CS/MATH 375 - Homework 6 Solutions

Adam Fasulo

November 28, 2025

1. Equivalence of Jacobi Formulations

To show that the vector-update and component-wise versions of the Jacobi algorithm are equivalent, we can start with the vector-update formula and show how it simplifies to the component-wise version.

The vector-update formula is given as:

$$x^{(k)} = D^{-1}(C_L + C_U)x^{(k-1)} + D^{-1}b$$

Where the system matrix A is decomposed such that $A = D - C_L - C_U$. Here, D is the diagonal of A, $-C_L$ is the strictly lower-triangular part of A, and $-C_U$ is the strictly upper-triangular part of A.

We can rewrite the vector-update formula by multiplying by D:

$$Dx^{(k)} = (C_L + C_U)x^{(k-1)} + b$$

The matrix $(C_L + C_U)$ is simply A with its diagonal entries set to zero, but with the sign flipped. That is, $(C_L + C_U) = D - A$. Looking at the i-th component of the equation gives us:

$$(Dx^{(k)})_i = ((C_L + C_U)x^{(k-1)})_i + b_i$$

The left side is straightforward, as D is a diagonal matrix:

$$a_{ii}x_i^{(k)}$$

The right side involves a matrix-vector product. The i-th component is:

$$\sum_{j=1}^{n}(C_L + C_U)_{ij}x_j^{(k-1)} + b_i$$

The elements of $(C_L + C_U)$ are defined as:

$$(C_L + C_U)_{ij} = \begin{cases} -a_{ij} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

Substituting this into the summation, we get:

$$\sum_{j=1, j\neq i}^{n} (-a_{ij})x_j^{(k-1)} + b_i$$

Now, setting the left and right sides equal:

$$a_{ii}x_i^{(k)} = b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}$$

Finally, we solve for $x_i^{(k)}$ by dividing by a_{ii}:

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

This can be rewritten as:

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

This is exactly the component-wise update formula given in the problem statement. Thus, the two forms are equivalent.

2. Jacobi Algorithms

(a) Component-wise Jacobi Function

For this problem, I wrote the following MATLAB function, `my_jacobi.m`, to implement the component-wise version of the Jacobi algorithm.

```matlab
function x = my_jacobi(A, b, tot_it)
    % Implements the component-wise Jacobi method.
    n = length(b);
    x = zeros(n, 1); % Initial guess
    x_new = zeros(n, 1); % To store the updated values

    for k = 1:tot_it % Iteration loop
        for i = 1:n % Loop over each component
            summation = 0;
            for j = 1:n
                if i ~= j
                    summation = summation + A(i, j) * x(j);
                end
            end
            x_new(i) = (b(i) - summation) / A(i, i);
        end
        x = x_new; % Update x for the next iteration
    end
end
```

Listing 1: Component-wise Jacobi implementation (‘my$_jacobi.m$‘).

(b) Vector-update Jacobi Function and Verification

Here is the MATLAB function, `my_vector_jacobi.m`, which implements the vector-update version of the Jacobi algorithm.

```matlab
function x = my_vector_jacobi(A, b, tot_it)
    % Implements the vector-update Jacobi method.
    n = length(b);
    x = zeros(n, 1); % Initial guess

    % Extract matrix components
    D = diag(diag(A)); % Diagonal of A
    R = A - D;         % Remainder (equivalent to -(C_L + C_U))

    for k = 1:tot_it % Iteration loop
        % Vector update formula: x_new = D \ (b - R*x)
        x = D \ (b - R * x);
    end
end
```

Listing 2: Vector-update Jacobi implementation (‘my$_vector_jacobi.m$‘).

To verify that both versions produce the same result, I ran them with $P = 20$ and 100 iterations. The output below confirms that the final residuals are identical, and the norm of the difference between the two solution vectors is zero.

```
Residual from component-wise Jacobi: 5.515596e+00
Residual from vector-update Jacobi: 5.515596e+00
Difference between solutions: 0.000000e+00
```

(c) Speed Comparison

To compare the speed of the two implementations, I timed how long each takes to perform 100 iterations with $P = 20$.

```
Time for component-wise Jacobi: 10.4514 seconds
Time for vector-update Jacobi: 0.0027 seconds
```

The **vector-update version is significantly faster**. This speed difference is because MATLAB is highly optimized for vectorized operations. The vector-update code relies on efficient, pre-compiled routines for matrix-vector multiplication (`R*x`) and solving linear systems with a diagonal matrix (`D \ ...`). In contrast, the component-wise version uses nested ‘for‘ loops, which are interpreted by MATLAB and run much more slowly.

(d) Error Analysis

Next, I investigated how the relative error of the Jacobi solution changes as the size of the linear system increases, while keeping the number of iterations fixed at 100. The "true solution" was found using MATLAB’s backslash operator. The faster vector-update version was used for this analysis.

The table below shows the relative error, $\frac{\|x_{true}-x\|_2}{\|x_{true}\|_2}$, for various system sizes.

P	**n (P^2)**	**Relative Error**
10	100	1.592735e-02
20	400	3.236866e-01
40	1600	7.421344e-01
80	6400	9.253894e-01
160	25600	9.803876e-01

As the system size n increases, the **relative error increases**. This happens because for the Poisson problem, as the grid becomes finer (larger P and n), the condition number of the matrix A increases. This means the system becomes more sensitive and harder to solve. The spectral radius of the Jacobi iteration matrix approaches 1, leading to much slower convergence. With a fixed number of iterations (100), the method does not have enough steps to get close to the true solution for these larger, more difficult problems.

3. Comparing Algorithms

(a) Gauss-Seidel Function

The following MATLAB function, `my_gauss_seidel.m`, implements the vector-update version of the Gauss-Seidel algorithm.

```matlab
function x = my_gauss_seidel(A, b, tot_it)
    % Implements the vector-update Gauss-Seidel method.
    n = length(b);
    x = zeros(n, 1); % Initial guess

    % Extract lower triangular part of A (D - C_L)
    L = tril(A);
    % Extract strictly upper triangular part of A (-C_U)
    U = A - L;

    for k = 1:tot_it
        % Update formula: x = L \ (b - U*x)
        x = L \ (b - U * x);
    end
end
```

Listing 3: Vector-update Gauss-Seidel implementation (‘my$_{g}auss_{s}eidel.m$‘).

(b) Conjugate Gradient Function

The following MATLAB function, `my_CG.m`, implements the Conjugate Gradient algorithm.

```matlab
function x = my_CG(A, b, tot_it)
    % Implements the Conjugate Gradient method.
    n = length(b);
    x = zeros(n, 1);  % Initial guess
    r = b - A * x;    % Initial residual
    p = r;            % Initial search direction
    rs_old = r' * r;

    for k = 1:tot_it
        Ap = A * p;
        alpha = rs_old / (p' * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rs_new = r' * r;

        % Check for convergence (optional, but good practice)
        if sqrt(rs_new) < 1e-10
            break;
        end

        p = r + (rs_new / rs_old) * p;
        rs_old = rs_new;
    end
end
```

Listing 4: Conjugate Gradient implementation (‘my$_{C}G.m$‘).

(c) Performance Comparison

I compared the performance of Jacobi, Gauss-Seidel (GS), and Conjugate Gradient (CG) by examining the relative error as the number of iterations varies for a fixed system size of $P = 160$ ($n = 25600$).

The results are summarized in the table below.

Num Iterations	**Error Jacobi**	**Error GS**	**Error CG**
50	9.901226e-01	9.803920e-01	2.232156e-01
100	9.803876e-01	9.613118e-01	8.489418e-03
200	9.613036e-01	9.245329e-01	3.682238e-06
400	9.245182e-01	8.557666e-01	1.958300e-13
800	8.557416e-01	7.341945e-01	1.958300e-13
1600	7.341549e-01	5.412264e-01	1.958300e-13

Discussion of Performance:

- **Jacobi**: This algorithm performs the worst, reducing the error very slowly. After 1600 iterations, the relative error is still very high at approximately 0.73.
- **Gauss-Seidel**: GS performs better than Jacobi, with the error decreasing at a faster rate. For SPD matrices like this one, the convergence rate of GS is roughly twice that of Jacobi. However, it is still a slow method, with an error of 0.54 after 1600 iterations.
- **Conjugate Gradient**: CG is by far the superior algorithm for this problem. It reduces the error by orders of magnitude more quickly than the other two methods. It achieves a relative error close to machine precision ($\approx 10^{-13}$) in only 400 iterations. This is expected, as CG is an optimal method for solving linear systems with symmetric positive-definite (SPD) matrices.

4. Comparing Cost with Direct Solvers

(a) Non-zeros per Row

The matrix `A = gallery('poisson', P)` arises from a finite difference discretization of the Poisson equation on a 2D grid using a 5-point stencil. Each interior grid point is connected to its four neighbors (left, right, up, down). Therefore, a row in the matrix `A` corresponding to an interior point will have exactly **5 non-zero elements**: one on the main diagonal (for the point itself) and four off-diagonals (for its neighbors). Rows corresponding to boundary points may have fewer non-zeros, but the maximum is 5.

(b) Sparse Jacobi Cost

For a general dense matrix, the cost of the matrix-vector multiplication in each Jacobi iteration is $O(n^2)$. However, our matrix A is sparse. If we take advantage of this sparsity, the cost changes. Since each row has at most 5 non-zeros, the matrix-vector product `R*x` can be computed in $O(n)$ time, as we only need to perform a constant number of multiplications and additions for each row. Therefore, the cost of one Jacobi iteration for this sparse system is $\mathbf{O(n)}$.

Did my implementations take advantage of sparsity?

- `my_jacobi` (component-wise): **No.** The inner loop 'for j = 1:n' iterates over every column for each row, regardless of whether 'A(i, j)' is zero. It performs $O(n^2)$ work.
- `my_vector_jacobi` (vector-update): **Yes.** The matrix 'A' generated by 'gallery' is a sparse matrix. When we compute 'R = A - D', 'R' is also stored as a sparse matrix. MATLAB's internal routines for sparse matrix-vector multiplication are highly optimized and have a computational cost proportional to the number of non-zero elements, which is $O(n)$ for this matrix.

(c) Banded LU Factorization Cost

The matrix A is an m-banded matrix, with $m = 2P = 2\sqrt{n}$. More specifically, it is a block tridiagonal matrix where the non-zero elements are clustered around the main diagonal. The half-bandwidth (the maximum distance of a non-zero element from the diagonal) is approximately $P = \sqrt{n}$.

The computational cost for LU factorization of a banded matrix with half-bandwidth w is $O(nw^2)$. In our case, $w \approx P = \sqrt{n}$. Therefore, the cost is:

$$O(n \cdot (\sqrt{n})^2) = O(n \cdot n) = \mathbf{O(n^2)}$$

So, the cost in the requested notation is $O(n^\alpha)$ where $\alpha = \mathbf{2}$.

(d) Break-even Point

Finally, I compared the cost of the faster iterative solver (sparse Jacobi) with the fast direct solver (banded LU).

- Cost of k iterations of sparse Jacobi (from part b): $O(k \cdot n)$
- Cost of banded LU factorization (from part c): $O(n^2)$

The iterative method becomes more expensive when its total cost exceeds that of the direct solver. We find the break-even point by setting the costs equal:

$$k \cdot n \approx n^2 \implies k \approx n$$

This means that the Jacobi algorithm becomes more expensive (in terms of Big-Oh complexity) than the specialized direct solver when the number of iterations k is on the order of the system size n.

For $n = 100$, the faster Jacobi algorithm becomes more expensive than the fast direct solver after approximately 100 iterations.