WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Español
فارسی
Français
한국어
हिन्दी
Bahasa Indonesia
Italiano
日本語
■■■■■
Polski
Português
Русский
Српски / srpski
Українська
Tiếng Việt
中文
Edit links

Gauss–Seidel method

From Wikipedia, the free encyclopedia

In numerical linear algebra, the **Gauss–Seidel method**, also known as the **Liebmann method** or the **method of successive displacement**, is an iterative method used to solve a linear system of equations. It is named after the German mathematicians Carl Friedrich Gauss and Philipp Ludwig von Seidel, and is similar to the Jacobi method. Though it can be applied to any matrix with non-zero elements on the diagonals, convergence is only guaranteed if the matrix is either diagonally dominant, or symmetric and positive definite. It was only mentioned in a private letter from Gauss to his student Gerling in 1823.[1] A publication was not delivered before 1874 by Seidel.

Contents [hide]

Description [edit]

The Gauss–Seidel method is an iterative technique for solving a square system of *n* linear equations with unknown **x**:

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

It is defined by the iteration

$$L_*\mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}^{(k)},$$

where $\mathbf{x}^{(k)}$ is the *k*th approximation or iteration of $\mathbf{x}$, $\mathbf{x}^{k+1}$ is the next or *k* + 1 iteration of $\mathbf{x}$, and the matrix *A* is decomposed into a lower triangular component L_*, and a strictly upper triangular component *U*: $A = L_* + U.$[2]

In more detail, write out *A*, **x** and **b** in their components:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Then the decomposition of *A* into its lower triangular component and its strictly upper triangular component is given by:

$$A = L_* + U \qquad \text{where} \qquad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$L_* \mathbf{x} = \mathbf{b} - U\mathbf{x}$$

The Gauss–Seidel method now solves the left hand side of this expression for **x**, using previous value for **x** on the right hand side. Analytically, this may be written as:

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$$

However, by taking advantage of the triangular form of L_*, the elements of $\mathbf{x}^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i,j = 1,2,\ldots,n.$$ [3]

The procedure is generally continued until the changes made by an iteration are below some tolerance, such as a sufficiently small residual.

Discussion [edit]

The element-wise formula for the Gauss–Seidel method is extremely similar to that of the Jacobi method.

The computation of $x_i^{(k+1)}$ uses only the elements of $\mathbf{x}^{(k+1)}$ that have already been computed, and only the elements of $\mathbf{x}^{(k)}$ that have not yet to be advanced to iteration k+1. This means that, unlike the Jacobi method, only one storage vector is required as elements can be overwritten as they are computed, which can be advantageous for very large problems.

However, unlike the Jacobi method, the computations for each element cannot be done in parallel. Furthermore, the values at each iteration are dependent on the order of the original equations.

Gauss-Seidel is the same as SOR (successive over-relaxation) with $\omega = 1$.

Convergence [edit]

The convergence properties of the Gauss–Seidel method are dependent on the matrix *A*. Namely, the procedure is known to converge if either:

- *A* is symmetric positive-definite,[4] or
- *A* is strictly or irreducibly diagonally dominant.

The Gauss–Seidel method sometimes converges even if these conditions are not satisfied.

Algorithm [edit]

Since elements can be overwritten as they are computed in this algorithm, only one storage vector is needed, and vector indexing is omitted. The algorithm goes as follows:

```
Inputs: A, b
Output: $\phi$

Choose an initial guess $\phi$ to the solution
repeat until convergence
    for i from 1 until n do
        $\sigma \leftarrow 0$
        for j from 1 until n do
            if j ≠ i then
                $\sigma \leftarrow \sigma + a_{ij}\phi_j$
            end if
        end (j-loop)
        $\phi_i \leftarrow \frac{1}{a_{ii}}(b_i - \sigma)$
    end (i-loop)
    check if convergence is reached
end (repeat)
```

Examples [edit]

An example for the matrix version [edit]

A linear system shown as $A\mathbf{x} = \mathbf{b}$ is given by:

$$A = \begin{bmatrix} 16 & 3 \\ 7 & -11 \end{bmatrix} \text{ and } b = \begin{bmatrix} 11 \\ 13 \end{bmatrix}.$$

We want to use the equation

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)})$$

in the form

$$\mathbf{x}^{(k+1)} = T\mathbf{x}^{(k)} + C$$

where:

$$T = -L_*^{-1}U \text{ and } C = L_*^{-1}\mathbf{b}.$$

We must decompose A into the sum of a lower triangular component L_* and a strict upper triangular component U:

$$L_* = \begin{bmatrix} 16 & 0 \\ 7 & -11 \end{bmatrix} \text{ and } U = \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix}.$$

The inverse of L_* is:

$$L_*^{-1} = \begin{bmatrix} 16 & 0 \\ 7 & -11 \end{bmatrix}^{-1} = \begin{bmatrix} 0.0625 & 0.0000 \\ 0.0398 & -0.0909 \end{bmatrix}.$$

Now we can find:

$$T = -\begin{bmatrix} 0.0625 & 0.0000 \\ 0.0398 & -0.0909 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix},$$

$$C = \begin{bmatrix} 0.0625 & 0.0000 \\ 0.0398 & -0.0909 \end{bmatrix} \times \begin{bmatrix} 11 \\ 13 \end{bmatrix} = \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix}.$$

Now we have T and C and we can use them to obtain the vectors $\mathbf{x}$ iteratively.

First of all, we have to choose $\mathbf{x}^{(0)}$: we can only guess. The better the guess, the quicker the algorithm will perform.

We suppose:

$$x^{(0)} = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}.$$

We can then calculate:

$$x^{(1)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.5000 \\ -0.8636 \end{bmatrix}.$$

$$x^{(2)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.5000 \\ -0.8636 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8494 \\ -0.6413 \end{bmatrix}.$$

$$x^{(3)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8494 \\ -0.6413 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8077 \\ -0.6678 \end{bmatrix}.$$

$$x^{(4)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8077 \\ -0.6678 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8127 \\ -0.6646 \end{bmatrix}.$$

$$x^{(5)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8127 \\ -0.6646 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8121 \\ -0.6650 \end{bmatrix}.$$

$$x^{(6)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8121 \\ -0.6650 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix}.$$

$$x^{(7)} = \begin{bmatrix} 0.000 & -0.1875 \\ 0.000 & -0.1193 \end{bmatrix} \times \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix} + \begin{bmatrix} 0.6875 \\ -0.7443 \end{bmatrix} = \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix}.$$

As expected, the algorithm converges to the exact solution:

$$\mathbf{x} = A^{-1}\mathbf{b} = \begin{bmatrix} 0.8122 \\ -0.6650 \end{bmatrix}.$$

In fact, the matrix A is strictly diagonally dominant (but not positive definite).

Another example for the matrix version [edit]

Another linear system shown as $A\mathbf{x} = \mathbf{b}$ is given by:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 7 \end{bmatrix} \text{ and } b = \begin{bmatrix} 11 \\ 13 \end{bmatrix}.$$

We want to use the equation

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)})$$

in the form

$$\mathbf{x}^{(k+1)} = T\mathbf{x}^{(k)} + C$$

where:

$$T = -L_*^{-1}U \text{ and } C = L_*^{-1}\mathbf{b}.$$

We must decompose A into the sum of a lower triangular component L_* and a strict upper triangular component U:

$$L_* = \begin{bmatrix} 2 & 0 \\ 5 & 7 \end{bmatrix} \text{ and } U = \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix}.$$

The inverse of L_* is:

$$L_*^{-1} = \begin{bmatrix} 2 & 0 \\ 5 & 7 \end{bmatrix}^{-1} = \begin{bmatrix} 0.500 & 0.000 \\ -0.357 & 0.143 \end{bmatrix}.$$

Now we can find:

$$T = -\begin{bmatrix} 0.500 & 0.000 \\ -0.357 & 0.143 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.000 & -1.500 \\ 0.000 & 1.071 \end{bmatrix},$$

$$C = \begin{bmatrix} 0.500 & 0.000 \\ -0.357 & 0.143 \end{bmatrix} \times \begin{bmatrix} 11 \\ 13 \end{bmatrix} = \begin{bmatrix} 5.500 \\ -2.071 \end{bmatrix}.$$

Now we have T and C and we can use them to obtain the vectors $\mathbf{x}$ iteratively.

First of all, we have to choose $\mathbf{x}^{(0)}$: we can only guess. The better the guess, the quicker will perform the algorithm.

We suppose:

$$x^{(0)} = \begin{bmatrix} 1.1 \\ 2.3 \end{bmatrix}.$$

We can then calculate:

$$x^{(1)} = \begin{bmatrix} 0 & -1.500 \\ 0 & 1.071 \end{bmatrix} \times \begin{bmatrix} 1.1 \\ 2.3 \end{bmatrix} + \begin{bmatrix} 5.500 \\ -2.071 \end{bmatrix} = \begin{bmatrix} 2.050 \\ 0.393 \end{bmatrix}.$$

$$x^{(2)} = \begin{bmatrix} 0 & -1.500 \\ 0 & 1.071 \end{bmatrix} \times \begin{bmatrix} 2.050 \\ 0.393 \end{bmatrix} + \begin{bmatrix} 5.500 \\ -2.071 \end{bmatrix} = \begin{bmatrix} 4.911 \\ -1.651 \end{bmatrix}.$$

$$x^{(3)} = \cdots.$$

If we test for convergence we'll find that the algorithm diverges. In fact, the matrix A is neither diagonally dominant nor positive definite. Then, convergence to the exact solution

$$\mathbf{x} = A^{-1}\mathbf{b} = \begin{bmatrix} -38 \\ 29 \end{bmatrix}$$

is not guaranteed and, in this case, will not occur.

An example for the equation version [edit]

Suppose given *k* equations where *x*$_n$ are vectors of these equations and starting point *x*$_0$. From the first equation solve for *x*$_1$ in terms of $x_{n+1}, x_{n+2}, \dots, x_n$. For the next equations substitute the previous values of *x*s.

To make it clear let's consider an example.

$$\begin{aligned}10x_1 - x_2 + 2x_3 &= 6,\\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25,\\ 2x_1 - x_2 + 10x_3 - x_4 &= -11,\\ 3x_2 - x_3 + 8x_4 &= 15.\end{aligned}$$

Solving for x_1, x_2, x_3 and x_4 gives:

$$\begin{aligned}x_1 &= x_2/10 - x_3/5 + 3/5,\\ x_2 &= x_1/11 + x_3/11 - 3x_4/11 + 25/11,\\ x_3 &= -x_1/5 + x_2/10 + x_4/10 - 11/10,\\ x_4 &= -3x_2/8 + x_3/8 + 15/8.\end{aligned}$$

Suppose we choose (0, 0, 0, 0) as the initial approximation, then the first approximate solution is given by

$$\begin{aligned}x_1 &= 3/5 = 0.6,\\ x_2 &= (3/5)/11 + 25/11 = 3/55 + 25/11 = 2.3272,\\ x_3 &= -(3/5)/5 + (2.3272)/10 - 11/10 = -3/25 + 0.23272 - 1.1 = -0.9873,\\ x_4 &= -3(2.3272)/8 + (-0.9873)/8 + 15/8 = 0.8789.\end{aligned}$$

Using the approximations obtained, the iterative procedure is repeated until the desired accuracy has been reached. The following are the approximated solutions after four iterations.

x_1	x_2	x_3	x_4
0.6	2.32727	−0.987273	0.878864
1.03018	2.03694	−1.01446	0.984341
1.00659	2.00356	−1.00253	0.998351
1.00086	2.0003	−1.00031	0.99985

The exact solution of the system is (1, 2, −1, 1).

An example using Python 3 and NumPy [edit]

The following numerical procedure simply iterates to produce the solution vector.

```python
import numpy as np

ITERATION_LIMIT = 1000

# initialize the matrix
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])
# initialize the RHS vector
b = np.array([6., 25., -11., 15.])

# prints the system
print("System:")
for i in range(A.shape[0]):
    row = ["{}*x{}".format(A[i, j], j + 1) for j in range(A.shape[1])]
    print(" + ".join(row), "=", b[i])
print()

x = np.zeros_like(b)
for it_count in range(ITERATION_LIMIT):
    print("Current solution:", x)
    x_new = np.zeros_like(x)

    for i in range(A.shape[0]):
        s1 = np.dot(A[i, :i], x_new[:i])
        s2 = np.dot(A[i, i + 1:], x[i + 1:])
        x_new[i] = (b[i] - s1 - s2) / A[i, i]

    if np.allclose(x, x_new, rtol=1e-8):
        break
    x = x_new

print("Solution:")
print(x)
error = np.dot(A, x) - b
print("Error:")
print(error)
```

Produces the output:

```
System:
10.0*x1 + -1.0*x2 + 2.0*x3 + 0.0*x4 = 6.0
-1.0*x1 + 11.0*x2 + -1.0*x3 + 3.0*x4 = 25.0
2.0*x1 + -1.0*x2 + 10.0*x3 + -1.0*x4 = -11.0
0.0*x1 + 3.0*x2 + -1.0*x3 + 8.0*x4 = 15.0

Current solution: [ 0.   0.   0.   0.]
Current solution: [ 0.6         2.32727273 -0.98727273  0.87886364]
Current solution: [ 1.03018182  2.03693802 -1.0144562   0.98434122]
Current solution: [ 1.00658504  2.00355502 -1.00252738  0.99835095]
Current solution: [ 1.00086098  2.00029825 -1.00030728  0.99984975]
Current solution: [ 1.00009128  2.00002134 -1.00003115  0.9999881 ]
Current solution: [ 1.00000836  2.00000117 -1.00000275  0.99999922]
Current solution: [ 1.00000067  2.00000002 -1.00000021  0.99999996]
Current solution: [ 1.00000004  1.99999999 -1.00000001  1.        ]
Current solution: [ 1.   2. -1.   1.]
Solution:
[ 1.  2. -1.  1.]
Error:
[  2.06480930e-08  -1.25551054e-08   3.61417563e-11   0.00000000e+00]
```

Program to solve arbitrary no. of equations using Matlab [edit]

The following code uses the formula

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i,j = 1,2,\ldots,n$$

```
function [x] = gauss_seidel(A, b, x0, iters)
    n = length(A);
    x = x0;
    for k = 1:iters
        for i = 1:n
            x(i) = (1/A(i, i))*(b(i) - A(i, 1:n)*x + A(i, i)*x(i));
        end
    end
end
```

See also [edit]

- Jacobi method
- Successive over-relaxation
- Iterative method. Linear systems
- Gaussian belief propagation
- Matrix splitting
- Richardson iteration

Notes [edit]

1. ^ Gauss 1903, p. 279; direct link.
2. ^ Golub & Van Loan 1996, p. 511.
3. ^ Golub & Van Loan 1996, eqn (10.1.3).
4. ^ Golub & Van Loan 1996, Thm 10.1.2.

References [edit]

- Gauss, Carl Friedrich (1903), *Werke* (in German) **9**, Göttingen: Köninglichen Gesellschaft der Wissenschaften.
- Golub, Gene H.; Van Loan, Charles F. (1996), *Matrix Computations* (3rd ed.), Baltimore: Johns Hopkins, ISBN 978-0-8018-5414-9.
- Black, Noel and Moore, Shirley, "Gauss-Seidel Method", *MathWorld*.

This article incorporates text from the article Gauss-Seidel_method on CFD-Wiki that is under the GFDL license.

External links [edit]

- Hazewinkel, Michiel, ed. (2001), "Seidel method", *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4
- Gauss–Seidel from www.math-linux.com
- Module for Gauss–Seidel Iteration
- Gauss–Seidel From Holistic Numerical Methods Institute
- Gauss Siedel Iteration from www.geocities.com
- The Gauss-Seidel Method
- Bickson
- Matlab code
- C code example

v · t · e	Numerical linear algebra [hide]
Key concepts	Floating point · Numerical stability
Problems	Matrix multiplication (algorithms) · Matrix decompositions · Linear equations · Sparse problems
Hardware	CPU cache · TLB · Cache-oblivious algorithm · SIMD · Multiprocessing
Software	BLAS · Specialized libraries · General purpose software

Categories: Numerical linear algebra | Relaxation (iterative methods)