Table of contents

1 Iterative solutions of linear equations

In the previous section we looked at what are known as *direct* methods for solving systems of linear equations. They are guaranteed to produce a solution with a fixed amount of work (we can even prove this in exact arithmetic!), but this fixed amount of work may be **very** large.

For a general $n \times n$ system of linear equations $A\vec{x} = \vec{b}$, the computation expense of all direct methods if $O(n^3)$. The amount of storage required for these approaches is $O(n^2)$ which is dominated by the cost of storing the matrix A. As n becomes larger the storage and computation work required limit the practicality of direct approaches.

As an alternative, we will propose some **iterative methods**. Iterative methods produce a sequence $(\vec{x}^{(k)})$ of approximations to the solution of the linear system of equations $A\vec{x} = \vec{b}$. The iteration is defined recursively and is typically of the form:

$$\vec{x}^{(k+1)} = \vec{F}(\vec{x}^{(k)}),$$

where $\vec{x}^{(k)}$ is now a vector of values and $\vec{F}$ is some vector function (which needs to be defined to define the method). We will need to choose a starting value $\vec{x}^{(k)}$ but there is often a reasonable approximation which can be used. Once all this is defined, we still need to decide when we need to stop!

Some very bad examples

Example 1
Consider

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)}.$$

Each iteration is very cheap to compute but very inaccurate - it never converges!
Example 2
Consider

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)} + A^{-1}(\vec{b} - A\vec{x}^{(k)}).$$

Each iteration is very expensive to compute - you have to invert A! - but it converges in just one step since

$$\begin{aligned} A\vec{x}^{(k+1)} &= A\vec{x}^{(k)} + AA^{-1}(\vec{b} - A\vec{x}^{(k)}) \\ &= A\vec{x}^{(k)} + \vec{b} - A\vec{x}^{(k)} \\ &= \vec{b}. \end{aligned}$$

Key idea

The key point here is we want a method which is both cheap to compute but converges quickly to the solution. One way to do this is to construct iteration given by

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)} + P(\vec{b} - A\vec{x}^{(k)}). \tag{1}$$

for some matrix P such that

- P is easy to compute, or the matrix vector product $P\vec{r}$ is easy to compute,
- P approximates A^{-1} well enough that the algorithm converges in few iterations.

We call $\vec{b} - A\vec{x}^{(k)} = \vec{r}$ the **residual**. Note in the above examples we would have $P = O$ (the zero matrix) or $P = A^{-1}$.

1.1 Jacobi iteration

One simple choice for P is given by the Jacobi method where we take $P = D^{-1}$ where D is the diagonal of A:

$$D_{ii} = A_{ii} \quad \text{and} \quad D_{ij} = 0 \text{ for } i \neq j.$$

The **Jacobi iteration** is given by

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)})$$

D is a *diagonal matrix*, so D^{-1} is trivial to form (as long as the diagonal entries are all nonzero):

$$(D^{-1})_{ii} = \frac{1}{D_{ii}} \quad \text{and} \quad (D^{-1})_{ij} = 0 \text{ for } i \neq j.$$

Remark.

- The cost of one iteration is $O(n^2)$ for a full matrix, and this is dominated by the matrix-vector product $A\vec{x}^{(k)}$.
- This cost can be reduced to $O(n)$ if the matrix A is sparse - this is when iterative methods are especially attractive (TODO add future ref).
- The amount of work also depends on the number of iterations required to get a "satisfactory" solution.
  - The number of iterations depends on the matrix;
  - Fewer iterations are needed for a less accurate solution;
  - A good initial estimate $\vec{x}^{(0)}$ reduces the required number of iterations.
- Unfortunately, the iteration might not converge!

The Jacobi iteration updates all elements of $\vec{x}^{(k)}$ *simultaneously* to get $\vec{x}^{(k+1)}$. Writing the method out component by component gives

$$\begin{aligned}
x_1^{(k+1)} &= x_1^{(k)} + \frac{1}{A_{11}}\left(b_1 - \sum_{j=1}^{n} A_{1j}x_j^{(k)}\right) \\
x_2^{(k+1)} &= x_2^{(k)} + \frac{1}{A_{22}}\left(b_2 - \sum_{j=1}^{n} A_{2j}x_j^{(k)}\right) \\
&\vdots \qquad\quad \vdots \\
x_n^{(k+1)} &= x_n^{(k)} + \frac{1}{A_{nn}}\left(b_n - \sum_{j=1}^{n} A_{nj}x_j^{(k)}\right).
\end{aligned}$$

Note that once the first step has been taken, $x_1^{(k+1)}$ is already known, but the Jacobi iteration does not make use of this information!

Example 1

Take two iterations of Jacobi iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1, 1)^T$:

$$\begin{pmatrix} 2 & 1 \\ -1 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ 0.5 \end{pmatrix}$$

Starting from $\vec{x}^{(0)} = (1,1)^T$, the first iteration is

$$\begin{aligned} x_1^{(1)} &= x_1^{(0)} + \frac{1}{A_{11}}\left(b_1 - A_{11}x_1^{(0)} - A_{12}x_2^{(0)}\right) \\ &= 1 + \frac{1}{2}(3.5 - 2\times 1 - 1\times 1) = 1.25 \\ x_2^{(1)} &= x_2^{(0)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(0)} - A_{22}x_2^{(0)}\right) \\ &= 1 + \frac{1}{4}(0.5 - (-1)\times 1 - 4\times 1) = 0.375. \end{aligned}$$

So we have $\vec{x}^{(1)} = (1.25, 0.375)^T$. Then the second iteration is

$$\begin{aligned} x_1^{(2)} &= x_1^{(1)} + \frac{1}{A_{11}}\left(b_1 - A_{11}x_1^{(1)} - A_{12}x_2^{(1)}\right) \\ &= 1.25 + \frac{1}{2}(3.5 - 2\times 1.25 - 1\times 0.375) = 1.5625 \\ x_2^{(2)} &= x_2^{(1)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(1)} - A_{22}x_2^{(1)}\right) \\ &= 0.375 + \frac{1}{4}(0.5 - (-1)\times 1.25 - 4\times 0.375) = 0.4375. \end{aligned}$$

So we have $\vec{x}^{(2)} = (1.5625, 0.4375)$.
Note the only difference between the formulae for Iteration 1 and 2 is the iteration number, the superscript in brackets. The exact solution is given by $\vec{x} = (1.5, 0.5)^T$.

We note that we can also slightly simplify the way the Jacobi iteration is written. We can expand A into $A = L + D + U$, where L and U are the parts of the matrix from below and above the diagonal respectively:

$$L_{ij} = \begin{cases} A_{ij} & \text{if } i < j \\ 0 & \text{if } i \geq j, \end{cases} \qquad U_{ij} = \begin{cases} A_{ij} & \text{if } i > j \\ 0 & \text{if } i \leq j. \end{cases}$$

The we can calculate that:

$$\begin{aligned} \vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} + D^{-1}(\vec{b} - (L + D + U)\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} - D^{-1}D\vec{x}^{(k)} + D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} - \vec{x}^{(k)} + D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}) \\ &= D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}). \end{aligned}$$

In this formulation, we do not explicitly form the residual as part of the computations. In practical situations, this may be a simpler formulation we can use if we have knowledge of the coefficients of A, but this is not always true!

1.2 Gauss-Seidel iteration

As an alternative to Jacobi iteration, the iteration might use $x_i^{(k+1)}$ as soon as it is calculated (rather than using the previous iteration), giving

$$\begin{aligned}
x_1^{(k+1)} &= x_1^{(k)} + \frac{1}{A_{11}}\left(b_1 - \sum_{j=1}^{n} A_{1j}x_j^{(k)}\right)\\
x_2^{(k+1)} &= x_2^{(k)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(k+1)} - \sum_{j=2}^{n} A_{2j}x_j^{(k)}\right)\\
x_3^{(k+1)} &= x_3^{(k)} + \frac{1}{A_{33}}\left(b_3 - \sum_{j=1}^{2} A_{3j}x_j^{(k+1)} - \sum_{j=3}^{n} A_{3j}x_j^{(k)}\right)\\
&\vdots \qquad \vdots\\
x_i^{(k+1)} &= x_i^{(k)} + \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} A_{ij}x_j^{(k)}\right)\\
&\vdots \qquad \vdots\\
x_n^{(k+1)} &= x_n^{(k)} + \frac{1}{A_{nn}}\left(b_n - \sum_{j=1}^{n-1} A_{nj}x_j^{(k+1)} - A_{nn}x_n^{(k)}\right).
\end{aligned}$$

Consider the system $A\vec{x} = b$ with the matrix A split as $A = L + D + U$ where D is the diagonal of A, L contains the elements below the diagonal and U contains the elements above the diagonal. The componentwise iteration above can be written in matrix form as

$$\begin{aligned}
\vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}(\vec{b} - L\vec{x}^{(k+1)} - (D+U)\vec{x}^{(k)})\\
&= \vec{x}^{(k)} - D^{-1}L\vec{x}^{(k+1)} + D^{-1}(\vec{b} - (D+U)\vec{x}^{(k)})\\
&= \vec{x}^{(k)} - D^{-1}L\vec{x}^{(k+1)} + D^{-1}L\vec{x}^{(k)} + D^{-1}(\vec{b} - (L+D+U)\vec{x}^{(k)})\\
\vec{x}^{(k+1)} + D^{-1}L\vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}L\vec{x}^{(k)} + D^{-1}(\vec{b} - (L+D+U)\vec{x}^{(k)})\\
D^{-1}(D+L)\vec{x}^{(k+1)} &= D^{-1}(D+L)\vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)})\\
(D+L)\vec{x}^{(k+1)} &= DD^{-1}(D+L)\vec{x}^{(k)} + DD^{-1}(\vec{b} - A\vec{x}^{(k)})\\
&= (D+L)\vec{x}^{(k)} + (\vec{b} - A\vec{x}^{(k)})\\
\vec{x}^{(k+1)} &= (D+L)^{-1}(D+L)\vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)})\\
&= \vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)}).
\end{aligned}$$

…and hence the **Gauss-Seidel** iteration

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)}).$$

That is, we use $P = (D + L)^{-1}$ in (1).

In general, we don't form the inverse of $D + L$ explicitly here since it is more complicated to do so than for simply computing the inverse of D.

Example 1

Take two iterations of Gauss-Seidel iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1, 1)^T$:

$$\begin{pmatrix} 2 & 1 \\ -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ 0.5 \end{pmatrix}$$

Starting from $\vec{x}^{(0)} = (1, 1)^T$ we have
Iteration 1:

$$\begin{aligned} x_1^{(1)} &= x_1^{(0)} + \frac{1}{A_{11}}(b_1 - A_{11}x_1^{(0)} - A_{12}x_2^{(0)}) \\ &= 2 + \frac{1}{2}(3.5 - 1 \times 2 - 1 \times 1) = 2.25 \\ x_2^{(1)} &= x_2^{(0)} + \frac{1}{A_{22}}(b_2 - A_{21}x_1^{(1)} - A_{22}x_2^{(0)}) \\ &= 1 + \frac{1}{4}(0.5 - (-1) \times 2.25 - 4 \times 1) = 0.6875. \end{aligned}$$

Iteration 2:

$$\begin{aligned} x_1^{(2)} &= x_1^{(1)} + \frac{1}{A_{11}}(b_1 - A_{11}x_1^{(1)} - A_{12}x_2^{(1)}) \\ &= 1.25 + \frac{1}{2}(3.5 - 2 \times 1.25 - 1 \times 0.4375) = 1.53125 \\ x_2^{(2)} &= x_2^{(1)} + \frac{1}{A_{22}}(b_2 - A_{21}x_1^{(2)} - A_{22}x_2^{(1)}) \\ &= 0.4375 + \frac{1}{4}(0.5 - (-1) \times 1.53125 - 4 \times 0.4375) = 0.5078125. \end{aligned}$$

Again, note the changes in the iteration number on the right hand side of these equations, especially the differences against the Jacobi method.

- What happens if the initial estimate is altered to $\vec{x}^{(0)} = (2, 1)^T$ (homework).

Example 2 (homework)

Take one iteration of (i) Jacobi iteration; (ii) Gauss-Seidel iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1, 2, 3)^T$:

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

Note that the exact solution to this system is $x_1 = 2, x_2 = 2, x_3 = 2$.

Remark.

- Here both methods converge, but fairly slowly. They might not converge at all!
- We will discuss convergence and stopping criteria in the next lecture.
- The Gauss-Seidel iteration generally out-performs the Jacobi iteration.
- Performance can depend on the order in which the equations are written.
- Both iterative algorithms can be made faster and more efficient for sparse systems of equations (far more than direct methods).

1.3 Python version of iterative methods

```
def jacobi_iteration(A, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n = system_size(A, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        for i in range(n):
            Axi = 0.0
            for j in range(n):
                Axi += A[i, j] * x[j]
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Axi)
        x = xnew.copy()

        if verbose:
```

```
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x


def gauss_seidel_iteration(A, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n = system_size(A, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        for i in range(n):
            Axi = 0.0
            for j in range(i):
                Axi += A[i, j] * xnew[j]
            for j in range(i, n):
                Axi += A[i, j] * x[j]
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Axi)
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x
```

```
A = np.array([[2.0, 1.0], [-1.0, 4.0]])
b = np.array([[3.5], [0.5]])
x0 = np.array([[1.0], [1.0]])

print("jacobi iteration")
x = jacobi_iteration(A, b, x0, 5, verbose=True)
print()
```

```
print("gauss seidel iteration")
x = gauss_seidel_iteration(A, b, x0, 5, verbose=True)
print()
```

```
jacobi iteration
starting value: x.T = [  1.0,  1.0 ]
after iter=0: x.T = [  1.250,  0.375 ]
after iter=1: x.T = [  1.5625,  0.4375 ]
after iter=2: x.T = [  1.53125,  0.51562 ]
after iter=3: x.T = [  1.49219,  0.50781 ]
after iter=4: x.T = [  1.49609,  0.49805 ]

gauss seidel iteration
starting value: x.T = [  1.0,  1.0 ]
after iter=0: x.T = [  1.2500,  0.4375 ]
after iter=1: x.T = [  1.53125,  0.50781 ]
after iter=2: x.T = [  1.49609,  0.49902 ]
after iter=3: x.T = [  1.50049,  0.50012 ]
after iter=4: x.T = [  1.49994,  0.49998 ]
```

1.4 Sparse Matrices

We met sparse matrices as an example of a special matrix format when we first thought about systems of linear equations. Sparse matrices are very common in applications and have a structure which is very useful when used with iterative methods. There are two main ways in which sparse matrices can be exploited in order to obtain benefits within iterative methods.

- The storage can be reduced from $O(n^2)$.
- The cost per iteration can be reduced from $O(n^2)$.

Recall that a sparse matrix is defined to be such that it has at most αn non-zero entries (where α is independent of n). Typically this happens when we know there are at most α non-zero entries in any row.

The simplest way in which a sparse matrix is stored is using three arrays:

- an array of floating point numbers (`A_real` say) that stores the non-zero entries;
- an array of integers (`I_row` say) that stores the row number of the corresponding entry in the real array;
- an array of integers (`I_col` say) that stores the column numbers of the corresponding entry in the real array.

This requires just $3\alpha n$ units of storage - i.e. $O(n)$.

Given the above storage pattern, the following algorithm will execute a sparse matrix-vector multiplication ($\vec{z} = A\vec{y}$) in $O(n)$ operations:

```
z = np.zeros((n, 1))
for k in range(nonzero):
    z[I_row[k]] = z[I_row[k]] + A_real[k] * y[I_col[k]]
```

- Here `nonzero` is the number of non-zero entries in the matrix.
- Note that the cost of this operation is $O(n)$ as required.

1.4.1 Python experiments

```
def system_size_sparse(A_real, I_row, I_col, b):
    n = len(b)
    nonzero = len(A_real)
    assert nonzero == len(I_row)
    assert nonzero == len(I_col)

    return n, nonzero
```

First let's adapt our implementations to use this sparse matrix format:

```
def jacobi_iteration_sparse(A_real, I_row, I_col, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n, nonzero = system_size_sparse(A_real, I_row, I_col, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    # determine diagonal
    # D[i] should be A_{ii}
    D = np.zeros_like(x)
    for k in range(nonzero):
```

```
        if I_row[k] == I_col[k]:
            D[I_row[k]] = A_real[k]

    for iter in range(max_iter):
        # precompute Ax
        Ax = np.zeros_like(x)
        for k in range(nonzero):
            Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * x[I_col[k]]

        for i in range(n):
            xnew[i] = x[i] + 1.0 / D[i] * (b[i] - Ax[i])
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x


def gauss_seidel_iteration_sparse(A_real, I_row, I_col, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n, nonzero = system_size_sparse(A_real, I_row, I_col, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        # precompute Ax using xnew if i < j
        Ax = np.zeros_like(x)
        for k in range(nonzero):
            if I_row[k] < I_col[k]:
                Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * xnew[I_col[k]]
            else:
                Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * x[I_col[k]]
```

```
        for i in range(n):
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Ax[i])
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x
```

Then we can test the two different implementations of the methods:

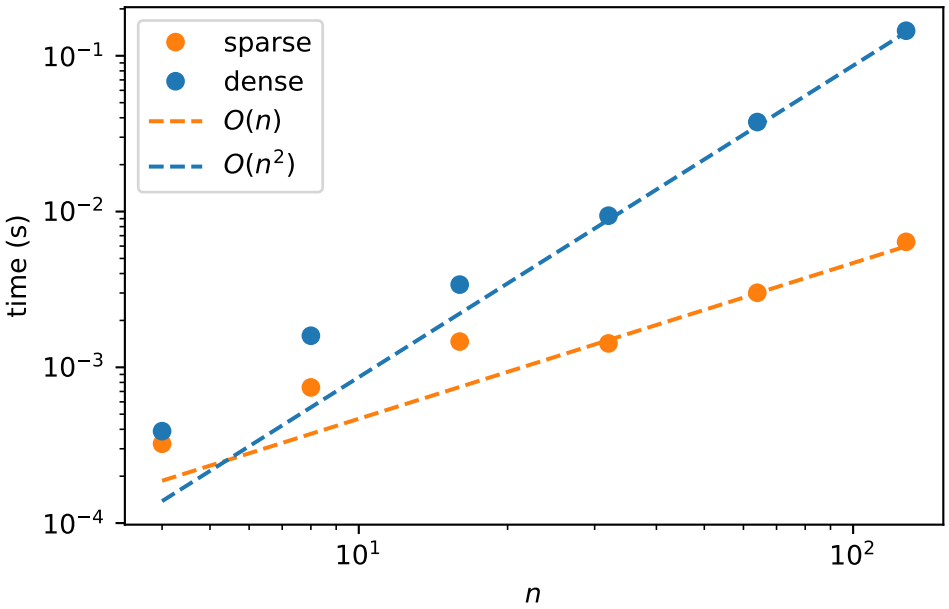
```
# random matrix
n = 4
nonzero = 10
A_real, I_row, I_col, b = random_sparse_system(n, nonzero)
print("sparse matrix:")
print("A_real =", A_real)
print("I_row = ", I_row)
print("I_col = ", I_col)
print()

# convert to dense for comparison
A_dense = to_dense(A_real, I_row, I_col)
print("dense matrix:")
print_array(A_dense)
print()

# starting guess
x0 = np.zeros((n, 1))

print("jacobi with sparse matrix")
x_sparse = jacobi_iteration_sparse(A_real, I_row, I_col, b, x0, max_iter=5, verbose=True)
print()

print("jacobi with dense matrix")
x_dense = jacobi_iteration(A_dense, b, x0, max_iter=5, verbose=True)
print()
```

```
sparse matrix:
A_real = [ -9.     5.     6.     4.   -13.5    7.5   29.25   6.   -11.25  18.5
```

```
   7.5    5.    26.5  -11.25 -13.5   -9.  ]
I_row =  [0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3]
I_col =  [3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0]

dense matrix:
A_dense = [  4.00,  6.00,  5.00, -9.00 ]
          [  6.00, 29.25,  7.50, -13.50 ]
          [  5.00,  7.50, 18.50, -11.25 ]
          [ -9.00, -13.50, -11.25, 26.50 ]

jacobi with sparse matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  1.500000,  1.000000,  1.067568, -0.273585 ]
after iter=1: x.T = [ -1.950025,  0.292302,  0.090388,  1.198496 ]
after iter=2: x.T = [  3.64518,  1.92998,  2.20492, -0.74858 ]
after iter=3: x.T = [ -5.83541, -0.65859, -1.15526,  2.88365 ]
after iter=4: x.T = [ 10.42016,  3.82414,  4.66527, -3.08137 ]

jacobi with dense matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  1.500000,  1.000000,  1.067568, -0.273585 ]
after iter=1: x.T = [ -1.950025,  0.292302,  0.090388,  1.198496 ]
after iter=2: x.T = [  3.64518,  1.92998,  2.20492, -0.74858 ]
after iter=3: x.T = [ -5.83541, -0.65859, -1.15526,  2.88365 ]
after iter=4: x.T = [ 10.42016,  3.82414,  4.66527, -3.08137 ]
```

We see that we get the same results!

Now let's see how long it takes to get a solution. The following plot shows the run times of using the two different implementations of the Jacobi method. We see that, as expected, the run time of the dense formulation is $O(n^2)$ and the run time of the sparse formulation is $O(n)$.



We say "as expected" because we have already counted the number of operations per iteration and these implementations compute for a fixed number of iterations. In the next section, we look at alternative stopping criteria.

1.5 Convergence of an iterative method

We have discussed the construction of **iterations** which aim to find the solution of the equations $A\vec{x} = \vec{b}$ through a sequence of better and better approximations $\vec{x}^{(k)}$.

In general the iteration takes the form

$$\vec{x}^{(k+1)} = \vec{F}(\vec{x}^{(k)})$$

here $\vec{x}^{(k)}$ is a vector of values and $\vec{F}$ is some vector-valued function which we have defined.

How can we decide if this iteration has converged? We need $\vec{x} - \vec{x}^{(k)}$ to be small, but we don't have access to the exact solution $\vec{x}$ so we have to do something else!

How do we decide that a vector/array is small? The most common measure is to use the "Euclidean norm" of an array (which you met last year!). This is defined to be the square root of the sum of squares of the entries of the array:

$$\|\vec{r}\| = \sqrt{\sum_{i=1}^{n} r_i^2}.$$

where $\vec{r}$ is a vector with n entries.

Examples

Consider the following sequence $\vec{x}^{(k)}$:

$$\begin{pmatrix}1\\-1\end{pmatrix}, \begin{pmatrix}1.5\\0.5\end{pmatrix}, \begin{pmatrix}1.75\\0.25\end{pmatrix}, \begin{pmatrix}1.875\\0.125\end{pmatrix}, \begin{pmatrix}1.9375\\-0.0625\end{pmatrix}, \begin{pmatrix}1.96875\\-0.03125\end{pmatrix}, \ldots$$

- What is $\|\vec{x}^{(1)} - \vec{x}^{(0)}\|$?
- What is $\|\vec{x}^{(5)} - \vec{x}^{(4)}\|$?

Let $\vec{x} = \begin{pmatrix}2\\0\end{pmatrix}$.

- What is $\|\vec{x} - \vec{x}^{(3)}\|$?
- What is $\|\vec{x} - \vec{x}^{(4)}\|$?
- What is $\|\vec{x} - \vec{x}^{(5)}\|$?

Rather than decide in advance how many iterations (of the Jacobi or Gauss-Seidel methods) to use stopping criteria:

- This could be a maximum number of iterations.
- This could be the *change* in values is small enough:

$$\|x^{(k+1)} - \vec{x}^{(k)}\| < tol,$$

- This could be the *norm of the residual* is small enough:

$$\|\vec{r}\| = \|\vec{b} - A\vec{x}^{(k)}\| < tol$$

In both cases, we call *tol* the **convergence tolerance** and the choice of *tol* will control the accuracy of the solution.

Discussion

What is a good convergence tolerance?

In general there are two possible reasons that an iteration may fail to converge.

- It may **diverge** - this means that $\|\vec{x}^{(k)}\| \to \infty$ as k (the number of iterations) increases, e.g.:

$$\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}4\\2\end{pmatrix}, \begin{pmatrix}16\\4\end{pmatrix}, \begin{pmatrix}64\\8\end{pmatrix}, \begin{pmatrix}256\\16\end{pmatrix}, \begin{pmatrix}1024\\32\end{pmatrix}, \ldots$$

- It may *neither* converge nor diverge, e.g.:

$$\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}2\\1\end{pmatrix}, \begin{pmatrix}3\\0\end{pmatrix}, \ldots$$

In addition to testing for convergence it is also necessary to include tests for failure to converge.

- Divergence may be detected by monitoring $\|\vec{x}^{(k)}\|$.
- Impose a maximum number of iterations to ensure that the loop is not repeated forever!

1.6 Summary

Many complex computational problems simply cannot be solved with today's computers using direct methods. Iterative methods are used instead since they can massively reduce the computational cost and storage required to get a "good enough" solution.

These basic iterative methods are simple to describe and program but generally slow to converge to an accurate answer - typically $O(n)$ iterations are required! Their usefulness for general matrix systems is very limited therefore - but we have shown their value in the solution of sparse systems however.

More advanced iterative methods do exist but are beyond the scope of this module - see Final year projects, MSc projects, PhD, and beyond!

1.7 Further reading

More details on these basic (and related) methods:

- Wikipedia: Jacobi method
- Wikipedia: Gauss-Seidel method
- Wikipedia: Iterative methods
  - see also Richardson method, Damped Jacobi method, Successive over-relaxation method (SOR), Symmetric successive over-relaxation method (SSOR) and Krylov subspace methods

More details on sparse matrices:

- Wikipedia Sparse matrix - including a long detailed list of software libraries support sparse matrices.
- Stackoverflow: Using a sparse matrix vs numpy array
- Jason Brownlee: A gentle introduction to sparse matrices for machine learning, Machine learning mastery

Some related textbooks:

- Jack Dongarra Templates for the solution of linear systems: Stopping criteria
- Jack Dongarra Templates for the solution of linear systems: Stationary iterative methods
- Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3rd ed.), Baltimore: Johns Hopkins, ISBN 978-0-8018-5414-9.
- Saad, Yousef (2003). Iterative Methods for Sparse Linear Systems (2nd ed.). SIAM. p. 414. ISBN 0898715342.

Some software implementations:

- `scipy.sparse` custom routines specialised to sparse matrices
- SuiteSparse, a suite of sparse matrix algorithms, geared toward the direct solution of sparse linear systems
- `scipy.sparse` iterative solvers: Solving linear problems
- PETSc: Linear system solvers - a high performance linear algebra toolkit