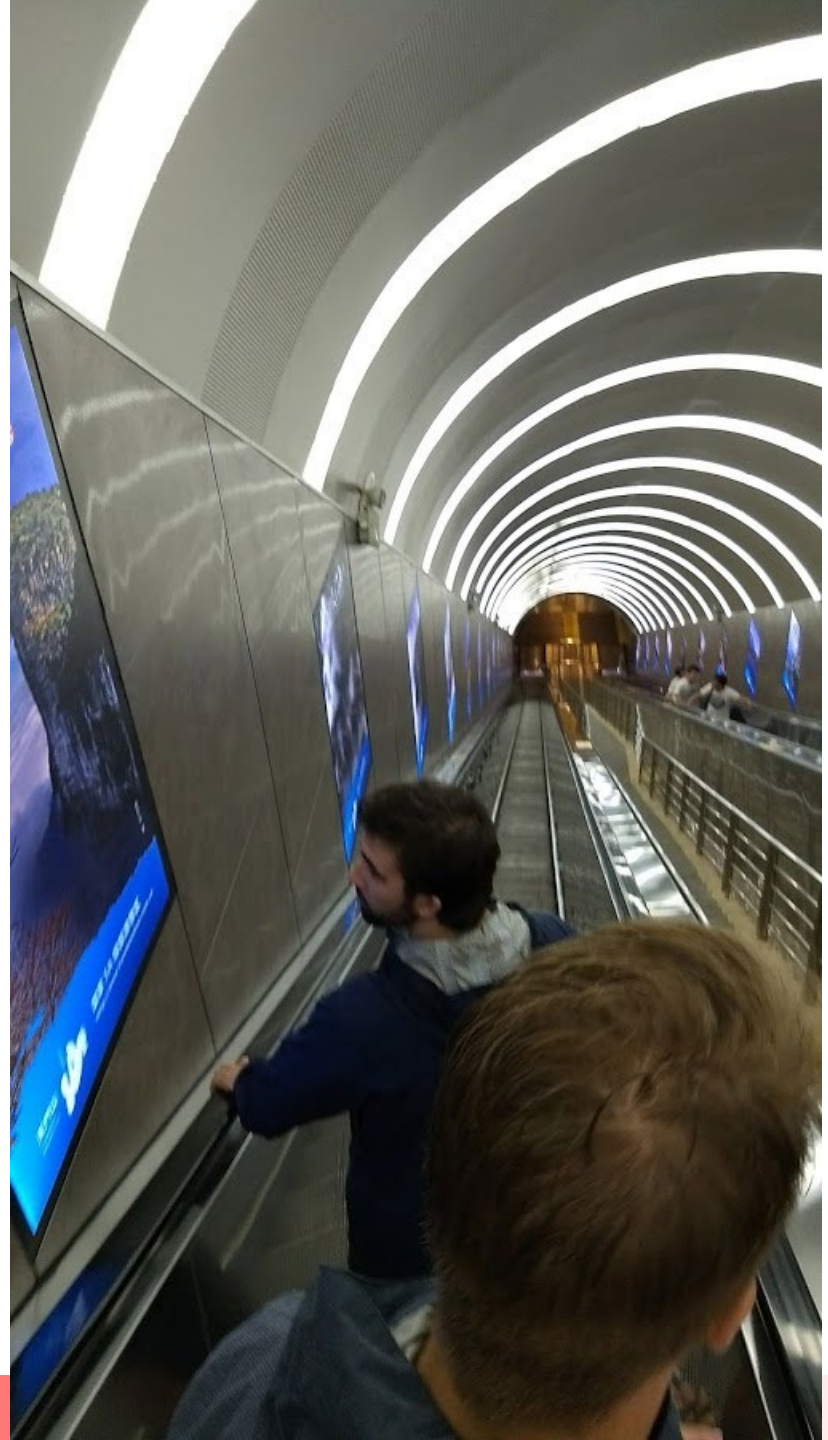
Unit 3.1 – Gauss Seidel



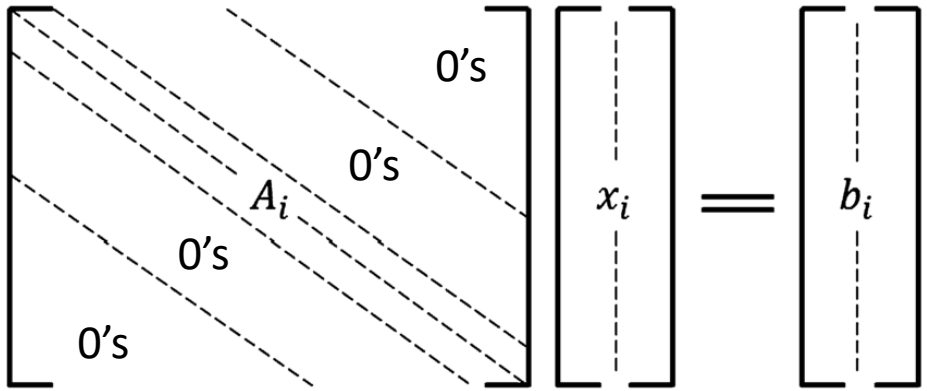
Direct Methods

- Fixed number of steps
- Guaranteed to produce a solution if one exists
- $\mathcal{O}(n^3)$ operations
- Useful for relatively small systems
- More appropriate for "full" systems

$$\begin{bmatrix} 4 & -1 & 3 & -2 & -3 & 1 \\ -3 & 7 & 2 & 2 & 3 & 8 \\ -6 & 3 & 11 & -5 & -2 & 1 \\ 2 & 9 & 13 & -3 & 7 & 1 \\ -8 & 11 & 1/4 & 7 & 6 & -5 \\ 15 & -1/2 & 3 & 5 & 3 & 11 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} -6 \\ -3 \\ 10 \\ 20 \\ 1/4 \\ 9 \end{Bmatrix}$$

Iterative Methods

- Unknown number of steps
- Approximates the solution
- May not guarantee solution
- Often $< \mathcal{O}(n^3)$ operations
- Manageable for large systems
- More appropriate for "sparse" systems

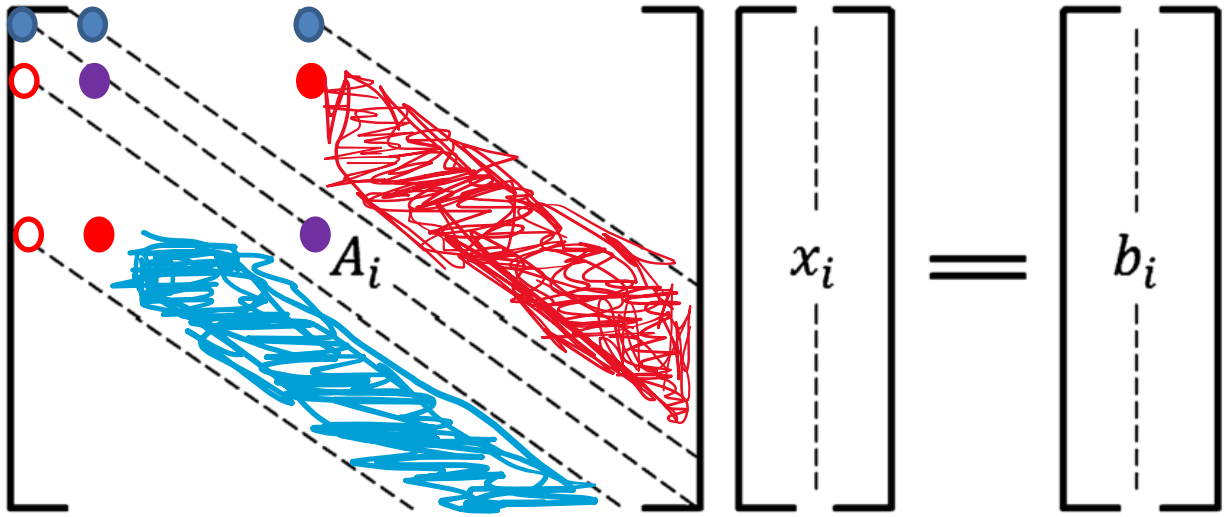


Fill-in

The difficulty with direct methods for sparse systems is the generation of nonzero entries during operations.

Memory must be available for the entire matrix.

No savings from storing only nonzero terms.



A better approach is to utilize iterative methods.

Jacobi Iteration

The basic idea of Jacobi iteration is to take the system of equations

$$\begin{bmatrix} 4 & 2 & -1 \\ 1 & 4 & 2 \\ -2 & 3 & 10 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 15 \\ 34 \end{Bmatrix}$$

And rewrite them as:

$$x_1 = \frac{1}{4}(5 - 2x_2 + x_3)$$

$$x_2 = \frac{1}{4}(15 - x_1 - 2x_3)$$

$$x_3 = \frac{1}{10}(34 + 2x_1 - 3x_2)$$

Clearly, this doesn't solve anything unless there is a guess for the solution

$$\begin{Bmatrix} x_1^0 \\ x_2^0 \\ x_3^0 \end{Bmatrix}$$

And we can write:

$$x_1^{n+1} = \frac{1}{4}(5 - 2x_2^n + x_3^n)$$

$$x_2^{n+1} = \frac{1}{4}(15 - x_1^n - 2x_3^n)$$

$$x_3^{n+1} = \frac{1}{10}(34 + 2x_1^n - 3x_2^n)$$

Gauss-Seidel Iteration

In practice, the coefficients are calculated one at a time, hence, the algorithm is slightly modified and called Gauss-Seidel (GS) iteration:

$$x_1^{n+1} = \frac{1}{4}(5 - 2x_2^n + x_3^n)$$

$$x_2^{n+1} = \frac{1}{4}(15 - x_1^{n+1} - 2x_3^n)$$

$$x_3^{n+1} = \frac{1}{10}(34 + 2x_1^{n+1} - 3x_2^{n+1})$$

Or written in a more general way:

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} A_{ij}x_j\right) \quad i = 1,2,\cdots,n \qquad (3.1\text{-}1)$$

An improvement can be made to the scheme by introducing an accelerating or retarding factor ω, so that the algorithm becomes:

$$x_i = \frac{\omega}{A_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} A_{ij}x_j\right) + (1-\omega)x_i$$

$$i = 1,2,\cdots,n \qquad (3.1\text{-}2)$$

ω is generally known as the relaxation factor. If <1 the technique is called underrelaxation and if >1 overrelaxation. An optimum value exists but needs to be estimated.

The code is *gaussSeidel* in the text.

Convergence Criteria

Since the iterative scheme only gives an estimate to the solution, we must somehow decide when the solution is "good enough."

Two primary means are used.

Small change in $\vec{x}^{n+1} - \vec{x}^{n}$ (3.1-3)

Small residual vector $\vec{r} = A\vec{x} - \vec{b}$ (3.1-4)

Note that $\vec{x}$ is a vector, so one needs to define just what small means. To do that we introduce the vector norm.

The numpy function

numpy.linalg.norm()

Accomplishes this and allows for multiple different definitions. The *2-norm* is recommended. Follow the link for more.

A note of caution related to computer arithmetic. If your problem is such that the elements of $\vec{x}$ or $\vec{r}$ are large, and thus its vector norm is large, you may run into roundoff errors.

Eq. 3.1-3 is an example of absolute error.

A better approach is to look at relative error defined by:

$$\frac{\|\vec{x}^{n+1} - \vec{x}^{n}\|}{\|\vec{x}^{n+1}\|} \qquad (3.1\text{-}5)$$

And

$$\frac{\|\vec{r}^{n+1}\|}{\|\vec{x}^{n+1}\|} = \frac{\|A\vec{x}^{n+1} - \vec{b}\|}{\|\vec{x}^{n+1}\|} \qquad (3.1\text{-}6)$$

Since these numbers will be closer to 1.

Diagonal Dominance

UNIVERSITY OF Cincinnati

An additional caveat is that solution convergence for GS can only be guaranteed if the matrix is diagonal dominant, which means:

$$|A_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |A_{ij}| \quad (\mathrm{i} = 1,2,\cdots,n)$$

(3.1-7)

Of course, non- or near-diagonal dominant matrix systems might still converge, but it is only guaranteed if Eq. 3.1-7 holds.

Unit 3.2 – Conjugate Gradient Method

UNIVERSITY OF Cincinnati

In this section we introduce the idea of optimization, which will be found in many places throughout this course. The thinking is to treat the system of equations as a function in $\vec{x}$ and seeking the minimum of that function.

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^T\vec{x} \qquad (3.2\text{-}1)$$

where:

$$\vec{x}^T = \{x_1, x_2, \cdots, x_n\}$$

$$\vec{x} = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix}$$

So that we get a scalar when we take $\vec{x}^T\vec{x}$

Hence, $f(\vec{x})$ is a scalar that can be minimized by zeroing its gradient.

$$\nabla f(\vec{x}) = A\vec{x} - \vec{b} = 0 \qquad (3.2\text{-}2)$$

The equivalent of solving our system.

The basic idea then is to choose a direction and minimize Eq. 3.2-1 in that direction. Of course, that is not the final optimal solution, so we then choose another direction that is orthogonal to it, a so-called conjugate direction. We can do this for each "direction" in the system until we get to the *n*-th equation, at which point an optimum is reached.

The algorithm uses a guess $\vec{x}_k$ and optimizes in a search direction $\vec{s}_k$.

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{s}_k \qquad (3.2\text{-}3)$$

To find α we insert 3.2-3 into our system.

$$A(\vec{x}_k + \alpha_k \vec{s}_k) = \vec{b} \qquad (3.2\text{-}4)$$

$$A\alpha_k \vec{s}_k = \vec{b} - A\vec{x}_k = \vec{r}_k \qquad (3.2\text{-}5)$$

$$\vec{s}_k{}^T A\alpha_k \vec{s}_k = \vec{s}_k{}^T \vec{r}_k \qquad (3.2\text{-}6)$$

$$\alpha_k = \frac{\vec{s}_k{}^T \vec{r}_k}{\vec{s}_k{}^T A\vec{s}_k} \qquad (3.2\text{-}7)$$

$\vec{s}_k$ starts off as the direction of steepest descent, i.e., ∇f., followed by a set of conjugate gradients, directions orthogonal to the first.

Conjugate Gradient (CG) Algorithm

input $\vec{x}_0$

$\vec{r}_0 = \vec{b} - A\vec{x}_0$

$\vec{s}_0 = \vec{r}_0$

for k in range(n)

$$\alpha_k = \frac{\vec{s}_k^T \vec{r}_k}{\vec{s}_k^T A\vec{s}_k}$$

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{s}_k$$

$$\vec{r}_{k+1} = \vec{b} - A\vec{x}_{k+1}$$

check $\|\vec{r}_{k+1}\| < \epsilon$ exit if yes

$$\beta_k = \frac{\vec{r}_{k+1}^T A\vec{s}_k}{\vec{s}_k^T A\vec{s}_k}$$

$$\vec{s}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{s}_k$$

conjGrad is the code in the text. To run you need to supply a function *Av(v).*

Limitations of Conjugate Gradient

The major limitation of the CG algorithm is that it applies formally only to symmetric, positive definite matrices.

Symmetric matrix has been defined in Section 2.4.

Positive definite has a much more esoteric mathematical description, suffice it to say that we would really prefer an algorithm that is not restricted to Symmetric Positive Definite matrices.

The bi-conjugate gradient method is one such approach, but it is unstable. A stabilized variant is bi-conjugate gradient stabilized (BiCGSTAB), however, the text does not include a code.

BiCGSTAB Algorithm

Unpreconditioned BiCGSTAB [edit]

To solve a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, BiCGSTAB starts with an initial guess $\boldsymbol{x}_0$ and proceeds as follows:

1. $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
2. Choose an arbitrary vector $\hat{\boldsymbol{r}}_0$ such that $(\hat{\boldsymbol{r}}_0, \boldsymbol{r}_0) \neq 0$, e.g., $\hat{\boldsymbol{r}}_0 = \boldsymbol{r}_0$. $(\boldsymbol{x},\boldsymbol{y})$ denotes the dot product of vectors $(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{y}$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $\boldsymbol{v}_0 = \boldsymbol{p}_0 = \boldsymbol{0}$
5. For $i = 1, 2, 3, \ldots$
    1. $\rho_i = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_{i-1})$
    2. $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
    3. $\boldsymbol{p}_i = \boldsymbol{r}_{i-1} + \beta(\boldsymbol{p}_{i-1} - \omega_{i-1}\boldsymbol{v}_{i-1})$
    4. $\boldsymbol{v}_i = \boldsymbol{A}\boldsymbol{p}_i$
    5. $\alpha = \rho_i/(\hat{\boldsymbol{r}}_0, \boldsymbol{v}_i)$
    6. $\boldsymbol{h} = \boldsymbol{x}_{i-1} + \alpha\boldsymbol{p}_i$
    7. If $\boldsymbol{h}$ is accurate enough, then set $\boldsymbol{x}_i = \boldsymbol{h}$ and quit
    8. $\boldsymbol{s} = \boldsymbol{r}_{i-1} - \alpha\boldsymbol{v}_i$
    9. $\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s}$
    10. $\omega_i = (\boldsymbol{t}, \boldsymbol{s})/(\boldsymbol{t}, \boldsymbol{t})$
    11. $\boldsymbol{x}_i = \boldsymbol{h} + \omega_i\boldsymbol{s}$
    12. If $\boldsymbol{x}_i$ is accurate enough, then quit
    13. $\boldsymbol{r}_i = \boldsymbol{s} - \omega_i\boldsymbol{t}$

BiCGSTAB is a member of a group of techniques called Krylov methods. Several methods can be found at pyamg.

Preconditioning

With many of these algorithms you will find them mentioning preconditioners. Mathematically speaking, a preconditioner is anything that changes our system such that the condition number of the matrix is closer to 1. So, for our system $A\vec{x} = \vec{b}$

$$PA\vec{x} = P\vec{b} \qquad (3.2\text{-}8)$$

Then apply the methods to $\acute{A}\vec{x} = \acute{\vec{b}}$, where $\acute{A} = PA$ and $\acute{\vec{b}} = P\vec{b}$.

Of course, the ultimate preconditioner is $P = A^{-1}$ since $A^{-1}A = I$ and $\mathrm{cond}(I) = 1$. So, a good preconditioner is one that is a good approximation to A^{-1}.

A successful possibility is *ILU(0)*, incomplete LU decomposition. The basic idea is quite simple, do *LU* decomposition, but don't change the entry if it is zero. This is particularly appropriate for sparse systems.