4.3 Appendix - Iterative methods for linear systems

The Richardson method The Richardson iteration is the simplest kind, in which \( M = cI \), where \( c > 0 \). The iteration is

\[
x^{(k+1)} = x^{(k)} + \frac{1}{c} (b - Ax^{(k)}).
\]

The (weighted) Jacobi method The next simplest iteration is the Jacobi method, in which \( M_j = D = \text{diag}(A) \)—a diagonal matrix whose entries are the diagonal entries of \( A \) \((D_{ii} = A_{ii})\). The weighted iteration is

\[
x^{(k+1)} = x^{(k)} + \omega D^{-1} (b - Ax^{(k)}),
\]

where usually \( 0 < \omega \leq 1 \) (\( \omega \) is also called a damping parameter). If \( \omega = 1 \), then we get the standard Jacobi method.

Another way to look at the Jacobi method is by the following splitting: \( A = D + L + U \), where \( D \) is the diagonal part of \( A \), \( L \) is the lower triangular part of \( A \) and \( U \) is the upper triangular part. Then, the weighted Jacobi method is

\[
x^{(k+1)} = (1 - \omega) x^{(k)} + \omega D^{-1} (b - (L + U) x^{(k)}),
\]

and is equivalent to the previous writing. Now, however, the iteration does not include the standard matrix-vector multiplication \( Ax \), but has a different matrix-vector product. A third way to write the Jacobi method is the scalar one:

\[
x^{(k+1)}_i = (1 - \omega)x^{(k)}_i + \omega \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x^{(k)}_j \right), \quad i = 1, ..., n.
\]

The Successive Over Relaxation (SOR) method The SOR method is a weighted version of GS (just like Jacobi and its weighted version), only here, we may choose a parameter \( \omega > 1 \). We start with the scalar form:

\[
x^{(k+1)}_i = (1 - \omega)x^{(k)}_i + \omega \frac{1}{a_{ii}} \left( b_i - \sum_{j < i} a_{ij} x^{(k+1)}_j - \sum_{j > i} a_{ij} x^{(k)}_j \right), \quad i = 1, ..., n.
\]
In vector form we get

\[ x^{(k+1)} = (D + \omega L)^{-1} \left((1 - \omega)Dx^{(k)} + \omega(b - Ux^{(k)})\right) , \]

or, more compactly

\[ x^{(k+1)} = x^k + \omega(D + \omega L)^{-1}(b - Ax^{(k)}) , \]

**Theorem 14.** If the matrix \( A \) is positive definite, then SOR converges for \( 0 < \omega < 2 \).

### 4.3.1 The variational meaning of GS

Assume that \( A \) is positive definite. Then we can show that the updates of Gauss-Seidel for each \( x_i \) are equivalent to minimizing the following function

\[ f(x) = \frac{1}{2}\|x - x^*\|_A^2 = \frac{1}{2}x^T Ax - x^T b + \frac{1}{2}(x^*)^T b, \]

with respect to \( x_i \). This function is equivalent to solving a linear system \( Ax = b \), and has a minimum of 0, which is obtained at \( x^* = A^{-1}b \). To show this, consider the necessary condition for a minimum point: \( \nabla f = 0 \). In our case it is simply the linear system \( Ax = b \). Note that the term \( \frac{1}{2}(x^*)^T b \) is just a constant scalar that does not play part in the minimization.

As noted before, in the Gauss-Seidel method we iterate over all entries \( i \) of \( x \), and change each scalar entry \( x_i \) so that the \( i \)-th equation is satisfied (or \( r_i = 0 \) for the residual vector \( r \)) given the other entries in the current approximation \( x \). By fulfilling the \( i \)-th equation in \( Ax = b \), we essentially zeroing the \( i \)-th entry of the gradient, which means we set \( \frac{\partial f}{\partial r_i} = 0 \). It means that we also minimize \( f() \) with respect to \( x_i \) at each update of \( x_i \). This is the variational property of Gauss-Seidel. From this we can learn that the value of \( f \) is monotonically decreasing with each update and since \( f \) is bounded from below, the series \( \{f(x)\} \) converges. It can be shown that if the matrix \( A \) is non-singular, the Gauss-Seidel method converges.

**Theorem 15.** If the matrix \( A \) is positive definite, then Gauss Seidel converges, and the values \( f(x^{(k)}) \) of the function \( f \) above are monotonically decreasing with the GS updates.
Example 9 (Variational property of Gauss Seidel). Consider the following linear system:

\[ A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 4 \end{bmatrix}. \]

It is easy to show that

\[ f(x) = \frac{1}{2} x^\top A x - x^\top b \]

\[ = x_1^2 + x_1 x_2 + 1.5 x_2^2 - 3 x_1 - 4 x_2. \]  

The condition \( \nabla f = 0 \) in this case is

\[ \frac{\partial f}{\partial x_1} = 2 x_1 + x_2 - 3 = 0 \]  

\[ \frac{\partial f}{\partial x_2} = x_1 + 3 x_2 - 4 = 0 \]

which is exactly the linear system \( A x = b \). We now show that zeroing each of the equations (32)-(33) alternately leads to a monotonically decreasing series \( f \), and that this process is equivalent to a general Gauss-Seidel routine. The code in this example plots the graph in Fig. 2.
4. Iterative methods for linear systems

4.3.2 One-point non-stationary iterative methods (Steepest Descent)

The methods presented earlier are called “stationary”. It means that the iteration function \( \phi \) that defines the method \( x^{(k+1)} = \phi(x^{(k)}) \) do not change with \( k \). To illustrate the meaning of “non-stationary” methods, consider, for example, the weighted Jacobi or Richardson
iterations mentioned earlier. These methods need a parameter (c or ω) to be chosen in some optimal way. The next example illustrate an approach that automatically chooses the parameter in each iteration.

**Steepest Descent for symmetric positive definite matrices** Assume again that $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite and consider the minimization of the following quadratic function

$$f(x) = \frac{1}{2} \|x^* - x\|^2_A = \frac{1}{2} x^T A x - x^T b + \frac{1}{2} (x^*)^T b.$$  

(35)

Recall that this function has a minimum of 0, obtained at $x = x^* = A^{-1} b$, and the condition $\nabla f = 0$ is the linear system $A x = b$. The steepest descent method is the most basic optimization method for minimizing a general function $f$ and is defined by

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)}) = x^{(k)} + \alpha (b - A x^{(k)}),$$

where $\alpha > 0$ is a parameter. The first equality is for a general $f$, and the second equality is for Eq. (35). This method is exactly the Richardson method with $\alpha$ instead of a constant $\frac{1}{c}$. The question is: can we choose $\alpha$ is some optimal way? Well, the optimal $\alpha$ is the value such that $\rho(I - \alpha A)$ is minimal, but this is very hard to choose. As an alternative, we can choose $\alpha$ to be optimal for each iteration in a greedy sense so we minimize (35). We define
the following scalar function \( g(\alpha) \):

\[
\begin{align*}
g(\alpha) & \triangleq f(x^{(k)} + \alpha r^{(k)}) = \frac{1}{2} \|x^* - x^{(k)} - \alpha r^{(k)}\|^2_A = \frac{1}{2} \|e^{(k)} - \alpha r^{(k)}\|^2_A \\
& = \frac{1}{2} ((e^{(k)})^\top A e^{(k)}) - \alpha (r^{(k)})^\top A e^{(k)} + \frac{1}{2} \alpha^2 ((r^{(k)})^\top A r^{(k)})
\end{align*}
\]

(36)

And the minimization of \( g \) with respect to \( \alpha \) is done by:

\[
\begin{align*}
g'(\alpha) &= -(r^{(k)})^\top A e^{(k)} + \alpha ((r^{(k)})^\top A r^{(k)}) = 0 \\
\Rightarrow \alpha_{opt} &= \frac{(r^{(k)})^\top A e^{(k)}}{(r^{(k)})^\top A r^{(k)}} = \frac{\langle r^{(k)}, e^{(k)} \rangle}{\langle r^{(k)}, r^{(k)} \rangle}.
\end{align*}
\]

(37)

Which leads to the choice of \( \alpha_{opt} \). This choice of \( \alpha \) is a good choice although it is not really optimal for the whole convergence process.

**Algorithm: The steepest descent method (for linear systems)**

\[
\begin{align*}
&\text{# Input: } A \in \mathbb{R}^{n \times n} \text{ SPD, } b \in \mathbb{R}^n, \quad x^{(0)} \in \mathbb{R}^n, \\
&\quad \text{maxIter, } \epsilon, \text{ Convergence criterion} \\
&\text{# Output: } x \text{ s.t } Ax \approx b \\
&\text{Define the first residual } r^{(0)} = b - Ax^{(0)} \\
&\text{for } k = 1, ..., \text{maxIter} \text{ do} \\
&\quad \text{Define a weight: } \alpha = \frac{\langle r^{(k-1)}, r^{(k-1)} \rangle}{\langle r^{(k-1)}, Ar^{(k-1)} \rangle} \\
&\quad x^{(k)} = x^{(k-1)} + \alpha r^{(k-1)}, \\
&\quad r^{(k)} = b - Ax^{(k)} = r^{(k-1)} - \alpha Ar^{(k-1)} \quad \# Ar^{(k-1)} \text{ is already} \\
&\quad \text{computed for calculating } \alpha. \\
&\quad \text{If convergence is reached, break} \\
&\text{end} \\
&\text{Return } x^{(k)} \text{ as the solution.}
\end{align*}
\]

**Algorithm 5:** The steepest descent method for linear systems.

We now examine the process. Consider the iteration with \( \alpha_{opt} \) using the inner products notation:

\[
x^{(k+1)} = x^{(k)} + \frac{\langle r^{(k)}, A e^{(k)} \rangle}{\langle r^{(k)}, A r^{(k)} \rangle} r^{(k)}
\]

Taking the minus of this equation and adding \( x^* \) we get an equation for the error

\[
e^{(k+1)} = e^{(k)} - \frac{\langle r^{(k)}, A e^{(k)} \rangle}{\langle r^{(k)}, A r^{(k)} \rangle} r^{(k)}.
\]

(38)
Notice that such equations appear in the Gram Schmidt orthogonalization process, and here, we’re making $e^{k+1} A$-orthogonal to the direction $r^{(k)}$. Although this looks appealing, it has a somewhat unwanted property: it also means that the residuals are orthogonal $\langle r^{(k+1)}, r^{(k)} \rangle = 0$, and since $x^{(k+1)} - x^{(k)} = \alpha r^{(k)}$, it also means that the directions are orthogonal to each other $\langle x^{(k+1)} - x^{(k)}, x^{(k)} - x^{(k-1)} \rangle = 0$, or that the algorithm is zigzagging as shown in the Fig. 3. The code is for the plot is below.
4.3.3 Krylov methods

Let us examine two steps of the the Steepest descent (SD) method:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{r}^{(k)} , \quad \text{and} \quad \mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \alpha^{(k-1)} \mathbf{r}^{(k-1)} .$$

Putting the two together leads to

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k-1)} + \alpha^{(k)} \mathbf{r}^{(k)} + \alpha^{(k-1)} \mathbf{r}^{(k-1)} .$$

This way recursively we can show that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \sum_{i=0}^{k} \alpha^{(i)} \mathbf{r}^{(i)} .$$

This means that

$$\mathbf{e}^{(k+1)} \in \mathbf{e}^{(0)} + \text{span}\{\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, ..., \mathbf{r}^{(k)}\} = \mathbf{e}^{(0)} + \text{span}\{\mathbf{r}^{(0)}, A\mathbf{r}^{(0)}, ..., A^k \mathbf{r}^{(0)}\} .$$

SD chooses the coefficients $\alpha^{(k)}$ in a certain way, but can we do better with this span? It turns out we can!

We will now generalized the SD method to be of the form $\mathbf{x}^{(k+1)} = \phi(\mathbf{x}^{(k)}, \mathbf{x}^{(k-1)}, ..., \mathbf{x}^{(0)})$, instead of being a one-point method that depends only on $\mathbf{x}^{(k)}$. The new family of methods
are defined by
\[ x^{(k+1)} = x^{(k)} + \sum_{i=0}^{k} \alpha_i^{(k)} r^{(i)}, \]  
(40)
and this time all the \( k+1 \) coefficients \( \{\alpha_i^{(k)}\}_{i=0}^{k} \) are chosen in some optimal way for each step. Note that (39) holds for this as well, and in other words, \( e^{(k+1)} \) is chosen to be out of the span in (39) such that it is optimal in some way. This will introduce a family of very popular solvers called **Krylov Methods**, and the subspace defined in (39) is called a Krylov subspace. By choosing some coefficients for each step, (39) is the same as:
\[ e^{(k+1)} = e^{(0)} + p_k(A) A e^{(0)} = (I - p_k(A) A) e^{(0)}, \]
where \( p_k \) is some polynomial of degree \( k \) that we need to choose in some optimal way.

### 4.3.4 Conjugate gradients (CG) and its variants

The discussion above fits any matrix \( A \), but for now we keep discussing the case of a symmetric positive definite \( A \). Let us now examine a particular way for choosing \( \alpha_i^{(k)} \), such that, similarly to SD, at step \( k \), the functional (35) is minimized over \( \text{span}\{r^{(0)}, r^{(1)}, \ldots, r^{(k)}\} \):
\[
\{\alpha_i^{(k)}\}_{i=0}^{k} = \arg \min_{\{\alpha_i\}_{i=0}^{k}} \left\{ f(x^{(k+1)}) \right\} = \arg \min_{\{\alpha_i\}_{i=0}^{k}} \left\{ f \left( x^{(k)} + \sum_{i=0}^{k} \alpha_i r^{(i)} \right) \right\}. \tag{41}
\]
This is similar to what we did in (36)-(37) for a single \( \alpha \) in SD. While it is possible to follow the derivation of (36)-(37) for this case as well, we will do that in another way.

We saw earlier that in SD, we get Eq. (38) for the error, where similarly to the Gram Schmidt orthogonalization process, we’re making \( e^{k+1} \) \( A \)-orthogonal to the direction \( r^{(k)} \).

Let us assume that we can build a set of vectors \( \{p^{(i)}\}_{i=0}^{k} \), such that
\[ \text{span} \left( \{p^{(i)}\}_{i=0}^{k} \right) = \text{span} \left( \{r^{(i)}\}_{i=0}^{k} \right), \]
and such that \( \langle p^{(j)}, A p^{(j)} \rangle = \delta_{ij} \), i.e., the vectors \( p^{(j)} \) are \( A \)-orthogonal. This can be achieved by a Gram Schmidt process, for example.

**The naive CG method**  Now we will see the first two steps of CG:
1. (SD step as initialization) Define \( p^{(0)} = r^{(0)} \). Calc: \( \alpha_0^{(0)} = \frac{\langle e^{(0)}, A p^{(0)} \rangle}{\langle p^{(0)}, A p^{(0)} \rangle} \).

Now \( x^{(1)} = x^{(0)} + \alpha_0^{(0)} p^{(0)} \). Note that \( \langle e^{(1)}, A p^{(0)} \rangle = 0 \).

Calc: \( r^{(1)} = r^{(0)} - \alpha_0^{(0)} A p^{(0)} \).

2. (CG 2nd step) Define an orthogonal direction by a GS step: \( p^{(1)} = r^{(1)} - \frac{\langle r^{(1)}, A p^{(0)} \rangle}{\langle p^{(0)}, A p^{(0)} \rangle} p^{(0)} \).

Perform a minimization:

\[
\alpha_0^{(1)}, \alpha_1^{(1)} = \arg \min_{\alpha_0, \alpha_1} g(\alpha_0, \alpha_1),
\]

where

\[
g(\alpha_0, \alpha_1) = f(x^{(1)} + \alpha_0 p^{(0)} + \alpha_1 p^{(1)}) = \frac{1}{2} \| e^{(1)} - \alpha_0 p^{(0)} - \alpha_1 p^{(1)} \|_A^2.
\]

This is again a quadratic minimization that leads (given the orthogonality of the \( p_i \)s) to

\[
\alpha_1^{(1)} = \frac{\langle e^{(1)}, A p^{(1)} \rangle}{\langle p^{(1)}, A p^{(1)} \rangle}
\]

and surprisingly \( \alpha_0^{(1)} = \frac{\langle e^{(1)}, A p^{(0)} \rangle}{\langle p^{(0)}, A p^{(0)} \rangle} = 0 \), because of Step 1. Note that on this step, we made \( e^{(2)} \) \( A \)-orthogonal to both \( p^{(1)} \) and \( p^{(0)} \).

Compute: \( x^{(2)} = x^{(1)} + \alpha_1^{(1)} p^{(1)}, r^{(2)} = r^{(1)} - \alpha_1^{(1)} A p^{(1)} \).

This means that at iteration \( k \) we should not minimize \( f \) over previous directions because of the orthogonalization. It turns out that the \( k \)-th iteration of CG can be performed using one variable \( \alpha^{(k)} \), because the rest are just zero. Assume that we have all the previous directions \( p^0, ..., p^{k-1} \) \( A \)-orthogonal, and assume that we have \( x^{(k)} \), and \( r^{(k)} \).

- First make \( p^{(k)} \) orthogonal to all previous directions by GS:

\[
p^{(k)} = r^{(k)} - \sum_{i=0}^{k-1} \frac{\langle r^{(k)}, A p^{(i)} \rangle}{\langle p^{(i)}, A p^{(i)} \rangle} p^{(i)}.
\] (42)

- Choose \( \alpha^{(k)} \) such that \( f(x^{k+1}) \) is minimized: \( \alpha^{(k)} = \frac{\langle e^{(k)}, A p^{(k)} \rangle}{\langle p^{(k)}, A p^{(k)} \rangle} \), or equivalently such that \( e^{(k+1)} \) is \( A \)-orthogonal to \( p^{(k)} \) (in addition to \( p^{(i)} \), for \( i = k - 1, ..., 0 \)).

- Compute: \( x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}, r^{(k+1)} = r^{(k)} - \alpha^{(k)} A p^{(k)} \).

The true CG method In the naive CG method we saw a nice process, which performs quite a lot of iterations in Eq. (42). For a large \( k \), this may be very expensive. In fact, this is not so different than minimizing (41) directly (it is about the same cost for \( k << n \)).
We will see now the true advantage of CG, by showing that (42) can be significantly reduced. For \( i < k - 1 \) we have that in (42)

\[
\langle r^{(k)}, Ap^{(i)} \rangle = \langle r^{(k)}, \frac{1}{\alpha^{(i)}} (r^{(i+1)} - r^{(i)}) \rangle = \langle \frac{1}{\alpha^{(i)}} (r^{(i+1)} - r^{(i)}) , Ae^{(k)} \rangle = 0,
\]

because at the conclusion of each CG step we have that \( e^{(k)} \) is \( A \)-orthogonal to all the \( p_i \)'s and the \( r_i \)'s for \( i < k \). This means that Eq. (42) has only one term in the sum, that we will denote as \( \beta \).

**Algorithm:** The Conjugate Gradient method

```plaintext
# Input: \( A \in \mathbb{R}^{n \times n} \) SPD, \( b \in \mathbb{R}^n \), \( x^{(0)} \in \mathbb{R}^n \),
maxIter, \( \epsilon \), Convergence criterion
# Output: \( x \) s.t \( Ax \approx b \)
Define the first residual \( p^{(0)} = r^{(0)} = b - Ax^{(0)} \)
for \( k = 1, ..., maxIter \) do
    Define a weight: \( \alpha = \frac{\langle r^{(k-1)}, p^{(k-1)} \rangle}{\langle p^{(k-1)}, Ap^{(k-1)} \rangle} = \frac{\langle r^{(k-1)}, p^{(k-1)} \rangle}{\langle p^{(k-1)}, Ap^{(k-1)} \rangle} \)
    \( x^{(k)} = x^{(k-1)} + \alpha p^{(k-1)} \),
    \( r^{(k)} = b - Ax^{(k)} = r^{(k-1)} - \alpha Ap^{(k-1)} \)
    If convergence is reached, break
    \( \beta = -\frac{\langle r^{(k)}, Ap^{(k-1)} \rangle}{\langle p^{(k-1)}, Ap^{(k-1)} \rangle} = -\frac{\langle r^{(k)}, r^{(k-1)} \rangle}{\langle p^{(k-1)}, r^{(k-1)} \rangle} \)
    \( p^{(k)} = r^{(k)} + \beta p^{(k-1)} \)
end
Return \( x^{(k)} \) as the solution.
```

**Algorithm 6:** The Conjugate Gradients method.

**Theorem 16.** If \( A \in \mathbb{R}^{n \times n} \), SPD and full rank, then the CG method converges to the solution of \( Ax = b \), at at most \( n \) iterations for any initial guess.

**Proof.** The proof follows immediately from (41): at the \( n \)-th iteration we will be forming a minimization over the full space \( \mathbb{R}^n \).

**Theorem 17.** If \( A \in \mathbb{R}^{n \times n} \), SPD and full rank, then the CG method produces iterates \( \{x^{(k)}\} \) which satisfy

\[
\|x^* - x^{(k)}\|_A \leq 2\|x^* - x^{(0)}\|_A \left( \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k
\]
4. Iterative methods for linear systems


While the steepest descent method converges according to the condition number of $A$, the CG method converges according to the square root of this condition number.

4.3.5 Other Krylov Methods

The Conjugate Gradient method fits only for symmetric positive definite systems. For other cases, other Krylov methods should be used. We will not cover this in detail in this course. For a symmetric but indefinite systems, the MINRES method should be used. For a non-symmetric case, the GCR or GMRES methods should be used.
4.3.6 Sparse Matrices

Iterative methods are useful when the multiplication of a matrix $A$ with a vector can be computed efficiently. One of the most common instances of such a case is sparse matrices. A matrix $A \in \mathbb{R}^{m \times n}$ is called sparse when the number of non-zeros in $A$ are much less than $mn$. Figure 4 demonstrates a non-zero structure of a sparse matrix, where the blue dots correspond to non-zero entries and the white space corresponds to zero entries.

**Common data structures for storing sparse matrices** The main advantage of sparse matrices is their compact storage. Assume the following matrix:
4. Iterative methods for linear systems

\[
A = \begin{bmatrix}
2 & 0 & 0 \\
1 & -2 & 0 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{bmatrix}
\]

1. **Coordinate list (COO)**: COO stores a list of (row, column, value) tuples. Ideally, the entries are sorted (by row index, then column index) to improve random access times.
   - rowIdx = [1, 2, 4, 2, 4]
   - colIdx = [1, 1, 2, 3]
   - values = [2, 1, -2, 1].

2. **Compressed sparse row (CSR)** In this storage format the non-zero values are sorted by rows, and we have a pointer list rowPtr to the beginning of each row. The list is of size \( m + 1 \), and row \( i \) starts at index \( \text{rowPtr}[i] \) and ends at \( \text{rowPtr}[i+1]-1 \). The matrix above is stored by:
   - rowPtr = [1, 2, 4, 4, 6]
   - colIdx = [1, 1, 2, 3]
   - values = [2, 1, -2, 1].
   - This approach is efficient for multiplying \( Ax \).

3. **Compressed sparse column (CSC)** The storage format is similar to CSR, only now the matrix is sorted by columns and not rows. That is, now we have a pointer list colPtr to the beginning of each column, of size \( n + 1 \). The matrix above is stored by:
   - colPtr = [1, 4, 5, 6]
   - rowIdx = [1, 2, 4, 2, 4]
   - values = [2, 1, -2, 1]. This approach is efficient for multiplying \( A^\top x \). This is the storage type for sparse matrices in Julia and MATLAB.
4.4 Appendix - Iterative methods for eigenproblems

4.4.1 The inverse power iteration

The inverse power iteration is conceptually similar to the power method, and it allows one to find an approximate eigenvector when an approximation to a corresponding eigenvalue is already known.

The inverse power iteration method starts with an approximation \( \mu \) for the eigenvalue corresponding to the desired eigenvector and a vector \( x^{(0)} \), either a randomly selected vector or an approximation to the eigenvector. The method is described by

\[
x^{k+1} = (A - \mu I)^{-1}x^{(k)}; \quad x^{(k+1)} \leftarrow \frac{x^{(k+1)}}{\|x^{(k+1)}\|},
\]

Note that at each iteration we need to solve a system like \((A - \mu I)y = x^{(k)}\) efficiently, which gives \(y = (A - \mu I)^{-1}x^{(k)}\). The method converges similarly to the power method, but corresponding to the matrix \((A - \mu I)^{-1}\). Its eigenvalues are:

\[
Av_i = \lambda_i v_i \Rightarrow Av_i - \mu v_i = \lambda_i v_i - \mu v_i \Rightarrow (A - \mu I)^{-1}v_i = (\lambda_i - \mu)^{-1}v_i.
\]

Therefore, the rate of convergence is

\[
\frac{|\mu - \lambda_{\text{closest to } \mu}|}{|\mu - \lambda_{\text{second closest to } \mu}|}.
\]

The linear systems in this method can be either directly solved by an LU factorization, or approximately solved by an iterative algorithm (Jacobi, Gauss-Seidel etc.). The rate of convergence given above is only true if the linear systems are exactly solved.
Example 10 (Computing PageRank, continued). Assume we have the stochastic matrix

\[ B = \begin{bmatrix} 0 & 1/2 & 0 & 1/3 \\ 1/3 & 0 & 0 & 1/3 \\ 1/3 & 1/2 & 0 & 1/3 \\ 1/3 & 0 & 1 & 1/3 \end{bmatrix} \]

and we wish to compute its leading eigenvector. We start the power method from \( x = [0.25, 0.25, 0.25, 0.25]^\top \).

Now we will try the inverse power method with \( \mu = 1 - \epsilon \) (because we know that if we put \( \mu = 1 \) the matrix will be singular). We see a much faster convergence (essentially we get a very accurate result after one iteration), because of the leading eigenvalue of \( (B - \mu I)^{-1} \) is \( \frac{1}{\epsilon} \), while the rest are \( O(1) \).
using LinearAlgebra

B = [0.0 0.5 0 1/3.0 ; 1/3.0 0 0 1/3 ; 1/3 0 1 0 ; 1/3 0 1/3 0]

x = ones(4)/4.0; epsilon = 1e-2;

display(B)
display(x)

II = Matrix(1.0I,4,4);

for k=1:5
    x .= (B - II + epsilon*II)\x;
    x .= x/norm(x,1);
    println(k,":",x);
end

println("Eigenvalues of the iteration matrix:")
d = eigvals(inv(B - II + epsilon*II));
display(d)

#=

Output:
4x4 Array{Float64,2}:
 0.0 0.5 0.0 0.333333
 0.333333 0.0 0.0 0.333333
 0.333333 0.5 0.0 0.333333
 0.333333 0.0 1.0 0.0

4-element Array{Float64,1}:
 0.25
 0.25
 0.25
 0.25

1: [0.20410770621552313, 0.18127686659409992, 0.2728308396214232, 0.34178458756895375]
2: [0.20454935116626155, 0.1818223376551904, 0.2727270135110712, 0.3409012976674795]
3: [0.204542169354725, 0.18181815090127848, 0.27272727079226877, 0.3409091566129065]
4: [0.2045454545433826, 0.18181818181662815, 0.2727272727267544, 0.34090909091323474]

Eigenvalues of the iteration matrix:
4-element Array{Complex{Float64},1}:
-0.7556675062972193 + 0.0im
-0.732431784218587 - 0.13045528498173736im
-0.732431784218587 + 0.13045528498173736im
100.00000000000014 + 0.0im

=#