CHAPTER

7

Gaussian Elimination

OUTLINE

Vaughn's been working on a couple of new pitches, the Eliminator and the Humilator, to complement his fastball, the Terminator.

***"Harry Doyle" in the movie* Major League II**

CHAPTER POINTS

- The most natural way to solve systems of linear equations by hand can be generalized into the algorithm known as Gaussian elimination.
- This algorithm works well for almost all systems, if we allow the order of the equations to be rearranged.
- The number of operations needed to perform Gaussian elimination scales as the number of equations cubed, $O(n^3)$, though on moderately-sized matrices we observe slightly lower growth in the time to solution.

7.1 A MOTIVATING EXAMPLE

In this chapter we will be interested in computing the solution to a system of linear, algebraic equations. The solution of such systems is the basis for many other numerical tech-

niques. For example, the solution of nonlinear systems is often reduced to the solution of successive linear systems that approximate the nonlinear system.

We desire to write a generic algorithm for solving linear systems. We want the systems to have arbitrary size, and arbitrary values for the coefficients. To develop this algorithm, we will start with a concrete example, and then generalize our approach.

We will begin with the following system:

$$3x_1 + 2x_2 + x_3 = 6, \tag{7.1a}$$
$$-x_1 + 4x_2 + 5x_3 = 8, \tag{7.1b}$$
$$2x_1 - 8x_2 + 10x_3 = 4. \tag{7.1c}$$

In matrix form this looks like

$$\begin{pmatrix} 3 & 2 & 1 \\ -1 & 4 & 5 \\ 2 & -8 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 4 \end{pmatrix}.$$

Another way to write this system is a notational shorthand called an augmented matrix, where we put the righthand side into the matrix separated by a vertical line:

$$\left(\begin{array}{ccc|c} 3 & 2 & 1 & 6 \\ -1 & 4 & 5 & 8 \\ 2 & -8 & 10 & 4 \end{array}\right).$$

We will store this matrix in Python for use later:

```
In [1]: import numpy as np
        aug_matrix = np.matrix([(3.0,2,1,6),(-1,4,5,8),(2,-8,10,4)])
        print(aug_matrix)

[[  3.   2.   1.   6.]
 [ -1.   4.   5.   8.]
 [  2.  -8.  10.   4.]]
```

A straightforward way to solve this system is to use the tools of elementary algebra and try to eliminate variables by adding and subtracting equations from each other. We could do this by taking the second row, and adding to it 1/3 times the first row. Here's how that's done in Python:

```
In [2]: #add row 2 to 1/3 times row 1
        row13 = aug_matrix[0]/3 #row1 * 1/3
        new_row2 = aug_matrix[1] + row13 #add 1/3 row 1 to row 2
        #replace row 2
        aug_matrix[1,:] = new_row2
        print("New matrix =\n",aug_matrix)

New matrix =
 [[  3.           2.           1.           6.        ]
 [  0.           4.66666667   5.33333333  10.        ]
 [  2.          -8.          10.           4.        ]]
```

This eliminated x_1 from the second equation. The next step would be to eliminate x_1 from the third equation by adding $-2/3$ times row 1 to row 3:

```
In [3]: #add row 3 to -2/3 times row 1
        row23 = -2*aug_matrix[0]/3 #row1 * -2/3
        new_row3 = aug_matrix[2] + row23 #add -2/3 row 1 to row 3
        #replace row 3
        aug_matrix[2] = new_row3
        print("New matrix =\n",aug_matrix)

New matrix =
 [[  3.           2.           1.           6.        ]
 [  0.           4.66666667   5.33333333  10.        ]
 [  0.          -9.33333333   9.33333333   0.        ]]
```

The x_2 term from the third equation can be removed by adding to row 3 the quantity $9\frac{1}{3}/4\frac{2}{3}$ times row 2:

```
In [4]: #add row 3 to 9.33333/4.66666 times row 2
        modrow = (9+1./3)/(4+2./3)*aug_matrix[1] #row2 * (-9+1./3)/(4+2.0/3)
        new_row3 = aug_matrix[2] + modrow #add -2/3 row 1 to row 3
        #replace row 3
        aug_matrix[2] = new_row3.copy()
        print("New matrix =\n",aug_matrix)

New matrix =
 [[  3.           2.           1.           6.        ]
 [  0.           4.66666667   5.33333333  10.        ]
 [  0.           0.          20.          20.        ]]
```

Notice that we have manipulated our original system into the equivalent system

$$\left(\begin{array}{ccc|c} 3 & 2 & 1 & 6 \\ 0 & 4\frac{2}{3} & 5\frac{1}{3} & 10 \\ 0 & 0 & 20 & 20 \end{array}\right),$$

or

$$\begin{pmatrix} 3 & 2 & 1 \\ 0 & 4\frac{2}{3} & 5\frac{1}{3} \\ 0 & 0 & 20 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 10 \\ 20 \end{pmatrix}.$$

We can easily solve this via system using a process called "back substitution". In back substitution we start at the last row and solve each equation in succession. In this particular example, take the last equation and solve for x_3, then plug the value of x_3 into the second equation, solve for x_2, and then plug both into the first equation and solve for x_1.

```
In [5]: #backsubstitution
        x3 = aug_matrix[2,3]/aug_matrix[2,2] #solve for x3
        print("x3 =",x3)
        #now solve for x2
        x2 = (aug_matrix[1,3] - x3*aug_matrix[1,2])/aug_matrix[1,1]
```

```
        print("x2 =",x2)
        #now solve for x1
        x1 = (aug_matrix[0,3] - x3*aug_matrix[0,2]-
              x2*aug_matrix[0,1])/aug_matrix[0,0]
        print("x1 =",x1)

x3 = 1.0
x2 = 1.0
x1 = 1.0
```

Therefore, the solution we get is

$$x_1 = 1, \qquad x_2 = 1, \qquad x_3 = 1.$$

We can check this solution by multiplying it by the original coefficient matrix, and showing that the result is equal to the system's righthand side:

```
In [6]: A = np.matrix([(3.0,2,1),(-1,4,5),(2,-8,10)])
        x = np.array([1,1,1])
        b = np.array([6,8,4])
        print(np.dot(A,x),"-",b,"=",np.dot(A,x)-b)

[[ 6.  8.  4.]] - [6 8 4] = [[ 0.  0.  0.]]
```

Our solution does indeed satisfy the original system. The method we used to solve this system goes by the name Gaussian elimination. The basic idea is to march through the system and eliminate variables one by one until the final equation of the system has only a single unknown. Then back substitution is used to solve for each variable, starting with the last one.

The formula for back-substitution can be written succinctly by noticing that each x_i is the sum of solutions of the unknowns further down the vector divided by the diagonal element of the modified matrix:

$$x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{j=i+1}^{3} A_{ij} x_i \right), \qquad i = 1, 2, 3 \tag{7.2}$$

with **A** being the matrix after applying Gaussian elimination.

7.2 A FUNCTION FOR SOLVING 3 × 3 SYSTEMS

The procedure we used to solve the system given in Eq. (7.1) can be generalized to any coefficients and righthand side. We could just copy and paste the code above, and then change the numbers if we wanted to solve another system, but that would be prone to errors. In the next code block, we define a function that mimics our algorithm above to change the augmented matrix into a form ready for back substitution. The function is then tested on the system in Eq. (7.1).

```
In [7]: def GaussElim33(A,b):
            """create a Gaussian elimination matrix for a 3x3 system

            Args:
                A: 3 by 3 array
                b: array of length 3
            Returns:
                augmented matrix ready for back substitution
            """
            #create augmented matrix
            aug_matrix = np.zeros((3,4))
            aug_matrix[0:3,0:3] = A
            aug_matrix[:,3] = b
            #augmented matrix is created
            for column in range(0,3):
                for row in range(column+1,3):
                    mod_row = aug_matrix[row,:]
                    mod_row -= (mod_row[column]/aug_matrix[column,column]*
                                aug_matrix[column,:])
                    aug_matrix[row] = mod_row
            return aug_matrix
        #test function on the problem above
        aug = GaussElim33(A,b)
        print(aug)
```

```
[[  3.           2.           1.           6.        ]
 [  0.           4.66666667   5.33333333  10.        ]
 [  0.           0.          20.          20.        ]]
```

As we can see, this gives us the same result we obtained from our hand calculation. We should test this function more thoroughly, but we will return to that later. The next step is to write a back substitution function that implements Eq. (7.4)

```
In [8]: def BackSub33(aug_matrix,x):
            """back substitute a 3 by 3 system after Gaussian elimination

            Args:
                aug_matrix: augmented matrix with zeros below the diagonal
                x: length 3 vector to hold solution
            Returns:
                nothing
            Side Effect:
                x now contains solution
            """
            #start at the end
            for row in [2,1,0]:
                RHS = aug_matrix[row,3]
                for column in range(row+1,3):
                    RHS -= x[column]*aug_matrix[row,column]
                x[row] = RHS/aug_matrix[row,row]
            return
        x = np.zeros(3)
        BackSub33(aug,x)
        print("The solution is ", x)
```

```
The solution is  [ 1.  1.  1.]
```

The back substitution function has the first example of using a side effect to change an input parameter that we have seen up to this point. To illustrate why we do this, we will revisit the topic of mutable types being passed to a function. Recall, the way that Python and NumPy work: when a NumPy array is passed to a function, the function does not copy the array into local scope. Rather, the function can modify the original values of the array. This is accomplished through a construct called pass by reference. What this means is that the function gets a reference to the memory where the original array lives, and can therefore modify that memory. The function does not make a copy of the NumPy array because the array could easily have millions of elements, and it is a bad idea to be heedlessly copying these large data structures because the computer could easily run out of memory. The upshot of pass by reference in the code above is that when we pass `x` into the function `BackSub33`, the function puts the solution into `x`. This changing of memory outside the function is a side effect because it is a way that a function interacts with the rest of the code not through the mechanism of returning information from a function.

NumPy arrays are not the only data structures that are passed by reference to a function. Any data type that can have its size modified is called a mutable type, and these are passed by reference to functions. We have already encountered two other mutable types in python: the list and dictionary. This means if you pass either of these two types to a function, that function can change the list or dictionary. It is good programming practice to make explicit in the comments and docstring what the side-effects are of a function so that the user knows when a function is called the original data may be changed.

BOX 7.1 PYTHON PRINCIPLE

Mutable data types such as lists, dictionaries, and NumPy arrays, are passed by reference to functions. This means that a function can change the values of a mutable data type when it is the parameter of the function. When a function changes the input data, this is a side effect, and it should be noted in the function's docstring.

We need more vigorous testing of our function above to convince ourselves that it is a tool for solving a general 3×3 system. To make this a general test, we will select a random matrix (using `np.random.random`) and then multiply it by a random vector to get the righthand side. Then when we solve the system the solution should be the original vector. It would be best to do this many times using a different matrix each time. The code below performs 100 such tests and asserts that the maximum absolute difference between the computed solution and the true solution should be less than 10^{-12}. An `assert` checks this, and error handling is used to give information about a failed test.

```
In [9]: tests = 100
        for i in range(tests):
            x = np.random.rand(3)
            A = np.random.rand(3,3)
            b = np.dot(A,x)
```

```
            aug = GaussElim33(A,b)
            sol = np.zeros(3)
            BackSub33(aug,sol)
            diff = np.abs(sol-x)
            try:
                assert(np.max(diff) < 1.0e-12)
            except AssertionError:
                print("Test failed with")
                print("A = ",A)
                print("x = ",x)
                raise
        print("All Tests Passed!")
```

```
All Tests Passed!
```

These tests pass, and we have some confidence that our algorithm is working. Later we will see how it might be improved to tackle some pathological cases.

7.3 GAUSSIAN ELIMINATION FOR A GENERAL SYSTEM

The code we wrote above only solved 3×3 systems, that is a useful tool, but we would not want to write a different function for every size system we want to solve. That is, we do not want to have to write a 4×4 solver and a 5×5 solver, etc. Therefore, we need to have a general Gaussian elimination code to solve the system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{7.3}$$

where $\mathbf{x}$ and $\mathbf{b}$ are vectors of length N, and $\mathbf{A}$ is an $N \times N$ matrix.

Due to the way we structured our 3×3 code, the main change we need to make to adapt the function systems of size N is in the `for` loops. On the whole the code is unchanged: all we must do is replace the 3's in the function with N's. The next code block has this function, and a 4×4 test that has solution $\mathbf{x} = (1, 2, 3, 4)$.

```
In [10]: def GaussElim(A,b):
             """create a Gaussian elimination matrix for a system

             Args:
                 A: N by N array
                 b: array of length N
             Returns:
                 augmented matrix ready for back substitution
             """
             [Nrow, Ncol] = A.shape
             assert Nrow == Ncol
             N = Nrow
             assert b.size == N
             #create augmented matrix
             aug_matrix = np.zeros((N,N+1))
             aug_matrix[0:N,0:N] = A
```

```
            aug_matrix[:,N] = b
            #augmented matrix is created
            for column in range(0,N):
                for row in range(column+1,N):
                    mod_row = aug_matrix[row,:]
                    mod_row -= (mod_row[column]/aug_matrix[column,column]*
                                aug_matrix[column,:])
                    aug_matrix[row] = mod_row
            return aug_matrix

        #let's try it on a 4 x 4 to start
        A = np.array([(3.0,2,1,1),(-1,4,5,-2),(2,-8,10,-3),(2,3,4,5)])
        answer = np.arange(4)+1.0 #1,2,3,4
        b = np.dot(A,answer)
        aug = GaussElim(A,b)
        print(aug)
```

```
[[  3.          2.          1.          1.         14.        ]
 [  0.          4.66666667  5.33333333 -1.66666667 18.66666667]
 [  0.          0.         20.         -7.         32.        ]
 [  0.          0.          0.          5.42857143 21.71428571]]
```

Notice that we had to determine the size of the system by looking at the size of the matrix and the vector. Also, using `assert` statements, the function checks that the sizes of the input data are compatible. The next step is to define the back substitution function. This function will generalize Eq. (7.4) to have the summation going to N rather than 3, and letting i range from 1 to N:

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{j=i+1}^{N} A_{ij}x_i\right), \qquad i = 1, 2, 3. \tag{7.4}$$

The resulting code is

```
In [11]: def BackSub(aug_matrix,x):
             """back substitute a N by N system after Gauss elimination

             Args:
                 aug_matrix: augmented matrix with zeros below the diagonal
                 x: length N vector to hold solution
             Returns:
                 nothing
             Side Effect:
                 x now contains solution
             """
             N = x.size
             [Nrow, Ncol] = aug_matrix.shape
             assert Nrow + 1 == Ncol
             assert N == Nrow
             for row in range(N-1,-1,-1):
                 RHS = aug_matrix[row,N]
                 for column in range(row+1,N):
                     RHS -= x[column]*aug_matrix[row,column]
```

```
            x[row] = RHS/aug_matrix[row,row]
        return

    x = np.zeros(4)
    BackSub(aug,x)
    print("The solution is ", x)
```

```
The solution is  [ 1.  2.  3.  4.]
```

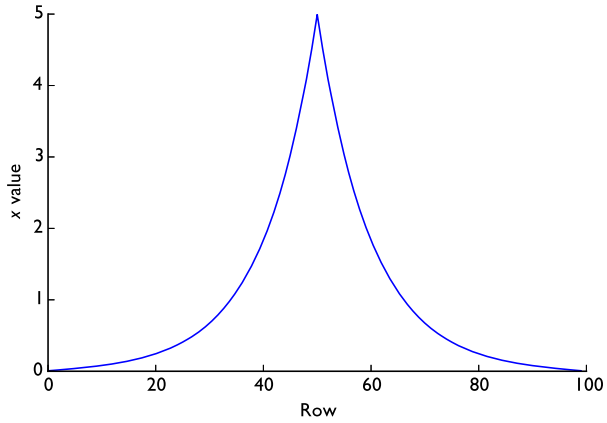
Applying the back substitution function gives us the expected solution, $\mathbf{x} = (1, 2, 3, 4)$.

Now that we have a generic solver, we can make a large matrix that has 2.01 on the diagonal and -1 on the immediate off diagonals, and create a simple righthand side. This code is an example of automatically filling in a matrix. This is an important technique to master because there are many different methods available for solving linear systems, yet filling the matrix can be the hardest part of solving the problem. In the code below we fill the diagonal by setting up a range that runs from 0 to the number of rows in the matrix minus one. We then fill the off diagonals using similar ranges. This type of matrix is called a tridiagonal matrix because it has entries in three diagonal lines in the matrix. We will see these matrices again when we discretize the neutron diffusion equation.

```
In [12]: mat_size = 100
         A = np.zeros((mat_size,mat_size))
         b = np.zeros(mat_size)
         diag = np.arange(mat_size)
         A[diag,diag] = 2.01
         belowDiagRow = np.arange(1,mat_size)
         A[belowDiagRow,belowDiagRow-1] = -1
         aboveDiagRow = np.arange(mat_size-1)
         A[aboveDiagRow,aboveDiagRow+1] = -1
         print(A)

         b[np.floor(mat_size/2)] = 1
         aug_mat = GaussElim(A,b)
         x = np.zeros(mat_size)
         BackSub(aug_mat,x)
         plt.plot(diag,x,color="blue")
         plt.xlabel("Row");
         plt.ylabel("x value");
         plt.show();
```

```
[[ 2.01 -1.    0.   ...,  0.    0.    0.  ]
 [-1.    2.01 -1.   ...,  0.    0.    0.  ]
 [ 0.   -1.    2.01 ...,  0.    0.    0.  ]
 ...,
 [ 0.    0.    0.   ...,  2.01 -1.    0.  ]
 [ 0.    0.    0.   ..., -1.    2.01 -1.  ]
 [ 0.    0.    0.   ...,  0.   -1.    2.01]]
```



As we will see later, this is related to the solution of the diffusion equation with a Dirac delta function source. The delta function source was specified by having the vector `b` be zero everywhere except for a single row where it was set to one.

7.4 ROUND OFF AND PIVOTING

For matrices with a large discrepancy in the magnitude of the elements, Gaussian elimination does not perform quite as well. This is due to the fact that the operations used in Gaussian elimination are done with finite precision arithmetic. When large numbers are divided by small numbers, the numerical round off error can build and lead to incorrect answers at the end of Gaussian elimination. An example matrix to demonstrate this is given below.

```
In [13]: epsilon = 1e-14
         A = np.array([(epsilon,-1,1),(-1,2,-1),(2,-1,0)])
         print(A)

[[  1.00000000e-14  -1.00000000e+00   1.00000000e+00]
 [ -1.00000000e+00   2.00000000e+00  -1.00000000e+00]
 [  2.00000000e+00  -1.00000000e+00   0.00000000e+00]]
```

We will set the righthand side of our system to a simple vector:

```
In [14]: b = np.array([0,0,1.0])
         print(b)

[ 0.  0.  1.]
```

Now we solve this system:

```
In [15]: aug_mat = GaussElim(A,b)
         x = np.zeros(3)
         BackSub(aug_mat,x)
         print("The solution is",x)
         print("The residual is",b-np.dot(A,x))
```

```
The solution is [ 0.96589403  0.96969697  0.96969697]
The residual is [ 0.         -0.00380294  0.03790891]
```

As you can see there is a noticeable difference between the calculated solution and the actual solution (as measured by the residual). The residual is the difference between the vector `b` and the matrix `A` times the solution. If the solution were exact, the residual would be zero. Part of the reason for this is that we have not used the largest row element as the *pivot* element, or the element we divide by when doing the elimination. In this example, the pivot element is near zero and finite precision error is magnified when we divide by this small number.

To correct this issue we could rearrange the rows so that we divide by larger elements. This is called *row pivoting*.

BOX 7.2 NUMERICAL PRINCIPLE

The error in an approximation is the difference between the true solution and the approximate solution, i.e., for an approximate solution $\hat{\mathbf{x}}$ to the vector $\mathbf{x}$ the error is defined as

$$\mathbf{e} = \mathbf{x} - \hat{\mathbf{x}}.$$

The residual is equal to the discrepancy between the righthand and lefthand sides of the original equations when the approximate solution is substituted. For a linear system of equations given by $\mathbf{Ax} = \mathbf{b}$, the residual, $\mathbf{r}$, for an approximation $\hat{\mathbf{x}}$ is

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}.$$

If one does not know the true solution, the residual can give a measure of the error because the exact solution has a residual of zero. Also, the residual and the error are related via the equation,

$$\mathbf{Ae} = \mathbf{r}.$$

To further demonstrate the need for row pivoting, consider the system:

```
In [16]: A = np.array([(1.0,0),(0,1.0)])
         b = np.array([2,6.0])
         print("A=\n",A)
         print("b=",b)

A=
 [[ 1.  0.]
 [ 0.  1.]]
b= [ 2.  6.]
```

The solution to this system is

```
In [17]: aug_mat = GaussElim(A,b)
         x = np.zeros(2)
         BackSub(aug_mat,x)
         print(x)

[ 2.  6.]
```

This is as expected because **A** is an identity matrix. Now what if we switch the rows in **A** and try the same thing:

```
In [18]: A = np.array([(0,1.0),(1.0,0)])
         b = np.array([6.,2.0])
         print("A =\n",A)
         print("b =",b)
         aug_mat = GaussElim(A,b)
         x = np.zeros(2)
         BackSub(aug_mat,x)
         print("x =",x)

A =
 [[ 0.  1.]
 [ 1.  0.]]
b = [ 6.  2.]
x = [ nan  nan]

-c:21: RuntimeWarning: divide by zero encountered in double_scalars
-c:21: RuntimeWarning: invalid value encountered in multiply
-c:17: RuntimeWarning: invalid value encountered in double_scalars
```

We do not get a solution because the diagonal element in the first row was zero. Therefore, we divided by zero during the Gaussian elimination process.

We can correct both of these issues by checking before we eliminate and rearranging the equations so that we divide by the largest possible entry. This is just rearranging the order that we solve the equations, and will just involve switching rows in the augmented matrix.

It is worth mentioning that largest element is relative to the size of the other entries in the row. If one row is (10, 11, 10) and another is (2, 1, 1) the 2 is actually a better pivot because it is the largest element in its row.

To do pivoting we will need a means to swap rows in our augmented matrix. The following function will modify the augmented matrix by swapping rows.

```
In [19]: def swap_rows(A, a, b):
             """Swap two rows in a matrix: switch row a with row b

             args:
             A: matrix to perform row swaps on
             a: row index of matrix
             b: row index of matrix

             returns: nothing

             side effects:
             changes A to have rows a and b swapped
             """
             assert (a>=0) and (b>=0)
             N = A.shape[0] #number of rows
             assert (a<N) and (b<N) #less than because 0-based indexing
             temp = A[a,:].copy()
             A[a,:] = A[b,:].copy()
             A[b,:] = temp.copy()
```

```
         print("Before swap, A =\n",A)
         swap_rows(A,0,1)
         print("After swapping 0 and 1, A =\n",A)

Before swap, A =
 [[ 0.  1.]
 [ 1.  0.]]
After swapping 0 and 1, A =
 [[ 1.  0.]
 [ 0.  1.]]
```

The next step is figuring out how to swap during each step. At the beginning of the solve we want find the maximum row element magnitude for each row and store it as a vector. The following code is an example of this:

```
In [20]: N = 5
         A = np.random.rand(N,N)
         print("A =\n",A)
         s = np.zeros(N)
         count = 0
         for row in A:
             s[count] = np.max(np.fabs(row))
             count += 1
         print("s =",s)

A =
 [[ 0.52115101  0.46323356  0.68539875  0.56665694  0.1093434 ]
 [ 0.00451899  0.29089231  0.9037581   0.83739104  0.90189019]
 [ 0.15378442  0.77964023  0.51662888  0.03317186  0.35942844]
 [ 0.24128529  0.03615319  0.43274014  0.68011597  0.42847979]
 [ 0.96757013  0.94404531  0.56392054  0.32454444  0.97512228]]
s = [ 0.68539875  0.9037581   0.77964023  0.68011597  0.97512228]
```

Then we will have to figure out which row has the largest scaled element in the pivot position and call the `swap_rows` function. We will use the `argmax` function which returns the index of the largest element in a vector.

```
In [21]: pivot_column = 2
         largest_pos = np.argmax(np.fabs(A[:,pivot_column]/s))
         print("Largest scaled element in column",
               pivot_column,"is in row",largest_pos)

Largest scaled element in column 2 is in row 1
```

Now can put this all together in a new version of Gaussian elimination. This function will find the largest elements in each row of the matrix, then proceed with Gaussian elimination where the pivot element is the element that is the largest element in remaining in the row.

```
In [22]: def GaussElimPivotSolve(A,b,LOUD=0):
             """create a Gaussian elimination with pivoting matrix for a system
```

```
    Args:
        A: N by N array
        b: array of length N
    Returns:
        solution vector in the original order
    """
    [Nrow, Ncol] = A.shape
    assert Nrow == Ncol
    N = Nrow
    #create augmented matrix
    aug_matrix = np.zeros((N,N+1))
    aug_matrix[0:N,0:N] = A
    aug_matrix[:,N] = b
    #augmented matrix is created

    #create scale factors
    s = np.zeros(N)
    count = 0
    for row in aug_matrix[:,0:N]: #don't include b
        s[count] = np.max(np.fabs(row))
        count += 1
    if LOUD:
        print("s =",s)
    if LOUD:
        print("Original Augmented Matrix is\n",aug_matrix)
    #perform elimination
    for column in range(0,N):

        #swap rows if needed
        largest_pos =(np.argmax(np.fabs(aug_matrix[column:N,column:N]/
                        s[column]))  + column)
        if (largest_pos != column):
            if (LOUD):
                print("Swapping row",column,"with row",largest_pos)
                print("Pre swap\n",aug_matrix)
            swap_rows(aug_matrix,column,largest_pos)
            #re-order s
            tmp = s[column]
            s[column] = s[largest_pos]
            s[largest_pos] = tmp
            if (LOUD):
                print("A =\n",aug_matrix)
        #finish off the row
        for row in range(column+1,N):
            mod_row = aug_matrix[row,:]
            mod_row -= (mod_row[column]/
                        aug_matrix[column,column]*aug_matrix[column,:])
            aug_matrix[row] = mod_row
    #now back solve
    x = b.copy()
    if LOUD:
        print("Final aug_matrix is\n",aug_matrix)
    BackSub(aug_matrix,x)
    return x
```

As a basic test we will solve a 3×3 system:

```
In [23]: #let's try it on a 3 x 3 to start
         A = np.array([(3.0,2,1),(-1,4,5),(2,-8,10)])
         answer = np.arange(3)+1.0 #1,2,3
         b = np.dot(A,answer)
         x = GaussElimPivotSolve(A,b,LOUD=True)
         print("The solution is",x)
         print("The residual (errors) are",np.dot(A,x)-b)

s = [  3.   5.  10.]
Original Augmented Matrix is
 [[  3.   2.   1.  10.]
 [ -1.   4.   5.  22.]
 [  2.  -8.  10.  16.]]
Swapping row 1 with row 2
Pre swap
 [[  3.           2.           1.          10.        ]
 [  0.           4.66666667   5.33333333  25.33333333]
 [  0.          -9.33333333   9.33333333   9.33333333]]
A =
 [[  3.           2.           1.          10.        ]
 [  0.          -9.33333333   9.33333333   9.33333333]
 [  0.           4.66666667   5.33333333  25.33333333]]
Final aug_matrix is
 [[  3.           2.           1.          10.        ]
 [  0.          -9.33333333   9.33333333   9.33333333]
 [  0.           0.          10.          30.        ]]
The solution is [ 1.  2.  3.]
The residual (errors) are [  0.00000000e+00   0.00000000e+00   3.55271368e-15]
```

Based on the residual, and the fact the we engineered the answer to be known, we can see that the solution is correct. To see the efficacy of the row pivoting we will try our function on the systems that did not work so well before. First we try the rearranged identity matrix:

```
In [24]: A = np.array([(0,1.0),(1.0,0)])
         b = np.array([6.,2.0])
         print("A =\n",A)
         print("b =",b)
         x = GaussElimPivotSolve(A,b,LOUD=False)
         print("x =",x)
         print("The residual (errors) are",np.dot(A,x)-b)

x = [ 2.  6.]
The residual (errors) are [ 0.  0.]
```

The answer is correct. Finally, we try the matrix that had a large discrepancy between the element sizes:

```
In [25]: epsilon = 1e-14
         A = np.array([(epsilon,-1,1),(-1,2,-1),(2,-1,0)])
         print(A)          b = np.array([0,0,1.0])
         x = GaussElimPivotSolve(A,b,LOUD=False)
         print("x =",x)
         print("The residual is",np.dot(A,x)-b)
```

```
x = [ 1.  1.  1.]
The residual is [  0.00000000e+00  -2.22044605e-16   0.00000000e+00]
```

Now we get the correct answer and the residual is effectively zero.

7.5 TIME TO SOLUTION FOR GAUSSIAN ELIMINATION

We will now discuss how long we should expect Gaussian elimination to take. We will do this through counting the number of floating point operations required to perform Gaussian elimination on an $n \times n$ matrix. A floating point operation is either addition, subtraction, multiplication, or division of floating point numbers. The counting of these operations for Gaussian elimination is straightforward, but tedious. In our derivation of the operation count we will ignore the extra column we added to get the augmented matrix because we care about how the operation count scales for large matrices. If the matrix is large, the addition of a single column is a small perturbation.

At the start of the algorithm we have to eliminate the first column from the $(n-1)$ rows that are not the first row. We first need to compute the elimination factors which involve dividing the row 1, column 1 element by the appropriate pivot element, requiring $(n-1)$ divisions because this must be done for each row that is not the first. Next, we multiply each row by the elimination factor. This involves $(n-1)^2$ multiplications because we multiply every element of the matrix, besides the first row and the first column, by the appropriate factor to eliminate the first column below the first row. Then we need to add $(n-1)$ elements from row 1, columns 2 through n to the corresponding elements in all the other rows: this is $(n-1)^2$ additions in all. Therefore after eliminating the first column we have performed $2(n-1)^2 + (n-1)$ floating point operations.

After eliminating the first column below the diagonal, we now have an $(n-1)$ by $(n-1)$ matrix that we need to eliminate a column from. This will require $2(n-2)^2 + (n-2)$ floating point operations. The column after that requires $2(n-3)^2 + (n-3)$, and so on until we get to the last column. Using summation notation we can write the total number of floating point operations as

$$\text{Total Floating Point Operations} = \sum_{l=1}^{n} \Big[2(n-l)^2 + (n-l) \Big].$$

We can factor this expression to get a simpler form:

$$\begin{aligned}
\sum_{\ell=1}^{n} \Big[2(n-\ell)^2 + (n-\ell) \Big] &= \sum_{\ell=1}^{n} \Big[2n^2 - 4n\ell + 2\ell^2 + n - \ell \Big] \\
&= 2n^3 + n^2 - \sum_{\ell=1}^{n} \Big[4n\ell - 2\ell^2 + \ell \Big] \\
&= 2n^3 + n^2 - 2n^2(n+1) - \frac{n(n+1)}{2} + \frac{n(n+1)(2n+1)}{3}
\end{aligned}$$

$$= \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n,$$

where we have used the identities

$$\sum_{\ell=1}^{n} \ell = \frac{n(n+1)}{2}, \qquad \sum_{\ell=1}^{n} \ell^2 = \frac{n(n+1)(2n+1)}{6}.$$

Therefore, the total number of floating point operations in Gaussian elimination for a matrix of size n by n is

$$\text{Total Floating Point Operations} = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n.$$

Often we are concerned with how the number of floating point operations scales as n increases. Therefore, we want to know how Gaussian elimination trends as n is large. For large n, $n^3 \gg n^2 \gg n$. For this reason we say that Gaussian elimination is an $O(n^3)$, or order n cubed, algorithm because the leading term when n is large is the n^3 term. What this means in practice is that if we double the number of rows in my matrix, i.e., n goes to $2n$, we should expect the code to take $2^3 = 8$ times longer. Nevertheless, this is only for large n as will see next.

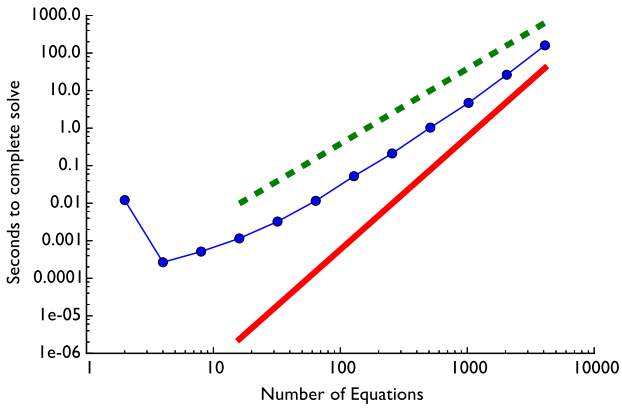
BOX 7.3 NUMERICAL PRINCIPLE

When talking about the number of operations for an algorithm, we often use Big-O notation to describe how the leading order term behaves as the problem gets bigger. The leading-order growth of Gaussian elimination scales as the number of equations, n, to the third power. Therefore, we say that this algorithm is $O(n^3)$. As an example, an algorithm that increased linearly in the problem size would be an $O(n)$ algorithm.

We can use the time module that Python provides to time how long it takes to run our Gaussian elimination code as we increase n.

```
In [26]: import time
         num_tests = 13
         N = 2**np.arange(num_tests)
         times = np.zeros(num_tests)
         for test in range(1,num_tests):
             A = np.random.rand(N[test],N[test])
             b = np.dot(A,np.ones(N[test]))
             x = np.zeros(N[test])
             start = time.clock()
             aug = GaussElim(A,b)
             BackSub(aug,x)
             end = time.clock()
             times[test] = end-start
         plt.loglog(N,times,'o-')
         y_comp = times.copy()/4
```

```
for comp_place in range(num_tests-1,0,-1):
    #because x goes up by factor 2 each time, time should go up by 8
    y_comp[comp_place -1] = np.exp(np.log(y_comp[test])-3*(
                            np.log(N[test])-np.log(N[comp_place-1])))
plt.loglog(N[4:num_tests],y_comp[4:num_tests],'r',linewidth=4)
#make a comparison line with slope 2
y_comp = times.copy()*4
for comp_place in range(num_tests-1,0,-1):
    y_comp[comp_place -1] = np.exp(np.log(y_comp[test])-2*(
                            np.log(N[test])-np.log(N[comp_place-1])))
plt.loglog(N[4:num_tests],y_comp[4:num_tests],'g-',linewidth=4)
plt.xlabel("Number of Equations")
plt.ylabel("Seconds to complete solve")
plt.show()
```



In this figure the solid line is a slope of 3 on a log-log scale (which corresponds time growing as the number of equations to the third power). If the time to solve a system was consistent with the theory, then this dots would be parallel to the red line. That assumes that the time it takes to execute the code is linearly related to the number of floating point operations. Starting around tens of equations, the slope is closer to 2, the slope of the dashed line. This implies that initially the time to solution is not growing as fast as the theory. The reason for this is that Python is not giving the CPU work to do fast enough so that the floating point operations are not the bottleneck to the calculation, rather, feeding instructions to the CPU is taking up the bulk of the time. As the matrices get bigger, each instruction takes longer and the overhead of feeding instructions is a smaller fraction of the execution time. We see this in the figure: as n grows, the slope is getting steeper: when we have thousands of equations the slope is around 2.6. This indicates that if we ran a large enough problem, and had enough patience and computer memory, we would see the predicted result.

FURTHER READING

Gaussian elimination is a classical method for solving linear systems, and there are many possible references for it. The book on numerical linear algebra by Trefethen and Bau [9] provides an in-depth discussion of the stability of Gaussian elimination with row pivoting.

PROBLEMS

Short Exercises

7.1. Solve the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ where

$$\mathbf{A} = \begin{pmatrix} -1 & 2 & 3 \\ 4 & -5 & 6 \\ 7 & 8 & -9 \end{pmatrix} \qquad \mathbf{b} = (12.9, -5.1, 10.7)^{\mathrm{T}}.$$

7.2. Consider the n by n matrix defined by

$$\mathbf{A} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \vdots & & & & \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{pmatrix},$$

and the vector

$$\mathbf{b} = (1, -1, 1, -1, \ldots)^{\mathrm{T}}.$$

Write a function to generate $\mathbf{A}$ and $\mathbf{b}$ as a function of n. Use Gaussian elimination to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $n = 2, 4, 8, 16$. What do you notice about your answers? Are the answers that you get from Gaussian elimination correct?

Programming Projects

1. Xenon Poisoning

^{135}Xe is produced in a nuclear reactor through two mechanisms:

- The production of ^{135}Te from fission, which then decays to ^{135}I via β decay with a half-life of 19 s, which then decays to ^{135}Xe via β decay with a half-life of 6.6 h.
- The production of ^{135}Xe directly from fission.

^{135}Xe is important because it has a very large capture cross-section of 2.6×10^6 barns. ^{135}Xe is also radioactive, and decays with a half-life of 9.1 h.

For a reactor with an average scalar flux, ϕ in units [neutrons/cm^2/s], the equations for the production of ^{135}Xe are (using the abbreviation number density [nuclei/cm^3] of ^{135}Xe$\rightarrow X$, ^{135}I$\rightarrow I$, ^{135}Te$\rightarrow T$),

$$\frac{d}{dt}T = \gamma_T \Sigma_f \phi - \lambda_T T,$$
$$\frac{d}{dt}I = \lambda_T T - \lambda_I I,$$
$$\frac{d}{dt}X = \lambda_I I + \gamma_X \Sigma_f \phi - (\lambda_X + \sigma_a^X \phi) X.$$

If we consider the steady-state limit, i.e., $\frac{d}{dt} = 0$, we have an equation for the equilibrium concentration of all three nuclides. For a ^{235}U fueled reactor, the fission yields for Te and Xe are $\gamma_T = 0.061$, $\gamma_X = 0.003$, and the macroscopic fission cross-section is $\Sigma_f = 0.07136\ \text{cm}^{-1}$. Compute the equilibrium concentrations of these three nuclides at power densities of 5, 50, and 100 W/cm^3, using the energy per fission of 200 MeV/fission. Also, compute the absorption rate density of neutrons in ^{135}Xe, given by $\sigma_a^X \phi X$.

2. *Flux Capacitor Waste*

In the movie *Back to the Future*, the time machine is fueled by plutonium. Assuming that the time machine is refueled with 1 mg of pure ^{239}Pu every year and the old fuel, still with the same mass, is placed in a disposal site. The stable lead is recovered from the site at a rate of 50% a year. What are the equilibrium mass of ^{239}Pu and its daughter products after this refueling cycle goes on for a long time. Also, compute the alpha particle and beta particle production rate. The decay chain for ^{239}Pu is given on page 92.

3. *Four-Group Reactor Theory*

A large subcritical system with a source has its scalar flux in each of four groups described by these four equations,

$$\Sigma_{a,1}\phi_1 + \Sigma_{s,1\to 2}\phi_1 = \chi_1 \sum_{g=1}^{4} \nu\Sigma_{f,g}\phi_g + Q_1$$
$$\Sigma_{a,2}\phi_2 + \Sigma_{s,2\to 3}\phi_2 = \chi_2 \sum_{g=1}^{4} \nu\Sigma_{f,g}\phi_g + \Sigma_{s,1\to 2}\phi_1 + Q_2$$
$$\Sigma_{a,3}\phi_3 + \Sigma_{s,3\to 4}\phi_3 = \chi_3 \sum_{g=1}^{4} \nu\Sigma_{f,g}\phi_g + \Sigma_{s,2\to 3}\phi_2 + Q_3$$
$$\Sigma_{a,4}\phi_4 = \chi_4 \sum_{g=1}^{4} \Sigma_{f,g}\phi_g + \Sigma_{s,3\to 4}\phi_3 + Q_4.$$

For source strength, $Q = [10^{12}, 0, 0, 0]$ n/cm^3/s and the cross-sections given below in Table 7.1 [10] for the group bounds, [1.35 MeV, 10 MeV], [9.1 keV, 1.35 MeV], [0.4 eV, 9.1 keV], and [0.0 eV, 0.4 eV], compute the scalar flux in each group in system where neutrons can only scatter within group or to the energy group below.

TABLE 7.1 Four Group Constants

	Group 1	Group 2	Group 3	Group 4
χ_g	0.57500	0.42500	0.00000	0.00000
$\nu\Sigma_{fg}$ (cm^{-1})	0.00480	0.00060	0.00885	0.09255
Σ_{ag} (cm^{-1})	0.00490	0.00280	0.03050	0.12100
$\Sigma_{sg\to g+1}$ (cm^{-1})	0.08310	0.05850	0.06510	
D_g (cm)	2.16200	1.08700	0.63200	0.35400

4. Matrix Inverse

The inverse of a matrix **A** is defined as the matrix such that

$$\mathbf{A}\mathbf{A}^{-1} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Given this fact, we can compute the inverse of an n by n matrix, column by column, by solving the following set of problems:

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i \qquad i = 1, \ldots, n,$$

where $\mathbf{b}_i$ is a vector of zeros except at position i where it has 1, and $\mathbf{x}_i$ is the ith column of the inverse. Using this formulation write a Python function that computes the inverse of a general $n \times n$ matrix. There are two ways of doing this, one way computes Gaussian elimination n times, the other more sophisticated way has the augmented matrix have n extra columns corresponding to all the $\mathbf{b}_i$. Test your function on the matrix from Short Exercise 7.1 of this chapter and demonstrate that the inverse matrix you compute gives the identity when multiplied by **A**.