value, or adding a scalar multiple of one row to another. Many systems, like the one displayed below, can be reduced to REF using only the third type of operation. First use multiples of the first row to get zeros below the diagonal in the first column, then use a multiple of the second row to get zeros below the diagonal in the second column.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ \color{red}{0} & 3 & 1 & 2 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ \color{red}{0} & 3 & 4 & 5 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & \color{red}{0} & 3 & 3 \end{array} \right]$$

Each of these operations is mathematically equivalent to left-multiplying by a *type III elementary matrix*, the identity with one non-diagonal non-zero term. If row operation k corresponds to matrix E_k , the following equation is $E_3E_2E_1A = U$.

$$\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{array} \right] \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{array} \right] \left[\begin{array}{ccc} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array} \right]$$

However, matrix multiplication is an inefficient way to implement row reduction. Instead, modify the matrix *in place* (without making a copy), changing only those entries that are affected by each row operation.

```

>>> import numpy as np
>>> A = np.array([[1, 1, 1, 1],
...               [1, 4, 2, 3],
...               [4, 7, 8, 9]], dtype=np.float)

# Reduce the 0th column to zeros below the diagonal.
>>> A[1,0:] -= (A[1,0] / A[0,0]) * A[0]
>>> A[2,0:] -= (A[2,0] / A[0,0]) * A[0]

# Reduce the 1st column to zeros below the diagonal.
>>> A[2,1:] -= (A[2,1] / A[1,1]) * A[1,1:]
>>> print(A)
[[ 1.  1.  1.  1.]
 [ 0.  3.  1.  2.]
 [ 0.  0.  3.  3.]]

```

Note that the final row operation modifies only part of the third row to avoid spending the computation time of adding 0 to 0.

If a 0 appears on the main diagonal during any part of row reduction, the approach given above tries to divide by 0. Swapping the current row with one below it that does not have a 0 in the same column solves this problem. This is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

Problem 1. Write a function which reduces a square matrix A to REF. You may assume that A is invertible and that a 0 will never appear on the main diagonal (so only use type III row reductions, not type II). Avoid operating on entries that you know will be 0 before and after a row operation.

Consider generating small random matrices as test cases with NumPy's `random` module (`np.random.randint()` may be particularly useful).

ACHTUNG!

Gaussian elimination is not always numerically stable. Suppose that, due to roundoff error, we have a matrix with a very small entry on the diagonal.

$$A = \begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix}$$

10^{-15} is essentially zero, but instead of swapping the first and second rows to put the A in REF, a computer might multiply the first row by 10^{15} and add it to the second row to eliminate the -1 . The resulting matrix is far from what we would expect on paper.

$$\begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 10^{-15} & 1 \\ 0 & 10^{15} \end{bmatrix}$$

Round-off error can propagate through many steps in a calculation. NumPy's routines that employ row reduction use several tricks to minimize the impact of round-off errors, but beware that these tricks can't fix every matrix.

The LU Decomposition

The *LU decomposition* of a square matrix A is a factorization $A = LU$ where U is the **upper** triangular REF of A and L is the **lower** triangular product of the type III elementary matrices whose inverses reduce A to U . The LU decomposition of A exists when A can be reduced to REF using only type III elementary matrices (no row swaps). However, the rows of A can always be permuted in a way such that the decomposition exists. If P is a permutation matrix encoding the appropriate row swaps, then the decomposition $PA = LU$ always exists.

Suppose A has an LU decomposition (not requiring row swaps). Then A can be reduced to REF with k row operations, corresponding to left-multiplying the type III elementary matrices E_1, \dots, E_k . Because there were no row swaps, each E_i is lower triangular, so each inverse E_i^{-1} is also lower triangular. Furthermore, since the product of lower triangular matrices is lower triangular, L is lower triangular.

$$\begin{aligned} E_k \dots E_2 E_1 A = U &\longrightarrow A = (E_k \dots E_2 E_1)^{-1} U \\ &= E_1^{-1} E_2^{-1} \dots E_k^{-1} U \\ &= LU \end{aligned}$$

We can thus compute L by right-multiplying the identity by the matrices used to reduce U . However, in this special situation, each right-multiplication only changes one entry of L , so we can avoid matrix multiplication altogether. The entire process, only slightly different than row reduction, is summarized below.

Algorithm 5.1

```

1: procedure LU_DECOMPOSITION( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                 ▷ Store the dimensions of  $A$ .
3:    $U \leftarrow \text{copy}(A)$                                     ▷ Make a copy of  $A$  with np.copy().
4:    $L \leftarrow I_m$                                          ▷ The  $m \times m$  identity matrix.
5:   for  $j = 0 \dots n - 1$  do
6:     for  $i = j + 1 \dots m - 1$  do
7:        $L_{i,j} \leftarrow U_{i,j} / U_{j,j}$ 
8:        $U_{i,j:} \leftarrow U_{i,j:} - L_{i,j} U_{j,j:}$ 
9:   return  $L, U$ 

```

Problem 2. Write a function that finds the LU decomposition of a square matrix. You may assume the decomposition exists and requires no row swaps.

Forward and Backward Substitution

If $A\mathbf{x} = \mathbf{b}$ and $PA = LU$, then $LU\mathbf{x} = PA\mathbf{x} = P\mathbf{b}$. This system can be solved by first solving $L\mathbf{y} = P\mathbf{b}$, then $U\mathbf{x} = \mathbf{y}$. Since L and U are both triangular, these systems can be solved with backward and forward substitution. We can thus compute the LU factorization of A once, then use substitution to efficiently solve $A\mathbf{x} = \mathbf{b}$ for various values of \mathbf{b} .

Since the diagonal entries of L are all 1, the triangular system $L\mathbf{y} = \mathbf{b}$ has the following form:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Matrix multiplication yields the following equations:

$$\begin{aligned} y_1 &= b_1 & y_1 &= b_1 \\ l_{21}y_1 + y_2 &= b_2 & y_2 &= b_2 - l_{21}y_1 \\ \vdots & & \vdots & \\ \sum_{j=1}^{k-1} l_{kj}y_j + y_k &= b_k & y_k &= b_k - \sum_{j=1}^{k-1} l_{kj}y_j \end{aligned} \quad (5.1)$$

The triangular system $U\mathbf{x} = \mathbf{b}$ yields similar equations, but in reverse order:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

$$\begin{aligned} u_{nn}x_n &= y_n & x_n &= \frac{1}{u_{nn}}y_n \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} & x_{n-1} &= \frac{1}{u_{n-1,n-1}}(y_{n-1} - u_{n-1,n}x_n) \\ \vdots & & \vdots & \\ \sum_{j=k}^n u_{kj}x_j &= y_k & x_k &= \frac{1}{u_{kk}} \left(y_k - \sum_{j=k+1}^n u_{kj}x_j \right) \end{aligned} \quad (5.2)$$

Problem 3. Write a function that, given A and \mathbf{b} , solves the square linear system $A\mathbf{x} = \mathbf{b}$. Use the function from Problem 2 to compute L and U , then use Equations 5.1 and 5.2 to solve for \mathbf{y} , then \mathbf{x} . You may again assume that there are no row swaps (so $P = I$ in this case).

SciPy

SciPy is a powerful scientific computing library built upon NumPy. It includes high-level tools for linear algebra, statistics, signal processing, integration, optimization, machine learning, and more.

SciPy is typically imported with the convention `import scipy as sp`. However, SciPy is set up in a way that requires its submodules to be imported separately.¹

Linear Algebra

NumPy and SciPy both have a linear algebra module, each called `linalg`, but SciPy's module is the larger of the two. Some of SciPy's common `linalg` functions are listed below. See <http://docs.scipy.org/doc/scipy/reference/linalg.html> for more documentation.

Function	Returns
<code>det()</code>	The determinant of a square matrix.
<code>eig()</code>	The eigenvalues and eigenvectors of a square matrix.
<code>inv()</code>	The inverse of an invertible matrix.
<code>norm()</code>	The norm of a vector or matrix norm of a matrix.
<code>solve()</code>	The solution to $A\mathbf{x} = \mathbf{b}$ (the system may not be square).

This library also includes routines for computing matrix decompositions.

```
>>> from scipy import linalg as la

# Make a random matrix and a random vector.
>>> A = np.random.random((1000,1000))
>>> b = np.random.random(1000)

# Compute the LU decomposition of A, including pivots.
>>> lu, piv = la.lu_factor(A)

# Use the LU decomposition to solve Ax = b.
>>> x = la.lu_solve((lu,piv), b)

# Check that the solution is legitimate.
>>> np.allclose(A.dot(x), b)
True
```

As with NumPy, SciPy's routines are all highly optimized. However, some algorithms are, by nature, faster than others.

¹SciPy modules like `linalg` are really *packages*, which need to be initialized separately.

Problem 4. Write a function that times different `scipy.linalg` functions for solving square linear systems.

For various values of n , generate a random $n \times n$ matrix A and a random $n \times 1$ vector \mathbf{b} using `np.random.random()`. Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

1. Invert A with `la.inv()` and left-multiply the inverse to \mathbf{b} .
2. Use `la.solve()`.
3. Use `la.lu_factor()` and `la.lu_solve()` to solve the system with the LU decomposition.
4. Use `la.lu_factor()` and `la.lu_solve()`, but only time `la.lu_solve()`.

Plot the system size n versus the execution times. Use log scales if needed.

NOTE

Numerically inverting matrices is so costly that there is hardly ever a good reason to do it. Use a specific solver, like `la.lu_solve()`, whenever possible.

Sparse Matrices

Large linear systems can have tens of thousands of parameters. Storing the corresponding matrices in memory can be difficult: a 100000×100000 system requires around 40 GB to store in a NumPy array (4 bytes per entry $\times 10^{10}$ entries). This is well beyond the amount of RAM in a normal laptop.

In applications where systems of this size arise, it is common that such matrices are *sparse*, meaning that most of the entries are 0. SciPy's `sparse` module provides tools for efficiently constructing and manipulating 1- and 2-D sparse matrices. A `sparse` matrix only stores the nonzero values and the positions of these values. For sufficiently sparse matrices, storing the matrix as a `sparse` matrix may only take megabytes, rather than gigabytes.

For example, diagonal matrices are sparse. Storing an $n \times n$ diagonal matrix in the naïve way means storing n^2 values in memory. It is almost always more efficient to instead store the diagonal entries in a 1-D array of n values. In addition to using less storage space, this allows for much faster matrix operations: the standard algorithm to multiply a matrix by a diagonal matrix involves n^3 steps, but most of these are multiplying by or adding 0. A smarter algorithm can accomplish the same task much faster.

SciPy has seven `sparse` matrix types. Each type is optimized either for storing sparse matrices whose nonzero entries follow certain patterns, or for performing certain computations.

Name	Description	Advantages
<code>bsr_matrix</code>	Block Sparse Row	Specialized structure.
<code>coo_matrix</code>	Coordinate Format	Conversion among sparse formats.
<code>csc_matrix</code>	Compressed Sparse Column	Column-based operations and slicing.
<code>csr_matrix</code>	Compressed Sparse Row	Row-based operations and slicing.
<code>dia_matrix</code>	Diagonal Storage	Specialized structure.
<code>dok_matrix</code>	Dictionary of Keys	Element access, incremental construction.
<code>lil_matrix</code>	Row-based Linked List	Incremental construction. Conversion to CSR or CSC format.

Creating Sparse Matrices

A regular, non-sparse matrix is called *full* or *dense*. Full matrices can be converted to each of the sparse matrix formats listed above. However, it is more memory efficient to never create the full matrix in the first place. There are really three different approaches for creating sparse matrices from scratch.

- **Coordinate Format:** When all of the nonzero values and their positions are known, create the entire sparse matrix at once as a `coo_matrix`. All nonzero values are stored as a location (or coordinate) and value. This format also supports very fast conversion between the other sparse matrix types.

```
>>> from scipy import sparse

# Define the rows, columns, and values separately.
>>> rows = np.array([0, 1, 0])
>>> cols = np.array([0, 1, 1])
>>> vals = np.array([3, 5, 2])
>>> A = sparse.coo_matrix((vals, (rows,cols)), shape=(3,3))
>>> print(A)
(0, 0)    3
(1, 1)    5
(0, 1)    2

# The toarray() method casts the sparse matrix as a NumPy array.
# However, using regular arrays forfeits all sparse-related optimizations.
>>> print(A.toarray())
[[3 2 0]
 [0 5 0]
 [0 0 0]]
```

- **DOK and LIL Formats:** If the matrix values and their locations are not known a priori, construct the matrix incrementally with `dok_matrix` or `lil_matrix`. Indicate the size of the matrix, then change individual values with regular slicing syntax.

```
>>> B = sparse.lil_matrix((2,6))
>>> B[0,2] = 4
>>> B[1,3:] = 9

>>> print(B.toarray())
[[ 0.  0.  4.  0.  0.  0.]
 [ 0.  0.  0.  9.  9.  9.]]
```

- **BSR and DIA Formats:** When the matrix has a particular sparsity pattern, such as nonzero blocks or diagonals, use `bsr_matrix` or `dia_matrix`. The functions `sparse.bmat()` and `sparse.diags()` are convenient ways to create these structures.

```
# Use sparse.bmat() to create a block matrix. Use 'None' for zero blocks.
>>> A = sparse.coo_matrix(np.ones((2,2)))
>>> B = sparse.coo_matrix(np.full((2,2), 2.))
>>> print(sparse.bmat([[ A , None, A ],
                      [None, B , None]], format='bsr').toarray())
[[ 1.,  1.,  0.,  0.,  1.,  1.],
 [ 1.,  1.,  0.,  0.,  1.,  1.],
 [ 0.,  0.,  2.,  2.,  0.,  0.],
 [ 0.,  0.,  2.,  2.,  0.,  0.]]

# Use sparse.diags() to create a matrix with diagonal entries.
>>> diagonals = [[1,2],[3,4,5],[6]] # List the diagonal entries.
>>> offsets = [ -1 , 0 , 3 ] # Specify the diagonal they go on.
>>> print(sparse.diags(diagonals, offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  4.  0.  0.]
 [ 0.  2.  5.  0.]]

# If all of the diagonals have the same entry, specify the entry alone.
>>> print(sparse.diags([1,3,6], offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  3.  0.  0.]
 [ 0.  1.  3.  0.]]
```

NOTE

A *banded* matrix is a square matrix whose only non-zero entries are on the main diagonal and on some diagonals on either side. If the nonzero entries are confined to the three central diagonals, the matrix is also called *tri-diagonal*. Banded matrices arise naturally in many applications, including numerical methods for solving differential equations.

Problem 5. Write a function that accepts an integer n and returns a sparse $n \times n$ tri-diagonal array A with 2's along the main diagonal and -1 's along the first subdiagonal and the first superdiagonal.

$$A = \begin{bmatrix} 2 & -1 & \cdots & 0 & 0 \\ -1 & 2 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}$$

Sparse Matrix Operations

Once a sparse matrix has been constructed, it should be converted to a `csr_matrix` or `csc_matrix` with the matrix's `tocsr()` or `tocsc()` method. The CSR and CSC formats are optimized for row or column operations, respectively. To choose the correct format to use, determine what direction the matrix will be traversed.

For example, in the matrix-matrix multiplication AB , A is traversed row-wise, but B is traversed column-wise. Thus A should be converted to a `csr_matrix` and B should be converted to a `csc_matrix`.

```
# Initialize a sparse matrix incrementally as a lil_matrix.
>>> A = sparse.lil_matrix((10000,10000))
>>> for k in xrange(10000):
...     A[np.random.randint(0,9999), np.random.randint(0,9999)] = k
...
>>> A
<10000x10000 sparse matrix of type '<type 'numpy.float64'>'
  with 9999 stored elements in LInked List format>

# Convert A to CSR and CSC formats to compute the matrix product AA.
>>> Acsr = A.tocsr()
>>> Acsc = A.tocsc()
>>> Acsr.dot(Acsc)
<10000x10000 sparse matrix of type '<type 'numpy.float64'>'
  with 10142 stored elements in Compressed Sparse Row format>
```

Beware that row-based operations on a `csc_matrix` are very slow, and similarly, column-based operations on a `csr_matrix` are very slow.

ACHTUNG!

Many familiar NumPy operations have analogous routines in the `sparse` module. These methods take advantage of the sparse structure of the matrices and are, therefore, usually significantly faster. However, SciPy's `sparse` matrices behave a little differently than NumPy arrays.

Operation	numpy	scipy.sparse
Component-wise Addition	<code>A + B</code>	<code>A + B</code>
Scalar Multiplication	<code>2 * A</code>	<code>2 * A</code>
Component-wise Multiplication	<code>A * B</code>	<code>A.multiply(B)</code>
Matrix Multiplication	<code>A.dot(B)</code>	<code>A * B, A.dot(B)</code>

Note in particular the difference between `A * B` for NumPy arrays and SciPy sparse matrices. Do **not** use `np.dot()` to try to multiply sparse matrices, as it may treat the inputs incorrectly. The syntax `A.dot(B)` is safest in most cases.

Finally, SciPy has a linear algebra library specifically for sparse matrices, called `scipy.sparse.linalg`. Like other SciPy modules, it must be imported specifically.

```
>>> from scipy.sparse import linalg as spla
```

Problem 6. Write a function that times regular and sparse linear system solvers.

For various values of n , generate the $n \times n$ matrix A described in Problem 5 and a random $n \times 1$ vector \mathbf{b} . Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

1. Convert A to CSR format, then use `scipy.sparse.linalg.spsolve()`.
2. Convert A to a NumPy array, then use `scipy.linalg.solve()`.

Plot the system size n versus the execution times. As always, use log scales where appropriate and use a legend to label each line.

ACHTUNG!

Because there are fast algorithms for solving a tri-diagonal linear system, you may think that there are fast algorithms for inverting a tri-diagonal matrix. In fact this is not true, and the inverse of a sparse matrix is usually not sparse. There is rarely a good reason to invert a matrix, sparse or dense.

NOTE

One way to view a sparse matrix is to convert it to a NumPy array with the `toarray()` method. If a matrix is too large to fit in memory as an array, it can still be visualized with Matplotlib's `plt.spy()`, which colors in the locations of the non-zero entries of the matrix. Try the following code for visualizing a simple banded matrix.

```
>>> from matplotlib import pyplot as plt
>>> A = sparse.diags([1, 2, 3, 4, 5], [-5000, -1000, 0, 1000, 5000],
...                  shape=(10000, 10000))
>>> plt.spy(A, markersize=1)
>>> plt.show()
```

See <http://docs.scipy.org/doc/scipy/reference/sparse.html> for additional details on SciPy's sparse module.

Additional Material

Improvements on the LU Decomposition

Vectorization

Algorithm 5.1 uses two loops to compute the LU decomposition. With a little vectorization, the process can be reduced to a single loop.

Algorithm 5.2

```

1: procedure FAST LU DECOMPOSITION( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:   for  $k = 0 \dots n - 1$  do
6:      $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$ 
7:      $U_{k+1:,k:} \leftarrow U_{k+1:,k:} - L_{k+1:,k} U_{k,k:}^\top$ 
8:   return  $L, U$ 

```

Note that step 7 is an *outer product*, not the regular dot product ($\mathbf{x}\mathbf{y}^\top$ instead of the usual $\mathbf{x}^\top\mathbf{y}$). Use `np.outer()` instead of `np.dot()` to get the desired result.

Pivoting

Gaussian elimination iterates through the rows of a matrix, using the diagonal entry $x_{k,k}$ of the matrix at the k th iteration to zero out all of the entries in the column below $x_{k,k}$ ($x_{i,k}$ for $i \geq k$). This diagonal entry is called the *pivot*. Unfortunately, Gaussian elimination, and hence the LU decomposition, can be very numerically unstable if at any step the pivot is a very small number. Most professional row reduction algorithms avoid this problem via *partial pivoting*.

The idea is to choose the largest number (in magnitude) possible to be the pivot by swapping the pivot row² with another row before operating on the matrix. For example, the second and fourth rows of the following matrix are exchanged so that the pivot is -6 instead of 2 .

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

A row swap is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix}$$

² *Complete pivoting* involves row and column swaps, but doing both operations is usually considered overkill.

For the LU decomposition, if the permutation matrix at step k is P_k , then $P = P_k \dots P_2 P_1$ yields $PA = LU$. The complete algorithm is given below.

Algorithm 5.3

```

1: procedure LU DECOMPOSITION WITH PARTIAL PIVOTING( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:    $P \leftarrow [0, 1, \dots, n-1]$  ▷ See tip 2 below.
6:   for  $k = 0 \dots n-1$  do
7:     Select  $i \geq k$  that maximizes  $|U_{i,k}|$ 
8:      $U_{k,k} \leftrightarrow U_{i,k}$  ▷ Swap the two rows.
9:      $L_{k,:k} \leftrightarrow L_{i,:k}$  ▷ Swap the two rows.
10:     $P_k \leftrightarrow P_i$  ▷ Swap the two entries.
11:     $L_{k+1:,k} \leftarrow U_{k+1:,k}/U_{k,k}$ 
12:     $U_{k+1:,k} \leftarrow U_{k+1:,k} - L_{k+1:,k}U_{k,k}^\top$ 
13:   return  $L, U, P$ 

```

The following tips may be helpful for implementing this algorithm:

1. Since NumPy arrays are mutable, use `np.copy()` to reassign the rows of an array simultaneously.
2. Instead of storing P as an $n \times n$ array, fancy indexing allows us to encode row swaps in a 1-D array of length n . Initialize P as the array $[0, 1, \dots, n]$. After performing a row swap on A , perform the same operations on P . Then the matrix product PA will be the same as $A[P]$.

```

>>> A = np.zeros(3) + np.vstack(np.arange(3))
>>> P = np.arange(3)
>>> print(A)
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

# Swap rows 1 and 2.
>>> A[1], A[2] = np.copy(A[2]), np.copy(A[1])
>>> P[1], P[2] = P[2], P[1]

>>> print(A) # A with the new row arrangement.
[[ 0.  0.  0.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]

>>> print(P) # The permutation of the rows.
[0 2 1]

>>> print(A[P]) # A with the original row arrangement.
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

```

There are potential cases where even partial pivoting does not eliminate catastrophic numerical errors in Gaussian elimination, but the odds of having such an amazingly poor matrix are essentially zero. The numerical analyst J.H. Wilkinson captured the likelihood of encountering such a matrix in a natural application when he said, “Anyone that unlucky has already been run over by a bus!”

In Place

The LU decomposition can be performed in place (overwriting the original matrix A) 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 of its entries are 1. This format saves an entire array of memory, and is how `scipy.linalg.lu_factor()` returns the factorization.

More Applications of the LU Decomposition

The LU decomposition can also be used to compute inverses and determinants.

- **Inverse:** $(PA)^{-1} = (LU)^{-1} \longrightarrow A^{-1}P^{-1} = U^{-1}L^{-1} \longrightarrow LUA^{-1} = P$. Solve $LU\mathbf{a}_i = \mathbf{p}_i$ with forward and backward substitution (as in Problem 3) for every column \mathbf{p}_i of P . Then

$$A^{-1} = \left[\begin{array}{c|c|c|c} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{array} \right],$$

the matrix where \mathbf{a}_k is the k th column.

- **Determinant:** $\det(A) = \det(P^{-1}LU) = \frac{\det(L)\det(U)}{\det(P)}$. The determinant of a triangular matrix is the product of its diagonal entries. Since every diagonal entry of L is 1, $\det(L) = 1$. Also, P is just a row permutation of the identity matrix (which has determinant 1), and a single row swap negates the determinant. Thus

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

where S is the number of row swaps.

The Cholesky Decomposition

A square matrix A is called *positive definite* if $\mathbf{z}^T A \mathbf{z} > 0$ for all nonzero vectors \mathbf{z} . In addition, A is called *Hermitian* if $A = A^H = \overline{A^T}$. If A is Hermitian positive definite, it has a *Cholesky Decomposition* $A = U^H U$ where U is upper triangular with real, positive entries on the diagonal. This is the matrix equivalent to taking the square root of a positive real number.

The Cholesky decomposition takes advantage of the conjugate symmetry of A to simultaneously reduce the columns *and* rows of A to zeros (except for the diagonal). It thus requires only half of the calculations and memory of the LU decomposition. Furthermore, the algorithm is *numerically stable*, which means that round-off errors

do not propagate throughout the computation. Because of its efficiency and stability, this decomposition is used when possible to solve least squares, optimization, and state estimation problems.

Algorithm 5.4

```

1: procedure CHOLESKY DECOMPOSITION( $A$ )
2:    $U \leftarrow A$  ▷ Copy  $A$  if desired.
3:   for  $i = 0 \dots n - 1$  do
4:     for  $j = i + 1 \dots n - 1$  do
5:        $U_{j,j} \leftarrow U_{j,j} - U_{i,j} \overline{U_{ij}} / U_{ii}$ 
6:      $U_{i,i} \leftarrow U_{i,i} / \sqrt{U_{ii}}$ 
7:   return  $U$ 

```

As with the LU decomposition, SciPy's `linalg` module has optimized routines, `la.cho_factor()` and `la.cho_solve()`, for using the Cholesky decomposition.