CHAPTER

# 9

# Iterative Methods for Linear Systems

OUTLINE



*These evils thou repeat'st upon thyself*
*Have banish'd me from Scotland. O my breast,*
*Thy hope ends here!*

***–"MacDuff" in the play* Macbeth *by William Shakespeare***

## CHAPTER POINTS

- Iterative methods are an alternative to direct methods such as Gaussian Elimination and LU factorization.
- These methods can be faster than direct methods.
- The number of iterations required to compute a solution may be less important than how long it takes each iteration to complete.

The methods we have talked about up to this point are called "direct" methods because they pass through the matrix a fixed number of times to get the answer. For example, Gaussian

elimination makes one pass through the augmented matrix to get it in upper triangular form, and then back solves to get the answers. You can say ahead of time exactly how many operations it will take to solve the system (modulo any pivoting). One thing that is also true is that no matter the system, the amount of work is the same if the number of equations is the same.

If we have a large system, it may be possible to solve the linear system in a faster way by making a guess at the solution and refining that guess until we are "close enough". These methods are called iterative methods, and most of the methods we cover require diagonal dominance. Diagonal dominance means that for each row $i$ in the matrix $\mathbf{A}$ of size $I$ by $I$

$$|A_{ii}| > \sum_{j=1, j\neq i}^{n} |A_{ij}|, \qquad i = 1 \ldots n$$

or in words, the diagonal is larger than the sum of the magnitudes all the off-diagonal elements of each row.

### BOX 9.1 NUMERICAL PRINCIPLE

When the absolute value of the each element of the main diagonal in the matrix is larger than the sum of the absolute values of the off diagonal terms in each row, the matrix is said to be diagonally dominant. In equation form an $n \times n$ matrix is diagonally dominant if

$$|A_{ii}| > \sum_{j=1, j\neq i}^{n} |A_{ij}|, \qquad i = 1 \ldots n.$$

## 9.1 JACOBI ITERATION

The first iterative method we will meet is the simplest, and it is called the Jacobi method. For a system $\mathbf{Ax} = \mathbf{b}$, what the Jacobi method does is start with a guess $\mathbf{x}^{(0)}$ and then solve each row of the system by evaluating the diagonal term at the $\ell + 1$ iteration and the off diagonal terms at the $\ell$ iteration. It then does this over and over until either the change $\|\mathbf{x}^{(\ell+1)} - \mathbf{x}^{(\ell)}\|$ is small, the residual $\|\mathbf{Ax}^{(\ell+1)} - \mathbf{b}\|$ is small, or the maximum number of iterations is met.

We will demonstrate how the Jacobi method works with an example. Consider the linear system of a $4 \times 4$ tri-diagonal matrix

$$\begin{pmatrix} 2.5 & -1 & 0 & 0 \\ -1 & 2.5 & -1 & 0 \\ 0 & -1 & 2.5 & -1 \\ 0 & 0 & -1 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Take an initial guess of $\mathbf{x}^{(0)} = [0, 0.5, 0.5, 0]^{\mathrm{T}}$. The first iteration looks like:

$$\begin{aligned} 2.5x_1^{(1)} - x_2^{(0)} &= 1 \\ -x_1^{(0)} + 2.5x_2^{(1)} - x_3^{(0)} &= 1 \\ -x_2^{(0)} + 2.5x_3^{(1)} - x_4^{(0)} &= 1 \\ -x_3^{(0)} + 2.5x_4^{(1)} &= 1. \end{aligned}$$

Each of these equations is independent of the other because we know the value of $x_i^{(0)}$ from our initial guess. Therefore, we can solve to get

$$x_1^{(1)} = 0.6,\ x_2^{(1)} = 0.6,\ x_3^{(1)} = 0.6,\ x_4^{(1)} = 0.6.$$

We can solve for $\mathbf{x}^{(2)}$ from the system

$$\begin{aligned} 2.5x_1^{(2)} - x_2^{(1)} &= 1, \\ -x_1^{(1)} + 2.5x_2^{(2)} - x_3^{(1)} &= 1, \\ -x_2^{(1)} + 2.5x_3^{(2)} - x_4^{(1)} &= 1, \\ -x_3^{(1)} + 2.5x_4^{(2)} &= 1, \end{aligned}$$

for $x_i^{(2)}$ and repeat the procedure until we are happy with the answer.

The general idea behind Jacobi iteration, is that given an initial guess $\mathbf{x}^{(0)}$, we compute

$$x_i^{(\ell+1)} = \frac{1}{A_{ii}} \left( b_i - \sum_{j=1, i \neq j}^{I} A_{ij} x_j^{(\ell)} \right), \qquad \text{for } i = 1 \ldots I,\ \ell = 0 \ldots .$$

We stop when the following criteria is met:

$$\frac{\|\mathbf{x}^{(\ell+1)} - \mathbf{x}^{(\ell)}\|_2}{\|\mathbf{x}^{(\ell+1)}\|_2} < \epsilon,$$

where $\epsilon$ is a user-defined tolerance, and the 2-norm is

$$\|\mathbf{y}\|_2 = \sqrt{\sum_{i=1}^{I} y_i^2}.$$

Now we will do these Jacobi iterations in Python on the system from above.

```
In [1]:  import numpy as np
         A = np.array([(2.5,-1,0,0),(-1,2.5,-1,0),(0,-1,2.5,-1),(0,0,-1,2.5)])
         print("Our matrix is\n",A)
         b = np.array([1.0,1,1,1])
         x = np.array([0,0.5,0.5,0])
         print("Our RHS is",b)
         print("The initial guess is",x)
```

```
Our matrix is
 [[ 2.5 -1.   0.   0. ]
 [-1.   2.5 -1.   0. ]
 [ 0.  -1.   2.5 -1. ]
 [ 0.   0.  -1.   2.5]]
Our RHS is [ 1.  1.  1.  1.]
The initial guess is [ 0.   0.5  0.5  0. ]
```

```
In [2]:  #first Jacobi iteration
         x_new = np.zeros(4)
         for row in range(4):
             x_new[row] = b[row]
             for column in range(4):
                 if column != row:
                     x_new[row] -= A[row,column]*x[column]
             x_new[row] /= A[row,row]
         #check difference
         relative_change = np.linalg.norm(x_new-x)/np.linalg.norm(x_new)
         print("New guess is",x_new)
         print("Norm of change is",relative_change)
```

```
New guess is [ 0.6  0.6  0.6  0.6]
Norm of change is 0.71686043892
```

```
In [3]:  #Second Jacobi Iteration
         x = x_new.copy() #replace old value
         x_new *= 0 #reset x_new
         for row in range(4):
             x_new[row] = b[row]
             for column in range(4):
                 if column != row:
                     x_new[row] -= A[row,column]*x[column]
             x_new[row] /= A[row,row]
         #check difference
         relative_change = np.linalg.norm(x_new-x)/np.linalg.norm(x_new)
         print("New guess is",x_new)
         print("Norm of change is",relative_change)
```

```
New guess is [ 0.64  0.88  0.88  0.64]
Norm of change is 0.259937622455
```

Notice that the change is going down. We will perform one more iteration in this example.

```
In [4]:  #Third Jacobi Iteration
         x = x_new.copy() #replace old value
         x_new *= 0 #reset x_new
         for row in range(4):
             x_new[row] = b[row]
             for column in range(4):
                 if column != row:
                     x_new[row] -= A[row,column]*x[column]
             x_new[row] /= A[row,row]
         #check difference
         relative_change = np.linalg.norm(x_new-x)/np.linalg.norm(x_new)
         print("New guess is",x_new)
         print("Norm of change is",relative_change)
```

**Data**: Matrix $\mathbf{A}$, vector $\mathbf{b}$, vector $\mathbf{x}_{\text{old}}$ (initial guess), `tolerance`
**Result**: The solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
$\mathbf{x} = 0$;
**while** $\|\mathbf{x} - \mathbf{x}_{\text{old}}\|_2 >$ `tolerance` **do**
  **for** $i \in [0,$ *number of rows of* $\mathbf{A}]$ **do**
    **for** $j \in [0,$ *number of columns of* $\mathbf{A}]$ **do**
      **if** $i \neq j$ **then**
        $x_i = x_i - A_{ij} x_{\text{old}\,j}$;
      **end**
    **end**
    $x_i = x_i / A_{ii}$;
  **end**
  Compute the change from $\mathbf{x}_{\text{old}}$ to $\mathbf{x}$;
  $\mathbf{x}_{\text{old}} = \mathbf{x}$;
**end**

Algorithm 9.1. Jacobi Iterations

```
New guess is [ 0.752  1.008  1.008  0.752]
Norm of change is 0.13524314758
```

We could continue by hand, but it makes sense at this point to write out the general algorithm. The Jacobi method is written in pseudocode in Algorithm 9.1.

This pseudocode is converted to a Python implementation in the code below. One change between the pseudocode and the Python implementation is that the user is not required to input an initial guess: if the user does not provide an initial guess, the initial guess is set to a random vector. Also, in practice it may be the case that the method does not converge because, for instance, the matrix is not diagonally dominant. To make sure that the function returns something in this case, we set a maximum number of iterations as a user input with a default value of 100.

```
In [5]: def JacobiSolve(A,b,x0=np.array([]),tol=1.0e-6,
                       max_iterations=100,LOUD=False,):
            """Solve a linear system by Jacobi iteration.
            Note: system must be diagonally dominant
            Args:
                A: N by N array
                b: array of length N
                x0: initial guess (if none given will be random)
                tol: Relative L2 norm tolerance for convergence
                max_iterations: maximum number of iterations
            Returns:
                The approximate solution to the linear system
            """
            [Nrow, Ncol] = A.shape
            assert Nrow == Ncol
            N = Nrow
            converged = False
```

```
    iteration = 1
    if (x0.size==0):
        #random initial guess
        x0 = np.random.rand(N)
    x = x0.copy()
    while not(converged):
        #replace old value
        x0 = x.copy()
        for row in range(N):
            x[row] = b[row]
            for column in range(N):
                if column != row:
                    x[row] -= A[row,column]*x0[column]
            x[row] /= A[row,row]
        relative_change = np.linalg.norm(x-x0)/np.linalg.norm(x)
        if (LOUD):
            print("Iteration",iteration,
                  ": Relative Change =",relative_change)
        if (relative_change < tol) or (iteration >= max_iterations):
            converged = True
        iteration += 1
    return x
```

We will try this function on our system from above. Only a portion of the output is shown because it takes tens of iterations to converge to the solution.

```
In [6]:  A = np.array([(2.5,-1,0,0),(-1,2.5,-1,0),(0,-1,2.5,-1),(0,0,-1,2.5)])
         print("Our matrix is\n",A)
         b = np.array([1.0,1,1,1])
         x = JacobiSolve(A,b,tol=1.0e-6,LOUD=True)
         print(x)

Our matrix is
 [[ 2.5 -1.   0.   0. ]
 [-1.   2.5 -1.   0. ]
 [ 0.  -1.   2.5 -1. ]
 [ 0.   0.  -1.   2.5]]
Iteration 1 : Relative Change = 1.0
Iteration 2 : Relative Change = 0.392232270276
Iteration 3 : Relative Change = 0.201990875657
...
Iteration 30 : Relative Change = 1.16661625519e-06
Iteration 31 : Relative Change = 7.55049332263e-07
[ 0.90908977  1.27272543  1.27272543  0.90908977]
```

Note that the iterations stopped when the change in the iterate was smaller than the tolerance $10^{-6}$. It is a good idea to check the answer by comparing $\mathbf{Ax}$ to $\mathbf{b}$.

```
In [7]:  #Check the answer
         print("Ax =",np.dot(A,x))
         print("b =",b)

Ax = [ 0.999999    0.99999837  0.99999837  0.999999  ]
b = [ 1.  1.  1.  1.]
```

We see that the solution does indeed have a small residual. Our Jacobi method could be used on a much larger matrix, as in the next example:

```
In [8]:  N = 100
         A = np.zeros((N,N))
         b = np.ones(N)
         #same structure as before
         for i in range(N):
             A[i,i] = 2.5
             if (i>0):
                 A[i,i-1] = -1
             if (i < N-1):
                 A[i,i+1] = -1
         x100 = JacobiSolve(A,b,tol=1.0e-6,LOUD=True)
         print(x100)

Iteration 1 : Relative Change = 1.0
Iteration 2 : Relative Change = 0.44285152295
Iteration 3 : Relative Change = 0.260809758969
...
Iteration 54 : Relative Change = 1.36670150228e-06
Iteration 55 : Relative Change = 1.09236419158e-06
Iteration 56 : Relative Change = 8.73100528097e-07
[ 0.99999923  1.49999848  1.74999775  1.87499706  1.93749641  1.96874581
  1.98437027  1.99218229  1.99608811  1.99804087  1.99901712  1.99950515
...
  1.99804087  1.99608811  1.99218229  1.98437027  1.96874581  1.93749641
  1.87499706  1.74999775  1.49999848  0.99999923]
```

### 9.1.1 Convergence of the Jacobi Method

We can demonstrate graphically the convergence of the Jacobi method for a 2 by 2 system. This will help us compare different methods and how they converge. For Jacobi, and subsequent methods, we will solve the system

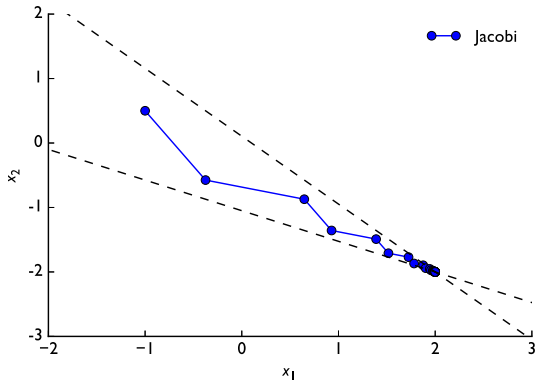
$$\begin{pmatrix} 2 & 1.9 \\ 1.9 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.2 \\ -4.2 \end{pmatrix}.$$

The solution to this system is $x_1 = 2$ and $x_2 = -2$. We will start the Jacobi method with an initial guess of $\mathbf{x}^{(0)} = (-1, 0.5)^{\mathrm{T}}$. This time we will accompany the solution with a graph showing how the approximate solution for Jacobi changes with each iteration.

```
In [9]: A = np.array([(2,1.9),(1.9,4)])
        solution = np.ones(2)*2
        solution[1] = -2
        b = np.dot(A,solution)
        x0 = np.array([-1,.5])
        xp, yp = JacobiSolve(A,b,x0=x0, LOUD=True)
        plt.plot(xp,yp,'o-',label='Jacobi')
        xpoint = np.linspace(-5,5,100)
        line1 = lambda x: (0.2 - 2*x)/1.9
```

```
        line2 = lambda x: (-4.2 - 1.9*x)/4
        plt.plot(xpoint,line1(xpoint),'-',color='black')
        plt.plot(xpoint,line2(xpoint),'-',color='black')
        plt.axis([-1.2,3,-3,2])
        plt.legend(); plt.xlabel('x$_1$'); plt.ylabel('x$_2$')
        plt.show()
```

```
Iteration 1 : Relative Change = 1.0
Iteration 2 : Relative Change = 0.643406164463
Iteration 3 : Relative Change = 0.244980153199
...
Iteration 33 : Relative Change = 1.10225320325e-06
Iteration 34 : Relative Change = 1.04360408096e-06
Iteration 35 : Relative Change = 4.97391183094e-07
```




In this figure, the dashed lines are the two lines that represent our system of equations

$$y = -\frac{2}{1.9}x + \frac{0.2}{1.9},$$
$$y = -\frac{1.9}{4}x - \frac{4.2}{4}.$$

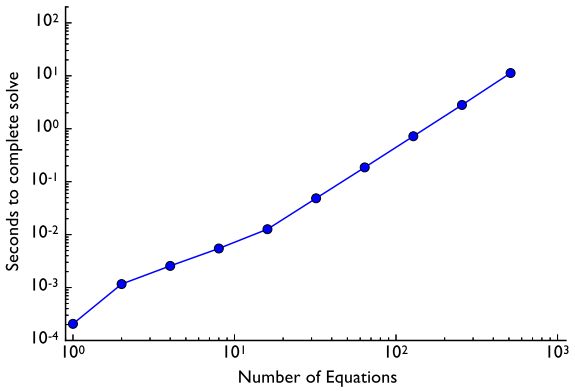
Where these lines intersect is the solution to our system. In this figure, there is one striking phenomenon we can see. The Jacobi solution bounces around in the area between the two lines. Because the approximation is always in this area and moves toward the solution, it will eventually get to the solution. Also, note that the initial steps are large, and the step size shrinks as the approximate solutions gets closer to the true answer. This is a common phenomenon for iterative methods. The behavior of Jacobi is similar when there are more equations than two, it just becomes harder to visualize the convergence.

### 9.1.2 Time to Solution for Jacobi Method

As we did with Gaussian elimination, we can time how long it takes to get a solution and see how that time scales with the number of unknowns by plotting the time to solution versus the number of equations on a log-log scale. The slope of this line (on the log-log scale) gives us the leading order behavior of the Jacobi method as the number of equations gets large.

Recall, for Gaussian elimination we observed a slope between 2 and 3.

```
In [10]: import time
         num_tests = 10
         I = 2**np.arange(num_tests)
         times = np.zeros(num_tests)
         for test in range(num_tests):
             N = I[test]
             A = np.zeros((N,N))
             b = np.ones(N)
             #same structure as before
             for i in range(N):
                 A[i,i] = 2.5
                 if (i>0):
                     A[i,i-1] = -1
                 if (i < N-1):
                     A[i,i+1] = -1
             start = time.clock()
             x = JacobiSolve(A,b)
             end = time.clock()
             times[test] = end-start
         plt.loglog(I,times,'o-')
         plt.xlabel("Number of Equations")
         plt.ylabel("Seconds to complete solve")
         plt.show()
         print("Approximate growth rate is n^",
               (np.log(times[test])-np.log(times[test-1]))/
              (np.log(I[test])-np.log(I[test-1])))
```




```
Approximate growth rate is n^ 2.00477104343
```

The slope of the line is 2, so that we can conclude that our implementation of Jacobi method requires $O(n^2)$ operations where $n$ is the number of equations. This means the time to solution is growing more slowly than we saw for Gauss elimination, which has a theoretical growth rate of three or $O(n^3)$. This means that for a large enough system, we will expect Jacobi be faster than Gaussian elimination. However, Jacobi can only be applied to diagonally dominant systems and, therefore, is not as general purpose as Gaussian elimination. Another

**Data**: Matrix $\mathbf{A}$, vector $\mathbf{b}$, vector $\mathbf{x}_{\text{old}}$ (initial guess), `tolerance`
**Result**: The solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
$\mathbf{x} = \mathbf{0}$;
**while** $|\mathbf{x} - \mathbf{x}_{\text{old}}|_2 >$ `tolerance` **do**
  $\mathbf{x}_{\text{old}} = \mathbf{x}$;
  $\mathbf{x} = (\mathbf{b} - \mathbf{Ax} + \text{diag}(\mathbf{A}) \cdot \mathbf{x})$;
**end**

Algorithm 9.2. Jacobi Algorithm without any `for` loops

difference between Jacobi and Gaussian elimination is that we cannot guarantee the number of iterations in a Jacobi solve, whereas with Gaussian elimination we know it takes one pass through the system to get a upper triangular form plus a back solve.

Just saying that Gaussian elimination is $O(n^3)$ and Jacobi is $O(n^2)$ does not mean that for a given problem Jacobi will be faster. The constant that multiples the leading order term could be large in one method making the comparison only valid as $n \to \infty$. For example, $n^3 < 1000n^2$ for $n < 1000$. Therefore, to compare two methods at a given system size requires knowing more than just the scaling. We will see this phenomenon concretely in the next section where we speed up Jacobi, but do not change its scaling as $n \to \infty$.

## 9.2 A FASTER JACOBI METHOD

With NumPy we can make a simpler Jacobi iteration that should also be faster by removing the inner two `for` loops. We can do this by using the fact that we can represent the Jacobi method using matrix-vector products, vector addition, and scalar division. To do this we notice that the inner update of a Jacobi iteration can be written as

$$x_i^{(\ell+1)} = \frac{1}{A_{ii}} \left( b_i - \sum_{j=1, i \neq j}^{I} A_{ij} x_j^{(\ell)} \right) = \frac{1}{A_{ii}} \left( b_i - \mathbf{a}_i \cdot \mathbf{x}^{(\ell)} + A_{ii} x_i^{(\ell)} \right),$$

where the $i$th row of the matrix is denoted as $\mathbf{a}_i$. Furthermore, we can write the entire update as a matrix vector product

$$\mathbf{x}^{(\ell+1)} = \frac{1}{\text{Diag}(\mathbf{A})} \left( \mathbf{b} - \mathbf{A} \cdot \mathbf{x}^{(\ell)} + \text{Diag}(\mathbf{A}) \cdot \mathbf{x}^{(\ell)} \right),$$

where the diagonal of the matrix is denoted as $\text{Diag}(\mathbf{A})$, and division by a vector is understood to be elementwise.

Therefore, instead of writing a `for` loop to perform the sum in the update and then using another `for` loop to look over the rows, we can use the optimized, built-in NumPy routine for a matrix vector product. The resulting function is not quite as easy to read, but the algorithm runs faster. We implement Algorithm 9.2 and test it out.

```
In [11]: def JacobiSolve_Short(A,b,x0=np.array([]),tol=1.0e-6,
                               max_iterations=100,LOUD=False):
             """Solve a linear system by Jacobi iteration.
             This implementation removes the for loops to make it faster
```

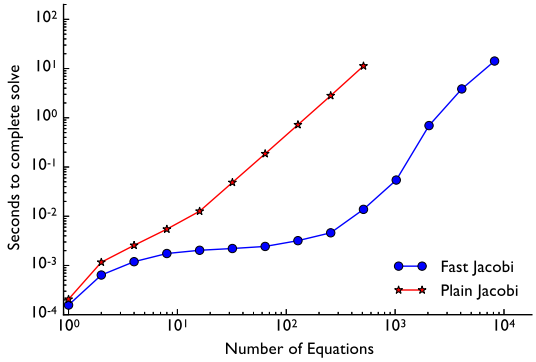
```
    Note: system must be diagonally dominant
    Args:
        A: N by N array
        b: array of length N
        tol: Relative L2 norm tolerance for convergence
        max_iterations: maximum number of iterations
    Returns:
        The approximate solution to the linear system
    """
    [Nrow, Ncol] = A.shape
    assert Nrow == Ncol
    N = Nrow
    converged = False
    iteration = 1
    if (x0.size==0):
        #random initial guess
        x0 = np.random.rand(N)
    x = x0.copy()
    while not(converged):
        x0 = x.copy() #replace old value
        #update is (b - whole row * x + diagonal part * x)/diagonal
        x = (b - np.dot(A,x0)+ A.diagonal()*x0)/A.diagonal()
        relative_change = np.linalg.norm(x-x0)/np.linalg.norm(x)
        if (LOUD):
            print("Iteration", iteration,
                  ": Relative Change =",relative_change)
        if (relative_change < tol) or (iteration >= max_iterations):
            converged = True
        iteration += 1
    return x
```

A reasonable first check is that the new algorithm gives the same answer as before.

```
In [12]: N = 100
         A = np.zeros((N,N))
         b = np.ones(N)
         #same structure as before
         for i in range(N):
             A[i,i] = 2.5
             if (i>0):
                 A[i,i-1] = -1
             if (i < N-1):
                 A[i,i+1] = -1
         x100_Short = JacobiSolve_Short(A,b,tol=1.0e-6,LOUD=False)
         print(x100_Short-x100)

[  0.00000000e+00   2.22044605e-16   0.00000000e+00   4.44089210e-16
   0.00000000e+00  -4.44089210e-16  -4.44089210e-16   0.00000000e+00
...
   0.00000000e+00  -4.44089210e-16   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
```

Those are almost identical. We can also demonstrate that this algorithm is much faster than the previous one.

Below, we show results from the same timing study that we did for plain Jacobi. The results in the figure show that we can solve more than 1000 equations in less than 1 second. The calculated growth rate of the time to complete the solve is $n^{2.12}$, basically quadratic scaling in the number of equations.




This curve is typical of the performance of an algorithm that is initially not filling up all the resources of the computer (fast memory called cache, for example). The bulk of the computation is spent on setting up the problem or moving memory around, two things that are not dependent on the number of floating point operations required. Eventually, once this set up time is small compared to the overall computation, we see that the scaling is about quadratic, which is better than the theoretical growth of Gaussian elimination. Despite having the same growth rate as the previous implementation, the absolute speed is much better.

One way to understand why the second implementation is faster is because the second has fewer `for` loops. When I have a `for` loop, the computer has to wait for each pass through the loop to complete before beginning the loop body again. With my "fast" implementation, all the work that needs to be done for an iteration is specified in a few lines using NumPy functions. This gives Python and NumPy the ability to keep the CPU of my computer completely filled up with tasks to do, instead of waiting for other tasks to finish and for memory to move around. As a result the time to solution is basically constant at less than 0.01 seconds until we get to about 256 equations.

## 9.3 GAUSS-SEIDEL

The Gauss-Seidel (GS) method is a twist on Jacobi iteration. The difference between the Jacobi and Gauss-Seidel method is that GS uses the most up to date information when performing an iteration. In equation form, Gauss-Seidel computes

$$x_i^{(\ell+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(\ell+1)} - \sum_{j=i+1}^{I} A_{ij}x_j^{(\ell)}\right), \qquad \text{for } i = 1\ldots I,\ \ell = 0\ldots. \tag{9.1}$$

In particular, when the method is updating $x_i$, it can use the updates already computed for the previous equations to get a better approximation: these are the $j = 1\ldots i-1$ terms on the right-hand side of Eq. (9.1). Because Gauss-Seidel is using the most up to date information,

**Data**: Matrix $\mathbf{A}$, vector $\mathbf{b}$, vector $\mathbf{x}$ (initial guess), `tolerance`
**Result**: The solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
$\mathbf{x}_{\text{old}} = \mathbf{0}$;
**while** $\|\mathbf{x} - \mathbf{x}_{\text{old}}\|_2 >$ `tolerance` **do**
  $x_i = b_i$;
  **for** $i \in [0,$ *number of rows of* $\mathbf{A}]$ **do**
    **for** $j \in [0,$ *number of columns of* $\mathbf{A}]$ **do**
      **if** $i \neq j$ **then**
        $x_i = x_i - A_{ij}x_j$;
      **end**
    **end**
    $x_i = x_i / A_{ii}$;
  **end**
  Compute the change from $\mathbf{x}_{\text{old}}$ to $\mathbf{x}$;
  $\mathbf{x}_{\text{old}} = \mathbf{x}$;
**end**

Algorithm 9.3. Gauss-Seidel Iterations

by the time it gets to the last equation it is basically using information all at iteration $\ell + 1$. Therefore, we might expect Gauss-Seidel to converge in fewer iterations than Jacobi. That being said, it may not be faster than Jacobi if the iterations take longer to compute than a Jacobi iteration.

The algorithm for Gauss-Seidel is given in Algorithm 9.3. Notice that the only change is that $\mathbf{x}_{\text{old}}$ is only used to compute the change during an iteration and is not otherwise used in the update.

Below, we implement a standard Gauss-Seidel algorithm in Python:

```
In [14]: def Gauss_Seidel_Solve(A,b,x0=np.array([]),tol=1.0e-6,
                                 max_iterations=100,LOUD=False):
             """Solve a linear system by Gauss-Seidel iteration.
             Note: system must be diagonally dominant
             Args:
                 A: N by N array
                 b: array of length N
                 x0: initial guess (if none given will be random)
                 tol: Relative L2 norm tolerance for convergence
                 max_iterations: maximum number of iterations
             Returns:
                 The approximate solution to the linear system
             """
             [Nrow, Ncol] = A.shape
             assert Nrow == Ncol
             N = Nrow
             converged = False
             iteration = 1
             if (x0.size==0):
                 x0 = np.random.rand(N) #random initial guess
             x = x0.copy()
             while not(converged):
                 x0 = x.copy() #replace old value
```

```
            for row in range(N):
                x[row] = b[row]
                for column in range(N):
                    if column != row:
                        #use x in update
                        x[row] -= A[row,column]*x[column]
                x[row] /= A[row,row]
            relative_change = np.linalg.norm(x-x0)/np.linalg.norm(x)
            if (LOUD):
                print("Iteration",iteration,
                      ": Relative Change =",relative_change)
            if (relative_change < tol) or (iteration >= max_iterations):
                converged = True
            iteration += 1
        return x
```

We now repeat the solution of a system that we previously solved using the Jacobi method, where it took 56 iterations. The code below performs the same test using Gauss-Seidel.

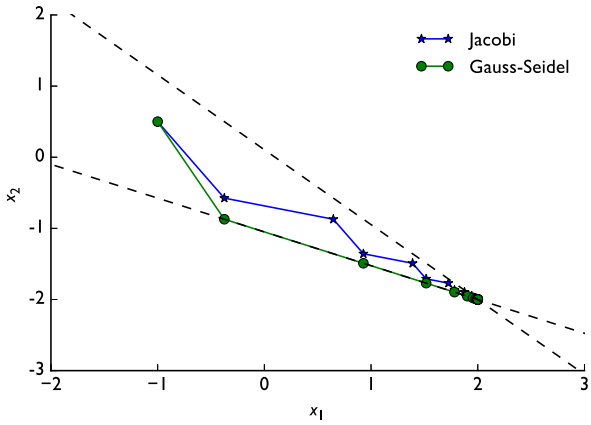
```
In [15]: N = 100
         A = np.zeros((N,N))
         b = np.ones(N)
         #same structure as before
         for i in range(N):
             A[i,i] = 2.5
             if (i>0):
                 A[i,i-1] = -1
             if (i < N-1):
                 A[i,i+1] = -1
         x100_GS = Gauss_Seidel_Solve(A,b,LOUD=True)
         print(x100_GS)
Iteration 1 : Relative Change = 1.0
Iteration 2 : Relative Change = 0.398660303888
Iteration 3 : Relative Change = 0.209236888361
...
Iteration 31 : Relative Change = 1.61012686252e-06
Iteration 32 : Relative Change = 1.07080293175e-06
Iteration 33 : Relative Change = 7.12128943502e-07
[ 0.99999896  1.49999827  1.74999781  1.8749975   1.9374973   1.96874716
  1.98437208  1.99218452  1.99609073  1.99804383  1.99902038  1.99950865
...
  1.99804605  1.99609309  1.99218699  1.98437462  1.96874972  1.93749981
  1.87499987  1.74999992  1.49999996  0.99999998]
```

The 33 iterations Gauss-Seidel required is quite a bit fewer than the 56 required by Jacobi (about 41% less). Using the most up-to-date information nearly cut the number of iterations in half. Compared to the Jacobi method, the complexity of the algorithm should be roughly the same per iteration because each iteration is solving a simple one-equation system for the next iteration's value for each unknown.

### 9.3.1 Convergence of Gauss-Seidel

We can compare the convergence between Jacobi and Gauss-Seidel graphically on the same 2 by 2 system as before and visually see how the iterations proceed. The figure below shows

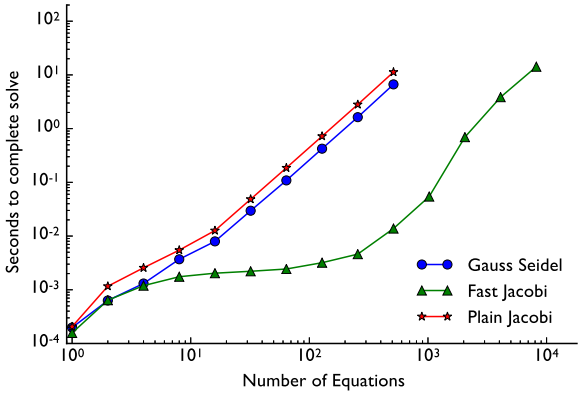
different behavior than the Jacobi method.




When we observe the convergence of Gauss-Seidel, we see that unlike Jacobi, it does not bounce between the lines. Rather it goes to one of the lines and then moves along that line to the solution. We can also see that Gauss-Seidel is "skipping" some of the Jacobi iterates, but getting to what looks like nearly the same points. For example, we can see that the third point for Gauss-Seidel is very near the fourth Jacobi point.

### 9.3.2 Time to Solution for Gauss-Seidel

We can do the same timing study we did for Jacobi. We expect Gauss-Seidel to be faster than the plain Jacobi (our first implementation), but probably slower than the fast Jacobi. We expect it to be faster than our simple Jacobi because it should take fewer iterations, and slower than fast Jacobi because there is not a way to implement Gauss-Seidel using matrix vector products, like we did in our fast Jacobi implementation and the reduction in iterations is not sufficient to overcome the additional cost per iteration. This is a visual demonstration of the benefit using up-to-date information in the iteration. In this test we observe a quadratic growth rate again.



Gauss-Seidel, like our two Jacobi implementations, appears to be an $O(n^2)$ algorithm. Due to the fewer number of iterations, the absolute time to solution is faster than plain Jacobi, but it is slower than fast Jacobi because each iteration is more expensive.

Before moving on, it is worthwhile to discuss in more detail why Gauss-Seidel cannot be done in the same fast manner as Jacobi. This is because each equation in Jacobi is independent of another: to update $x_{10}^{(\ell)}$ does not depend on the update to $x_9^{(\ell)}$, for example. This independence in updates is not the case for Gauss-Seidel iterations: $x_{10}^{(\ell)}$ cannot be updated until $x_9^{(\ell)}$ is. This means that I cannot write the algorithm as briefly as I could Jacobi. Gauss-Seidel has an intra-iteration dependence that makes each row's unknown dependent on those that come before it. This dependence is the fundamental reason we do not have a fast Gauss-Seidel implementation.

All hope is not lost for Gauss-Seidel, however. One can do something fancy called red-black Gauss-Seidel where all the even unknowns (called the red unknowns) are updated, and then all the odd unknowns (the black unknowns) are updated. Given the structure of the tridiagonal matrix above, to do a Gauss-Seidel update on an even unknown only requires knowledge of odd unknowns, and vice-versa. The updates for red and black unknowns can be done independently. This can make for a fast algorithm, however it gets more complicated because you need to split your system into "red" and "black" parts. As a general algorithm, this can be much trickier to code. For a tridiagonal system, the coding is pretty straightforward and comprises an exercise at the end of the chapter.

## 9.4 SUCCESSIVE OVER-RELAXATION

Though it sounds like an injury you can get sitting in your easy chair, over-relaxation takes the update from a Gauss-Seidel iteration and moves the solution further in that direction, i.e., it over relaxes it. The basic idea is to combine the current iterate with the Gauss-Seidel calculation of the next iterate using a factor $\omega$:

$$x_i^{(\ell+1)} = (1-\omega)x_i^{(\ell)} + \frac{\omega}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(\ell+1)} - \sum_{j=i+1}^{I} A_{ij}x_j^{(\ell)}\right), \quad \text{for } i = 1\ldots I,\ \ell = 0\ldots. \tag{9.2}$$

Clearly, if $\omega = 1$ then nothing has changed about the update. On the other hand, making $\omega > 1$ over-relaxes the system. In a sense it tries to take the correction to the current iteration and move the solution farther in that direction. The algorithm hardly changes at all. All we do is compute the update, and then compute the linear combination in Eq. (9.2).

```
In [18]: def SOR_Solve(A,b, x0= np.array([]),tol=1.0e-6,
                        omega=1,max_iterations=100,LOUD=False):
             """Solve a linear system by Gauss-Seidel iteration with SOR.
             Note: system must be diagonally dominant
             Args:
                 A: N by N array
                 b: array of length N
                 x0: initial guess (if none given will be random)
                 tol: Relative L2 norm tolerance for convergence
```

```
        omega: the over-relaxation parameter
        max_iterations: maximum number of iterations
    Returns:
        The approximate solution to the linear system
    """
    [Nrow, Ncol] = A.shape
    assert Nrow == Ncol
    N = Nrow
    converged = False
    iteration = 1
    if (x0.size==0):
        #random initial guess
        x0 = np.random.rand(N)
    x = x0.copy()
    while not(converged):
        x0 = x.copy() #replace old value
        for row in range(N):
            x[row] = b[row]
            for column in range(N):
                if column != row:
                    x[row] -= A[row,column]*x[column]
            x[row] /= A[row,row]
            x[row] = (1.0-omega) * x0[row] + omega*x[row]
        relative_change = np.linalg.norm(x-x0)/np.linalg.norm(x)
        if (LOUD):
            print("Iteration",iteration,
                  ": Relative Change =",relative_change)
        if (relative_change < tol) or (iteration >= max_iterations):
            converged = True
        iteration += 1
    return x
```

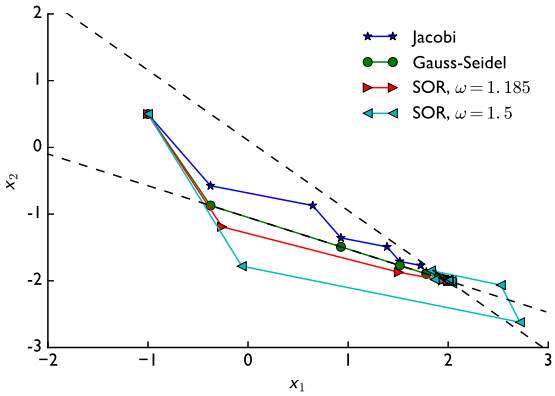
Using our standard example of 100 tri-diagonal equations, we can get a reduction in the number of iterations using $\omega = 1.2$:

```
In [19]: N = 100
         A = np.zeros((N,N))
         b = np.ones(N)
         #same structure as before
         for i in range(N):
             A[i,i] = 2.5
             if (i>0):
                 A[i,i-1] = -1
             if (i < N-1):
                 A[i,i+1] = -1
         x100_GS11 = SOR_Solve(A,b,omega=1.2,LOUD=True)

Iteration 1 : Relative Change = 1.0
Iteration 2 : Relative Change = 0.348994473141
Iteration 3 : Relative Change = 0.15758242743
...
Iteration 20 : Relative Change = 3.29823212222e-06
Iteration 21 : Relative Change = 1.76671341242e-06
Iteration 22 : Relative Change = 9.46306176483e-07
```

That saved us 11 iterations: Gauss-Seidel required 33 compared with 22 here. If we tweak $\omega$ some more, we get that $\omega \approx 1.3$ is the best value of $\omega$, leading to 19 iterations.

### 9.4.1 Convergence of SOR

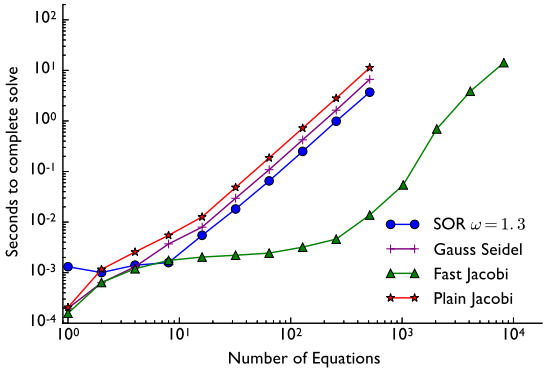
On our 2 by 2 system, we expect that the graphical convergence of SOR should look similar to that of Gauss-Seidel, but exaggerated because SOR is taking the Gauss-Seidel update and moving further in that direction. The figure below compares an SOR solution with a tuned value of $\omega$ with an SOR solution using too large a value of $\omega$.




With SOR the convergence with a tuned value of $\omega$, in this case $\omega = 1.185$, is similar to Gauss-Seidel, but faster. This happens because SOR allows the solution to go off the dashed lines during convergence to get a better approximation. Indeed, after the first iteration, the iterates zoom into the solution quickly. There can be too much of good thing, however. When the value of $\omega$ is too large, in this case $\omega = 1.5$, the approximation overshoots the true solution, before coming back. This leads to the solution requiring more iterations than Gauss-Seidel.

### 9.4.2 Time to Solution for SOR

The results from the timing study on SOR is given next. In this figure we observe the same trend as the standard Jacobi, and Gauss-Seidel methods.



Once again, we get a little faster because we saved on some iterations, also the growth rate shrunk mildly. It is worth noting at this point that the improvement from plain Jacobi to Gauss-Seidel to SOR are all numerical improvements: that is the numerical method improved resulting in fewer iterations. The speed increase going to Fast Jacobi is due to an implementation improvement: the fast implementation uses the computer's resources better.

One drawback with SOR is that it is not generally possible to determine ahead of time what the appropriate value of $\omega$ is. Experience on similar systems is generally required to determine the value of $\omega$. As a result, it may be more expensive to determine $\omega$ properly, by trial and error, than the savings it gives on a similar system. In the timing study above, the value of $\omega$ used was determined using this trial and error approach.

## 9.5 CONJUGATE GRADIENT

The conjugate gradient method will work on systems where the matrix $\mathbf{A}$ is symmetric, and positive definite (it does not need to be diagonally dominant in this case). Positive definite means that for any $\mathbf{x}$ that is not all zeros,

$$\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} > 0.$$

We will need this fact when deriving the method because we will divide by $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$. We will also need the definition of the residual for iteration $\ell$, $\mathbf{r}^{(\ell)}$, given by

$$\mathbf{r}^{(\ell)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(\ell)}.$$

To derive the method consider the function of $\mathbf{x}$:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x}, \tag{9.3}$$

the minimum of this function occurs when the gradient of $f(\mathbf{x})$ with respect to $\mathbf{x}$ is 0 or when

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

Therefore, we can derive an iterative method by computing $f(\mathbf{x}^{(0)})$ for an initial guess and refining the solution by decreasing the value of $f(\mathbf{x}^{(1)})$ and continuing on. In particular we write

$$\mathbf{x}^{(\ell+1)} = \mathbf{x}^{(\ell)} + \alpha_\ell \mathbf{s}^{(\ell)}.$$

The vector $\mathbf{s}$ is the search direction, and $\alpha_\ell$ is the step length. All that we have said up to this point is that at each iteration we are moving the solution in a particular direction $\mathbf{s}^{(\ell)}$ by an amount $\alpha_\ell$. We would like to pick $\alpha_\ell$ so that the error after step $\ell + 1$ the function is at a minimum along direction $\mathbf{s}^{(\ell)}$. To accomplish this we take the derivative of $f(\mathbf{x}^{(\ell+1)})$ with respect to $\alpha_\ell$ and set it to zero,

$$\frac{\partial}{\partial \alpha_\ell} f(\mathbf{x}^{(\ell+1)}) = \left(\frac{\partial}{\partial \mathbf{x}^{(\ell+1)}} f(\mathbf{x}^{(\ell+1)})\right)^{\mathrm{T}} \frac{\partial}{\partial \alpha_\ell}(\mathbf{x}^{(\ell+1)})$$

$$
\begin{aligned}
&= \left(\mathbf{r}^{(\ell+1)}\right)^{\mathrm{T}} \mathbf{s}^{(\ell)} \\
&= -\left(\mathbf{A}\left(\mathbf{x}^{(\ell)} + \alpha_\ell \mathbf{s}^{(\ell)}\right) - \mathbf{b}\right)^{\mathrm{T}} \mathbf{s}^{(\ell)} \\
&= \left(\mathbf{r}^{(\ell)} - \mathbf{A}\alpha_\ell \mathbf{s}^{(\ell)}\right)^{\mathrm{T}} \mathbf{s}^{(\ell)} \\
&= 0.
\end{aligned}
$$

When we solve this for $\alpha_\ell$, we get,

$$\alpha_\ell = \frac{\left(\mathbf{r}^{(\ell)}\right)^{\mathrm{T}} \mathbf{s}^{(\ell)}}{\left(\mathbf{s}^{(\ell)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{s}^{(\ell)}}. \tag{9.4}$$

We still have not specified the search direction.

The power of the conjugate gradient method is in how it selects the search direction. To understand this we will need to understand what it means for two vectors to be conjugate. Two vectors, $\mathbf{u}$ and $\mathbf{v}$, are conjugate with respect to a matrix $\mathbf{A}$ if

$$\mathbf{u}^{\mathrm{T}}\mathbf{A}\mathbf{v} = 0.$$

It is true that if $\mathbf{u}$ is conjugate to $\mathbf{v}$, then $\mathbf{v}$ is conjugate to $\mathbf{u}$. The property of conjugacy is related to orthogonality of two vectors. If two vectors are orthogonal then,

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^{\mathrm{T}}\mathbf{v} = 0.$$

When two vectors are conjugate, they are orthogonal after one is multiplied by $\mathbf{A}$.

Conjugate gradient seeks search directions that are conjugate to all the previous search directions. What this means is that every search direction is orthogonal to each of the previous directions when multiplied by $\mathbf{A}$. That is, we do not step in the same direction multiple times. To do this we want to write the search direction for the $\ell + 1$ step as a linear combination of the residual plus a constant times the previous step as

$$\mathbf{s}^{(\ell+1)} = \mathbf{r}^{(\ell+1)} + \beta_l \mathbf{s}^{(\ell)}. \tag{9.5}$$

We want step $\ell + 1$ to be conjugate to the previous step so that $(\mathbf{s}^{(\ell)})^{\mathrm{T}}\mathbf{A}\mathbf{s}^{(\ell+1)} = 0$. We want to pick $\beta_\ell$ so that this is the case. We will multiply Eq. (9.5) by $(\mathbf{s}^{(\ell)})^{\mathrm{T}}\mathbf{A}$ and set the result to zero to get

$$\left(\mathbf{s}^{(\ell)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{r}^{(\ell+1)} + \beta_\ell \left(\mathbf{s}^{(\ell)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{s}^{(\ell)} = 0. \tag{9.6}$$

Solving for $\beta_\ell$ gives

$$\beta_\ell = -\frac{\left(\mathbf{r}^{(\ell+1)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{s}^{(\ell)}}{\left(\mathbf{s}^{(\ell)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{s}^{(\ell)}}. \tag{9.7}$$

This process that will ensure that every subsequent search direction is conjugate to all previous search directions. To set the initial search direction, we set $\mathbf{s}^{(0)} = \mathbf{r}^{(0)}$. This also bounds

**Data**: Matrix $\mathbf{A}$, vector $\mathbf{b}$, vector $\mathbf{x}$ (initial guess)
**Result**: The solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
$\mathbf{residual} = \mathbf{b} - \mathbf{Ax}$;
$\mathbf{s} = \mathbf{residual}$;
**while** *change in* $\mathbf{x}$ *is small* **do**
    compute $\alpha$ defined by Eq. (9.4);
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{s}$;
    $\mathbf{residual} = \mathbf{b} - \mathbf{Ax}$;
    compute $\beta$ defined by Eq. (9.7);
    compute $\mathbf{s}$ defined by Eq. (9.5);
**end**

Algorithm 9.4. Conjugate Gradient Algorithm

the number of iterations in the solution. For an $N$ by $N$ matrix there are at most $N$ mutually conjugate vectors. This means that there are at most $N$ iterations in a CG solve because each step is in a conjugate direction *and* we minimize the error along each step.

We have talked through the mathematics of how conjugate gradient works, we write the method in pseudocode in Algorithm 9.4.

Notice that the CG algorithm is expressed entirely in matrix-vector products and vector addition/subtraction. This algorithm for conjugate gradient is implemented in Python below.

```
In [22]: def CG(A,b, x= np.array([]),tol=1.0e-6,
                max_iterations=100,LOUD=False):
             """Solve a linear system by Conjugate Gradient
             Note: system must be SPD
             Args:
                 A: N by N array
                 b: array of length N
                 x0: initial guess (if none given will be random)
                 tol: Relative L2 norm tolerance for convergence
                 max_iterations: maximum number of iterations
             Returns:
                 The approximate solution to the linear system
             """
             [Nrow, Ncol] = A.shape
             assert Nrow == Ncol
             N = Nrow
             converged = False
             iteration = 1
             if (x.size==0):
                 #random initial guess
                 x = np.random.rand(N)
             r = b - np.dot(A,x)
             s = r.copy()
             while not(converged):
                 denom = np.dot(s, np.dot(A,s))
                 alpha = np.dot(s,r)/denom
                 x = x + alpha*s
```

```
            r = b - np.dot(A,x)
            beta = - np.dot(r,np.dot(A,s))/denom
            s = r + beta * s
            relative_change = np.linalg.norm(r)
            if (LOUD):
                print("Iteration",iteration,
                      ": Relative Change =",relative_change)
            if (relative_change < tol) or (iteration >= max_iterations):
                converged = True
            iteration += 1
        return x
```

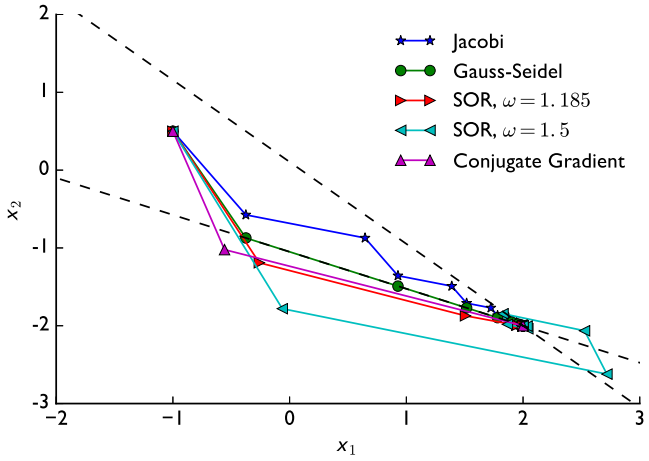
The results from our standard example are below.

```
In [23]: N = 100
         A = np.zeros((N,N))
         b = np.ones(N)
         #same structure as before
         for i in range(N):
             A[i,i] = 2.5
             if (i>0):
                 A[i,i-1] = -1
             if (i < N-1):
                 A[i,i+1] = -1
         x100_CG = CG(A,b,LOUD=True)


Iteration 1 : Relative Change = 8.2630468612
Iteration 2 : Relative Change = 5.11537771495
Iteration 3 : Relative Change = 1.90670704394
...
Iteration 22 : Relative Change = 2.34099376734e-06
Iteration 23 : Relative Change = 1.1809715533e-06
Iteration 24 : Relative Change = 5.42003345606e-07
```

A couple of things to note. The number of iterations of this algorithm is about the same as Gauss-Seidel with SOR, in other words it converges faster than Jacobi. Also, there are no `for` loops in the algorithm, it is all matrix-vector operations. Therefore, it should be competitive with our fast implementation of Jacobi in terms of speed.

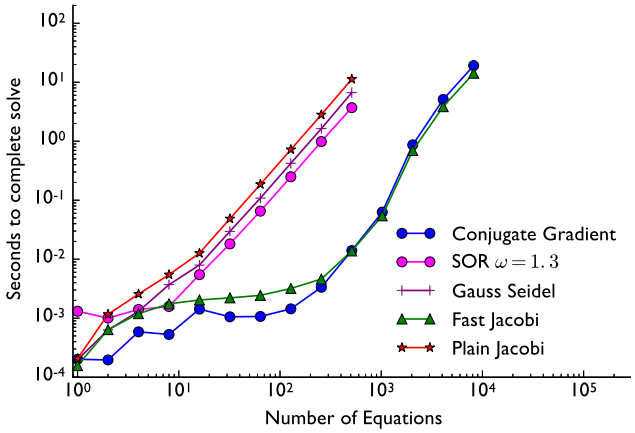
### 9.5.1 Convergence of CG

On our 2 by 2 system, we can compare the graphical convergence of conjugate gradient relative to the other methods. Given that our search directions will be conjugate, we expect that the solution should converge in two iterations because there are not more than two conjugate directions in a two-dimensional space. After the first iteration, the conjugate gradient approximate moves directly to the solution, as predicted.




## 9.5.2 Time to Solution for CG

We now replicate the timing study we did before for Jacobi, Gauss-Seidel, and SOR for CG. Given that the number of iterations was similar to SOR for our $N = 100$ system, we expect the scaling for CG to be similar to SOR. However, the CG algorithm is expressed in terms of matrix-vector product so we expect the time to solution should exhibit the behavior we saw with the fast Jacobi method: for small systems the time to solution will be roughly constant, and as the system gets larger the time to solution should scale as $O(N^2)$.




In these results the time to solution for conjugate gradient is roughly constant until there are hundreds of equations. Past this point, the time to solution grows roughly as the number of equations squared. In the end the time to solution is roughly equal to that from Jacobi. Given that the two methods have a similar time to solution, there are still reasons to favor conjugate gradient: the matrix need not be diagonally dominant and it is guaranteed to converge in $N$ iterations.

## 9.6 TAKING ADVANTAGE OF TRI-DIAGONAL FORM

In the example problems above we have been storing the whole $N$ by $N$ matrix, when really we only need about $3N$ numbers because the rest are zeros. We can do this by reformulating our matrix to be $N$ by 3 and putting the matrix elements in the appropriate place as done here:

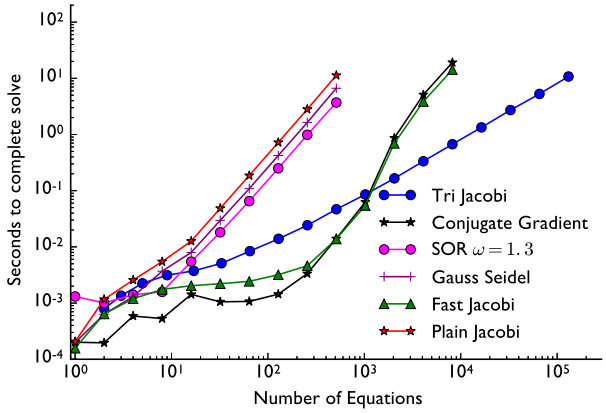
```
In [26]:  N = 10
          A_tri = np.zeros((N,3))
          b = np.ones(N)
          #same structure as before
          #but fill it more easily
          A_tri[:,1] = 2.5 #middle column is diagonal
          A_tri[:,0] = -1.0 #left column is left of diagonal
          A_tri[:,2] = -1.0 #right column is right of diagonal
          A_tri[0,0] = 0 #remove left column in first row
          A_tri[N-1,2] = 0 #remove right column in last row
          print("Our matrix is\n",A_tri)

Our matrix is
 [[ 0.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5 -1. ]
 [-1.   2.5  0. ]]
```

This is a much smaller matrix, but to use it we need to define special algorithms. We will do this for Jacobi because it is the simplest to show here. The modification for Gauss-Seidel, SOR, or conjugate gradient would also be possible (in fact implementing Gauss-Seidel is an exercise at the end of this chapter). The code below is our Jacobi method for a tri-diagonal system. In the code, the major change is that the main diagonal is always in column 1 of a row and the off diagonals are in columns 0 and 2 respectively.

```
In [27]: def JacobiTri(A,b,tol=1.0e-6,max_iterations=100,LOUD=False):
             """Solve a tridiagonal system by Jacobi iteration.
             Note: system must be diagonally dominant
             Args:
                 A: N by 3 array
                 b: array of length N
                 tol: Relative L2 norm tolerance for convergence
                 max_iterations: maximum number of iterations
             Returns:
                 The approximate solution to the linear system
             """
             [Nrow, Ncol] = A.shape
             assert 3 == Ncol
```

```
N = Nrow
converged = False
iteration = 1
x0 = np.random.rand(N) #random initial guess
x = np.zeros(N)
while not(converged):
    x0 = x.copy() #replace old value
    for i in range(1,N-1):
        x[i] = (b[i] - A[i,0]*x0[i-1] - A[i,2]*x0[i+1])/A[i,1]
    i = 0
    x[0] = (b[i]  - A[i,2]*x0[i+1])/A[i,1]
    i = N-1
    x[i] = (b[i] - A[i,0]*x0[i-1])/A[i,1]
    relative_change = (np.linalg.norm(x-x0)/
                       np.linalg.norm(x))
    if (LOUD):
        print("Iteration",iteration,
              ": Relative Change =",relative_change)
    if (relative_change < tol) or (iteration >= max_iterations):
        converged = True
    iteration += 1
return x_new
```

The results from our timing study indicate that the rate of growth in the solution time is linear, rather than quadratic. This is because the number of elements in the system is growing linearly in $n$, whereas the elements in a full matrix grows as the number of equations squared.




When the solution time is only growing linearly, we can run matrices as large as 100,000 by 100,000 in as little as 10 seconds. This is about 3 orders of magnitude improvement on a simple problem. This last figure tells makes three key points that are common in numerical methods:

**1.** The number of iterations is not necessarily as important as the speed at which the iterations are performed (compare SOR with fast Jacobi).

2. Eventually, the method that grows more slowly will be more efficient. This is seen in the fact that eventually the tridiagonal Jacobi method is the fastest because it grows linearly in the number of equations whereas the other methods grow quadratically in the number of equations.

This will be our last chapter on linear solvers. In the future we will use them to solve a variety of problems. Linear solvers also form the basis for several other numerical methods: nonlinear solvers, eigenvalue solvers, and discretizations for initial and boundary value problems. The fact that linear solvers are the foundation for so many other topics in scientific computing is one of the motivations for starting with that topic.

## FURTHER READING

There are a variety of iterative methods that can be used to solve linear systems that extend beyond those discussed here. The classic reference for these methods is Saad's treatment aimed at *sparse* linear systems [12], i.e., systems where the matrix has many zero entries, such as a tridiagonal matrix. Trefethen and Bau also provide detailed discussion of iterative methods [9].

## PROBLEMS

### Short Exercises

**9.1.** Write a Python function called `isSymmetric` which takes a single parameter `A` and checks if the NumPy matrix `A` is symmetric. It should return 1 if the matrix is symmetric, and 0 if the matrix is non-symmetric.

**9.2.** Write a Python function called `isDiagonallyDominant` which takes a single parameter `A` and checks if the NumPy matrix `A` is diagonally dominant. It should return 1 if the matrix is diagonally dominant, and 0 if the matrix is not diagonally dominant.

### Programming Projects

#### *1. Exiting Gracefully*

The Jacobi and Gauss-Seidel implementations in this chapter have a maximum number of iterations before they return a solution. They do not, however, tell the user that the maximum number of iterations was reached. Modify the Jacobi implementation given above to alert the user that the maximum number of iterations has been reached. You can do this by inserting a `print` statement that is executed or by using an `assert` statement.

#### *2. Tri-diagonal Gauss-Seidel*

Write a Gauss-Seidel solver for tri-diagonal matrices. The implementation should take as input a tri-diagonal matrix just as the tri-diagonal Jacobi defined previously. Test your implementations on the same timing study performed above in Section 9.6. Comment on the results.

```
import numpy as np
import math
delta = 0.05;
L = 1.0;
k = 0.001;
ndim = round(L/delta)
nCells = ndim*ndim;
A = np.zeros((nCells,nCells));
b = np.zeros(nCells)
#save us some work for later
idelta2 = 1.0/(delta*delta);

#now fill in A and b
for cellVar in range(nCells):
    xCell = cellVar % ndim; #x % y means x modulo y
    yCell = (cellVar-xCell)/ndim;
    xVal = xCell*delta + 0.5*delta;
    yVal = yCell*delta + 0.5*delta;
    #put source only in the middle of the problem
    if ( ( math.fabs(xVal - L*0.5) < .25*L) and
         ( math.fabs(yVal - L*0.5) < .25*L) ):
        b[cellVar] = 1;
    #end if

    A[cellVar,cellVar] = 4.0*k*idelta2;

    if (xCell > 0):
      A[cellVar,ndim*yCell + xCell -1] = -k*idelta2;
    if (xCell < ndim-1):
      A[cellVar,ndim*yCell + xCell + 1] = -k*idelta2;
    if (yCell > 0):
      A[cellVar,ndim*(yCell-1) + xCell] = -k*idelta2;
    if (yCell < ndim-1):
      A[cellVar,ndim*(yCell+1) + xCell] = -k*idelta2;

if (nCells <= 20):
    #print the matrix
    print("The A matrix in Ax = b is\n",A)

    #print the righthand side
    print("The RHS is",b)

x, residual = CG(A,b,LOUD=True,max_iterations=1000)
```

Algorithm 9.5. 2-D Heat Equation Code

### *3. 2-D Heat Equation*

Below is a program in Algorithm 9.5, which builds a matrix and righthand side for a heat conduction problem in 2-D. The discretization of the 2-D heat equation gives a linear system $\mathbf{Ax} = \mathbf{b}$ where the solution vector $\mathbf{x}$ is the temperature at the grid points of the 2-D domain. In particular

1. The 2-D heat equation in a homogeneous material of constant conductivity $k$ with a uniform volumetric heat source $q$ and zero-temperature conditions on the boundary of the rectangular domain of length $L_x$ and width $L_y$ is

$$-k\nabla^2 T = q, \qquad \text{for } x \in [0, L_x] \quad y \in [0, L_y].$$

   With the boundary condition

$$T(x, y) = 0 \qquad \text{for } x, y \text{ on the boundary.}$$

2. The 2-D Laplacian is discretized using a mathematical technique known as finite differences (which we will see later on).
3. As a result of the spatial discretization, the heat equation forms a linear system of the form $\mathbf{Ax} = \mathbf{b}$. The code below forms this matrix.
4. The size of this matrix is `nCells` is determined by the value of $\Delta$ (the distance between points at which we want to evaluate the temperature), and the size of $L_x$ and $L_y$. In the code $L_x = L_y$ and this value is called $L$. The total number of values in each direction is `ndim`$= L/\Delta$ leading to a total number of unknowns is `nCells = ndim * ndim`.

Your work:

1. Look at Algorithm 9.5. It is a working program except that the conjugate gradient function has been deleted from it. To make it run you will have to add in a conjugate gradient solver. The conjugate gradient solver given in class should return a vector containing the residual at each iteration.
2. Plot the logarithm of the 2-norm of the residual, $\|\mathbf{Ax}_i - \mathbf{b}\|_2$ error versus the iteration number.
3. The solution vector **x** contains the `nCells = ndim * ndim` unknowns. The $x$ and $y$ positions corresponding to a given row of the matrix are defined in the lines that assign `xVal` and `yVal`. Plot the 2-D temperature distribution in the two following cases:
   **a.** A very coarse grid (small number of grid points)
   **b.** A fine grid (high number of grid points).