Jacobi method

In numerical linear algebra, the **Jacobi method** (or **Jacobi iterative method**[1]) is an algorithm for determining the solutions of a diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization. The method is named after Carl Gustav Jacob Jacobi

Contents

Description

Let

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

be a square system of n linear equations, where:

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

Then A can be decomposed into a diagonal component D, and the remainder R:

$$A = D + R \qquad \text{where} \qquad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

The solution is then obtained iteratively via

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}),$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of $\mathbf{x}$ and $\mathbf{x}^{(k+1)}$ is the next or $k + 1$ iteration of $\mathbf{x}$. The element-based formula is thus:

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

The computation of $x_i^{(k+1)}$ requires each element in $\mathbf{x}^{(k)}$ except itself. Unlike the Gauss–Seidel method, we can't overwrite $x_i^{(k)}$ with $x_i^{(k+1)}$, as that value will be needed by the rest of the computation. The minimum amount of storage is two vectors of size n.

Algorithm

Input: initial guess $x^{(0)}$ to the solution, (diagonal dominant) matrix A, right-hand side vector b, convergence criterion
Output: solution when convergence is reached
Comments: pseudocode based on the element-based formula above

$k = 0$
while convergence not reached **do**
  for i := 1 **step until** n **do**
    $\sigma = 0$
    for j := 1 **step until** n **do**
      if $j \neq i$ **then**
        $\sigma = \sigma + a_{ij} x_j^{(k)}$
      end
    end
    $x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - \sigma)$
  end
  $k = k + 1$
end

Convergence

The standard convergence condition (for any iterative method) is when the spectral radius of the iteration matrix is less than 1:

$$\rho(D^{-1}R) < 1.$$

A sufficient (but not necessary) condition for the method to converge is that the matrix A is strictly or irreducibly diagonally dominant. Strict row diagonal dominance means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

The Jacobi method sometimes converges even if these conditions are not satisfied.

Example

A linear system of the form $Ax = b$ with initial estimate $x^{(0)}$ is given by

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 7 \end{bmatrix}, \ b = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \quad \text{and} \quad x^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

We use the equation $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$, described above, to estimate x. First, we rewrite the equation in a more convenient form $D^{-1}(b - Rx^{(k)}) = Tx^{(k)} + C$, where $T = -D^{-1}R$ and $C = D^{-1}b$. Note that $R = L + U$ where L and U are the strictly lower and upper parts of A. From the known values

$$D^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix}, \; L = \begin{bmatrix} 0 & 0 \\ 5 & 0 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

we determine $T = -D^{-1}(L + U)$ as

$$T = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix} \left\{ \begin{bmatrix} 0 & 0 \\ -5 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} \right\} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix}.$$

Further, C is found as

$$C = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix} \begin{bmatrix} 11 \\ 13 \end{bmatrix} = \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix}.$$

With T and C calculated, we estimate x as $x^{(1)} = Tx^{(0)} + C$:

$$x^{(1)} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix} = \begin{bmatrix} 5.0 \\ 8/7 \end{bmatrix} \approx \begin{bmatrix} 5 \\ 1.143 \end{bmatrix}.$$

The next iteration yields

$$x^{(2)} = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix} \begin{bmatrix} 5.0 \\ 8/7 \end{bmatrix} + \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix} = \begin{bmatrix} 69/14 \\ -12/7 \end{bmatrix} \approx \begin{bmatrix} 4.929 \\ -1.714 \end{bmatrix}.$$

This process is repeated until convergence (i.e., until $\|Ax^{(n)} - b\|$ is small). The solution after 25 iterations is

$$x = \begin{bmatrix} 7.111 \\ -3.222 \end{bmatrix}.$$

Another example

Suppose we are given the following linear system:

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

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

$$\begin{aligned} x_1 &= (6 + 0 - (2 * 0))/10 = 0.6, \\ x_2 &= (25 + 0 - 0 - (3 * 0))/11 = 25/11 = 2.2727, \\ x_3 &= (-11 - (2 * 0) + 0 + 0)/10 = -1.1, \\ x_4 &= (15 - (3 * 0) + 0)/8 = 1.875. \end{aligned}$$

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

x_1	x_2	x_3	x_4
0.6	2.27272	-1.1	1.875
1.04727	1.7159	-0.80522	0.88522
0.93263	2.05330	-1.0493	1.13088
1.01519	1.95369	-0.9681	0.97384
0.98899	2.0114	-1.0102	1.02135

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

An example using Python and Numpy

The following numerical procedure simply iterates to produce the solution vector

```
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[: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, atol=1e-10, rtol=0.):
        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         2.27272727 -1.1         1.875     ]
Current solution: [ 1.04727273  1.71590909 -0.80522727  0.88522727]
Current solution: [ 0.93263636  2.05330579 -1.04934091  1.13088068]
Current solution: [ 1.01519876  1.95369576 -0.96810863  0.97384272]
Current solution: [ 0.9889913   2.01141473 -1.0102859   1.02135051]
Current solution: [ 1.00319865  1.99224126 -0.99452174  0.99443374]
Current solution: [ 0.99812847  2.00230688 -1.00197223  1.00359431]
Current solution: [ 1.00062513  1.9986703  -0.99903558  0.99888839]
Current solution: [ 0.99967415  2.00044767 -1.00036916  1.00061919]
Current solution: [ 1.0001186   1.99976795 -0.99982814  0.99978598]
```

```
Current solution: [ 0.99994242  2.00008477 -1.00006833  1.0001085 ]
Current solution: [ 1.00002214  1.99995896 -0.99996916  0.99995967]
Current solution: [ 0.99998973  2.00001582 -1.00001257  1.00001924]
Current solution: [ 1.00000409  1.99999268 -0.99999444  0.9999925 ]
Current solution: [ 0.99999816  2.00000292 -1.0000023   1.00000344]
Current solution: [ 1.00000075  1.99999868 -0.99999899  0.99999862]
Current solution: [ 0.99999967  2.00000054 -1.00000042  1.00000062]
Current solution: [ 1.00000014  1.99999976 -0.99999982  0.99999975]
Current solution: [ 0.99999994  2.0000001  -1.00000008  1.00000011]
Current solution: [ 1.00000003  1.99999996 -0.99999997  0.99999995]
Current solution: [ 0.99999999  2.00000002 -1.00000001  1.00000002]
Current solution: [ 1.          1.99999999 -0.99999999  0.99999999]
Current solution: [ 1.  2. -1.  1.]
Solution:
[ 1.  2. -1.  1.]
Error:
[ -2.81440107e-08   5.15706873e-08  -3.63466359e-08   4.17092547e-08]
```

Weighted Jacobi method

The weighted Jacobi iteration uses a parameter ω to compute the iteration as

$$\mathbf{x}^{(k+1)} = \omega D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}) + (1-\omega)\,\mathbf{x}^{(k)}$$

with $\omega = 2/3$ being the usual choice.[2]

Recent developments

In 2014, a refinement of the algorithm, called *scheduled relaxation Jacobi (SRJ) method*, was published.[1][3] The new method employs a schedule of over- and under-relaxations and provides performance improvements for solving elliptic equations discretized on large two- and three-dimensional Cartesian grids. The described algorithm applies the well-known technique of polynomial (Chebyshev) acceleration to a problem with a known spectrum distribution that can be classified either as a multi-step method or a one-step method with a non-diagonal preconditioner. However, none of them are Jacobi-like methods.

Improvements published[4] in 2015.

See also

- Gauss–Seidel method
- Successive over-relaxation
- Iterative method § Linear systems
- Gaussian Belief Propagation
- Matrix splitting

References

1. Johns Hopkins University (June 30, 2014). "19th century math tactic gets a makeover—and yields answers up to 200 times faster" (http://phys.org/news/2014-06-19th-century-math-tactic-makeoverand.html). *Phys.org*. Douglas, Isle Of Man, United Kingdom: Omicron Technology Limited. Retrieved 2014-07-01.
2. Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems* (2 ed.). SIAM. p. 414. ISBN 0898715342.
3. Yang, Xiang; Mittal, Rajat (June 27, 2014). "Acceleration of the Jacobi iterative method by factors exceeding 100 using scheduled relaxation". *Journal of Computational Physics* **274**: 695–708. doi:10.1016/j.jcp.2014.06.010 (https://doi.org/10.1016%2Fj.jcp.2014.06.010)
4. Adsuara, J. E.; Cordero-Carrión, I.; Cerdá-Durán, P.; Aloy, M. A. (2015-11-11). "Scheduled Relaxation Jacobi method: improvements and applications". *Journal of Computational Physics* **321**: 369–413. arXiv:1511.04292 (https://arxiv.org/abs/1511.04292). doi:10.1016/j.jcp.2016.05.053 (https://doi.org/10.1016%2Fj.jcp.2016.05.053)

External links

- Hazewinkel, Michiel, ed. (2001) [1994], "Jacobi method", *Encyclopedia of Mathematics*, Springer Science+Business Media B.V. / Kluwer Academic Publishers, ISBN 978-1-55608-010-4
- This article incorporates text from the article Jacobi_method on CFD-Wiki that is under the GFDL license.
- Black, Noel; Moore, Shirley; and Weisstein, Eric W. "Jacobi method". *MathWorld*.
- Jacobi Method from www.math-linux.com
- Numerical matrix inversion