README:Case Study 2 – Krylov Subspace Methods

Overview

This repository contains my solutions for Case Study 2, which is divided into three parts:

1. **Arnoldi iteration**:
   Implementation of the Arnoldi iteration to generate an orthonormal basis for the Krylov subspace and to compute the corresponding upper Hessenberg matrix.
   The code is in `Arnoldi.cpp`.
2. **Serial GMRES**:
   Implementation of a serial variant of the GMRES algorithm to solve a square linear system $Ax = b$. The algorithm uses the Arnoldi process to build a Krylov subspace and then solves a small least-squares problem (using Givens rotations) to update the approximate solution. The residual history is recorded and normalized as $\|r_k\|/\|b\|$. The code is in `serial-GMRES.cpp`.
3. **Parallel GMRES**:
   A parallel version of the GMRES algorithm is implemented using MPI. The key computational kernels (matrix–vector product, vector norm, and the modified Gram–Schmidt process) are parallelized using MPI calls (`MPI_Allreduce` and `MPI_Allgather`) to ensure global consistency. This version is compared with the serial implementation to verify correctness. The code is in `parallel-GMRES.cc`.

File Structure

```
Assignment2/
├── README.md/pdf           # This file.
├── Arnoldi.cpp             # Implementation of the Arnoldi iteration (2.1).
├── serial-GMRES.cpp        # Serial GMRES implementation (2.2).
├── parallel-GMRES.cpp       # MPI-based (parallel) GMRES implementation (2.3).
├── plot.py                 # Python script to plot serial-GMRES.cpp.
├── plot2.py                 # Python script to plot parallel-GMRES.cpp.
├── plot_serial.png
├── plot_parallel.png
├── plot_parallel2.png
└── Makefile
```

Dependencies

- **C++ Compiler**: `g++` (with C++17 support)
- **MPI**: Open MPI (or an equivalent MPI implementation)
- **Python 3**: with `NumPy` and `Matplotlib` for plotting residual history.

Build Instructions

A Makefile is provided. To compile all executables, run:

```
make
```

When I run it, it shows as follows:

```
yujinhan@Yujins-MacBook-Pro A2 % make
mpicxx -std=c++17 -O2 -Wall -Wextra -o parallel-GMRES parallel-GMRES.cc
g++ -std=c++17 -O2 -Wall -Wextra -o serial-GMRES serial-GMRES.cc
g++ -std=c++17 -O2 -Wall -Wextra -o Arnoldi Arnoldi.cc
```

Arnoldi Iteration (Implemented in `Arnoldi.cpp`)

1. Overview

In this part of the assignment, the goal is to implement the Arnoldi iteration, which is a key process used in Krylov subspace methods (such as GMRES) to generate an orthonormal basis for the Krylov subspace and the corresponding upper Hessenberg matrix. The underlying idea is to start with an initial vector **b** and compute the sequence:

$$b,\ Ab,\ A^2b,\ \ldots,\ A^kb \tag{1}$$

while orthogonalizing each new vector against all previously computed ones. In this implementation, the modified Gram–Schmidt process is employed for the orthogonalization step to ensure improved numerical stability compared to the classical Gram–Schmidt algorithm.

---

2. Code Description

2.1 Functions

1. **printMatrix `and` printVector**
    - These helper routines are used to print matrices and vectors in a neatly formatted manner.
2. **norm2**
    - This function computes the Euclidean norm (L2 norm) of a given vector.

- It accumulates the sum of squares of the vector elements and returns the square root of this sum.
  This function is crucial for normalizing vectors later in the algorithm.

3. **Matrix–Vector Multiplication Functionality**
   - This code multiplies an $n \times n$ matrix A with an n-dimensional vector v to return $w = Av$.
   - There is no standalone function defined in the code ,including in the Arnoldi iteration loop.
   - Example from the code:

```
// Compute v = A * Q[:,j]
for (int i = 0; i < n; i++) {
    for (int p = 0; p < n; p++) {
        v[i] += A[i][p] * Q[p][j];
    }
}
```

4. **Other (check functions)**

   checkOrthogonality

   - This function computes the matrix Q^TQ to verify that each column of Q is normalized (diagonal entries close to 1) and that different columns are nearly orthogonal (off-diagonal entries close to 0).

   checkArnoldiRelation

   - This optional function verifies the Arnoldi relation:

$$AQ_m \approx Q_{m+1}H_m, \tag{2}$$

     where Q_m consists of the first m columns of Q.
   - It calculates the Frobenius norm of the difference $AQ_m - Q_{m+1}H_m$ to assess the accuracy of the relation.

```
// checkOrthogonality(Q);

// checkArnoldiRelation(A, Q, H, iterations);
```

   At the end of the code, I commented out the output of these two functions to make the output clearer.
   This is just the code I used to check the answer.

2.2. Arnoldi Iteration Core

Function: `arnoldi`

This function implements the Arnoldi iteration and accepts the following parameters:

- `b`: the initial vector (of length n)
- `A`: the $n \times n$ matrix
- `k`: the number of iterations (producing $k+1$ orthonormal vectors); the Hessenberg matrix H is of size $(k+1) \times k$
- `tol`: a tolerance value for early stopping if the new vector becomes nearly zero

Key Steps in the Function:

1. **Initialization**
   - The initial vector b is stored as the first column of the orthonormal basis matrix Q.
   - b is normalized using its L2 norm (computed by `norm2(b)`), so that the first column becomes

$$q_0 = \frac{b}{\|b\|}. \tag{3}$$

2. **Iteration Loop** (for $j = 0, 1, \ldots, k-1$):
   - **Matrix–Vector Product**:
     - Compute $v = A \times q_j$ using a nested loop over the matrix rows and columns. This step is equivalent to the functionality of a standalone `matVecProduct` function.
   - **Orthogonalization (Modified Gram–Schmidt)**:
     - For each previously computed vector q_i (where $i = 0$ to j):
       - Compute the projection coefficient:

$$h_{ij} = q_i^T v. \tag{4}$$

       - Subtract the projection from v:

$$v = v - h_{ij} q_i. \tag{5}$$

   - **Residual Norm and Early Termination**:
     - Compute the norm of the updated vector v and store it as $h_{j+1,j}$.
     - If $h_{j+1,j}$ is smaller than the tolerance `tol`, the process terminates early.
   - **Normalization**:
     - Normalize v to obtain the new orthonormal vector q_{j+1}, and assign it as the next column of Q.
3. **Return**:
   - The function returns a pair containing the Hessenberg matrix H and the orthonormal basis matrix Q.

2.3. Main Function

- **Matrix A Definition:**
  - A 10×10 matrix A is defined according to the assignment specifications, with specific values for diagonal and off-diagonal elements.
- **Vector b Definition:**
  - A 10-dimensional vector b is defined with the following values:

    ```
    std::vector<double> b = {0.7575, 2.7341, -0.5556, 1.1443, 0.6453, -0.0855, -0.6237, -0.4652, 2.3829, -0.1205};
    ```

  - **Note:** Although the description previously mentioned $b[i] = \frac{i+1}{10}$, the actual code uses the specific values shown above.
- **Iteration Count:**
  - The number of iterations is set to 9, meaning the algorithm will produce 10 orthogonal vectors.
- **Calling the Arnoldi Function:**
  - The `arnoldi(b, A, iterations)` function is invoked to generate the Hessenberg matrix H and the orthonormal basis Q.

Output:

- The program prints each orthogonal vector correctly by iterating over the columns of Q (each column corresponds to an orthonormal vector), followed by printing the Hessenberg matrix H, as follows:

```
for (unsigned int j = 0; j < Q[0].size(); j++)
{
    std::cout << "The " << j << "th orthogonal vector is: " << std::endl;
    for (unsigned int i = 0; i < Q.size(); i++)
    {
        std::cout << Q[i][j] << " ";
    }
    std::cout << std::endl;
}
```

Run the code:

```
./Arnoldi
```

We get the answers:

```
yujinhan@Yujins-MacBook-Pro A2 % ./Arnoldi
Matrix size: 10x10, Iterations: 9
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
The 0th orthogonal vector is:
0.187114 0.675363 -0.137241 0.282659 0.159399 -0.0211197 -0.154063 -0.114911
0.588611 -0.0297653
The 1th orthogonal vector is:
0.336015 -0.00240904 0.477757 0.262672 0.266658 0.21068 0.418734 0.460981 0.0309363
0.292487
The 2th orthogonal vector is:
-0.389027 0.303149 0.160124 -0.395881 -0.434703 0.423846 0.213208 -0.0226751
0.204944 0.343236
The 3th orthogonal vector is:
0.482155 0.195425 0.130783 -0.245507 -0.204947 -0.199988 0.532691 -0.446324
-0.141576 -0.258747
The 4th orthogonal vector is:
0.241538 -0.618526 -0.139186 -0.134089 -0.0198293 0.0579334 0.0840135 -0.203612
0.66323 0.172024
The 5th orthogonal vector is:
0.0949366 0.0593639 0.26343 -0.374105 -0.244497 -0.62489 -0.213643 0.4815 0.224315
-0.00663082
The 6th orthogonal vector is:
-0.450956 0.00876295 -0.150426 -0.2216 0.458607 -0.113615 0.495293 0.193588 0.22301
-0.411186
The 7th orthogonal vector is:
-0.323522 -0.0645239 0.689643 0.0634809 0.284602 -0.214487 -0.151598 -0.497933
0.0893309 0.0750341
The 8th orthogonal vector is:
0.15666 -0.0688031 0.33801 -0.143001 -0.0638964 0.51011 -0.29964 0.112242 0.121622
-0.674206
The 9th orthogonal vector is:
-0.261425 -0.136809 0.0944158 0.634501 -0.562976 -0.150192 0.246831 0.0571003
0.166845 -0.264407
The H matrix is:
     8.23156      18.0851     -0.62317     -3.54855      1.94926     -2.51269
-2.54991     -1.34719      1.57728
```

```
     18.1692      36.6892       6.4957     -4.19768     0.498176      -3.6563
 -0.956791     0.459278      3.43433
            0      9.33664      -2.5339    -0.451811     -1.06803     0.833658
 0.206075    -0.288543      0.79659
            0            0      7.72862     -1.85311     -6.69688     -2.02569
 -1.18686      2.78013     0.484962
            0            0            0      4.37355      1.23876      6.71937
 -1.36992      3.02143      4.71412
            0            0            0            0      3.31053      6.62381
  1.41879     -3.43179      -3.9318
            0            0            0            0            0      2.25512
  1.60813    -0.180356      1.07255
            0            0            0            0            0            0
  2.70059      2.50007     0.747663
            0            0            0            0            0            0
     0      6.46454     0.436884
            0            0            0            0            0            0
     0            0       5.4138
```

Serial Implementation of GMRES (Implemented in `serial-GMRES.cpp`)

1. Overview

In this part, we implement the **GMRES algorithm in a serial environment**. The GMRES method approximates the solution x of a linear system:

$$Ax = b \tag{6}$$

by iteratively constructing a Krylov subspace and minimizing the residual $\|r_k\| = \|b - Ax_k\|$ at each step. Specifically:

1. **Krylov Subspace Construction**:

   Generate an orthonormal basis Q and a Hessenberg matrix H (size $(m+1) \times m$) via a process similar to Arnoldi iteration.

2. **Least-Squares Solution**:
   Use Givens rotations to solve the small least-squares problem $\min \|g - Hy\|$, where g is initialized to βe_1 ($\beta = \|b\|$).

3. **Update Approximate Solution**:
   Once the least-squares system is solved for y, compute $x_k = x_0 + Qy$. The residual norm is recorded at each iteration for convergence analysis.

2.1. Functions

1. `norm2`
   This function computes the Euclidean norm (L2 norm) of a given vector.
2. `matVecProduct`
   Multiplies an $n \times n$ matrix A with an n-dimensional vector v to produce $w = Av$:

```
std::vector<double> matVecProduct(const std::vector<std::vector<double>> &A,
                                  const std::vector<double> &v)
{
    int n = A.size();
    std::vector<double> w(n, 0.0);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            w[i] += A[i][j] * v[j];
        }
    }
    return w;
}
```

3. `generateMatrixA` / `generateVectorb`

- `generateMatrixA(n)` creates an $n \times n$ tridiagonal matrix with diagonal entries = -4 and sub/super-diagonal entries = 1.
- `generateVectorb(n)` creates an n-dimensional vector b, where:

$$b[i] = \frac{i+1}{n} \tag{6}$$

4. `applyGivensRotation` / `generateGivensRotation`

- Compute and apply Givens rotations for zeroing out subdiagonal elements in the least-squares problem.
- These routines are crucial for efficiently maintaining an upper-triangular form of the Hessenberg matrix during GMRES iterations.

2.2. The GMRES Function

The core GMRES logic is in `gmres(...)`, which accepts:

- **Input**:
  - A ($n \times n$ matrix) and b (The right-hand side vector)

- m: number of iterations (in practice $m = \frac{n}{2}$)
- `tol`: tolerance

- **Procedure**:
  1. **Initialization**:
     - Set $x_0 = 0$. Compute $\beta = \|b\|$.
     - Set the first column of V to $\frac{b}{\beta}$.
     - Initialize the vector g with $g[0] = \beta$.
  2. **Krylov Subspace via Arnoldi-like Process**:
     - For $k = 0$ to $m - 1$:
       - Compute $w = Av_k$ with `matVecProduct`.
       - Orthogonalize w against existing basis vectors via modified Gram–Schmidt (`gramSchmidt`).
       - Store $\|w\|$ in $H_{k+1,k}$. If it is below `tol`, break early.
       - Normalize w to get v_{k+1}.
  3. **Givens Rotations and Least-Squares**:
     - For each iteration, apply or update Givens rotations to keep H nearly upper triangular.
     - Update g accordingly, and compute the new residual $\|r_k\|$ from $\|g\|$.
     - Record the normalized residual $\frac{\|r_k\|}{\|b\|}$ in `residualHistory`.
  4. **Solve Upper Triangular System**:
     - After finishing or hitting early stop, use back substitution to solve for y.
     - Compute final $x = x_0 + Vy$.

- **Output**:

  Returns the final approximate solution x and a vector of residual norms (`residualHistory`).

2.3. Main Function

In `main()`:

1. Iterates over a set of matrix dimensions $\{8, 16, 32, 64, 128, 256\}$.
2. For each n, calls `generateMatrixA(n)` and `generateVectorb(n)`.
3. Runs `gmres(A, b, m, tol)` with $m = \frac{n}{2}$.
4. Prints the final solution x and writes the normalized residual history to `gmres_residuals.txt` (as well as to the console).

Run the code:

```
yujinhan@Yujins-MacBook-Pro A2 % ./serial-GMRES
```

We get the answers:

```
----------------------------
Dimension size n = 8

Final solution x:
-0.0620722
-0.124144
-0.186216
-0.248289
-0.310361
-0.364653
-0.397312
-0.349344

Normalized residual history (||r_k|| / ||b||):
iter 1: 0.216587
iter 2: 0.053266
iter 3: 0.0134143
iter 4: 0.00332198

----------------------------
Dimension size n = 16

Final solution x:
-0.0312492
-0.0624984
-0.0937476
-0.124997
-0.156246
-0.187495
-0.218744
-0.249993
-0.281243
-0.312451
-0.343555
...
iter 120: 4.01855e-16
iter 121: 4.01855e-16
iter 122: 4.01855e-16
iter 123: 4.01855e-16
iter 124: 4.01855e-16
```

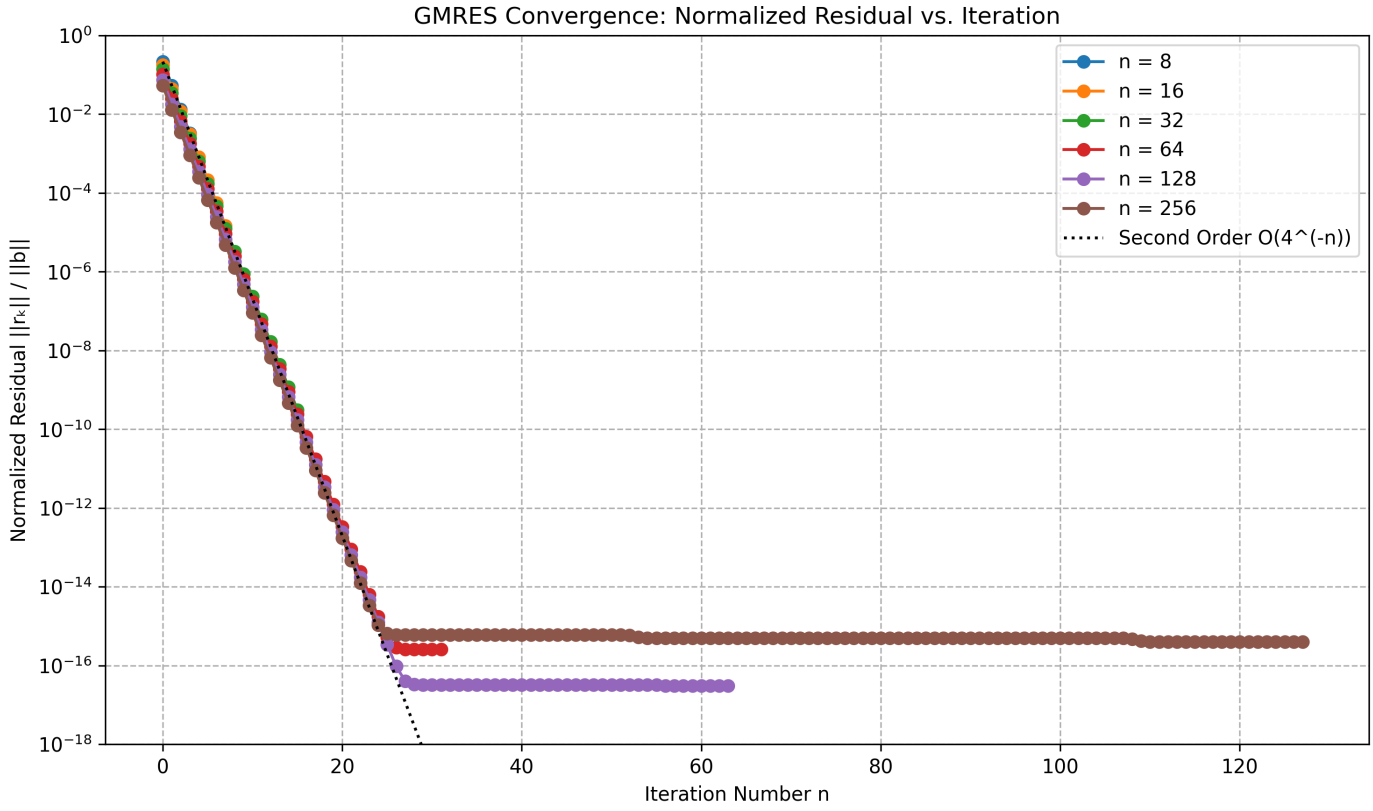
```
iter 125: 4.01855e-16
iter 126: 4.01855e-16
iter 127: 4.01855e-16
iter 128: 4.01855e-16

Residual data written to gmres_residuals.txt.
```

2.4. Plot

Then we use python to make the plot:

```
python3 plot.py
```

Below is an example of the GMRES Convergence plot (semi-log scale) generated by `plot.py`, showing the normalized residual $\|r_k\|/\|b\|$ versus iteration number for different matrix sizes n. We can see that for each n, the residual typically drops rapidly and eventually levels off near machine precision. The dotted line is a reference to a second-order convergence rate $\mathcal{O}(4^{-n})$, though actual rates may vary slightly:



After running the code implementation for various matrix sizes n, we record the normalized residual $\|r_k\|_2/\|b\|_2$ at each iteration k. We then use a semi-log graph to plot this ratio against the iteration number:

- **y-axis**: $\|r_k\|_2/\|b\|_2$
  This is the Euclidean (L2) norm of the residual $r_k = b - Ax_k$ divided by the Euclidean norm of the right-hand side b. By using this ratio, we get a dimensionless quantity indicating how effectively the algorithm is reducing the residual relative to the original problem scale.
- **x-axis**: Iteration number k
- **Scale**: Semi-log (logarithmic on the y-axis)
  This allows us to easily observe decreases of several orders of magnitude in the residual.

A plot (generated by `plot.py`) might look like the figure below, showing that for each n, $\|r_k\|_2/\|b\|_2$ rapidly decreases (often by many orders of magnitude) over the allotted iterations. The dashed line indicates a reference second-order convergence rate $\mathcal{O}(4^{-n})$, though the actual convergence curve depends on the matrix's spectral properties.

By inspecting this plot, we confirm that the GMRES method reduces the relative residual $\|r_k\|_2/\|b\|_2$ within a modest number of iterations ($m = n/2$), demonstrating the algorithm's convergence behavior for our test systems.

Parallel Implementation of GMRES (Implemented in `parallel-GMRES.cpp`)

1. Overview

We present a parallel variant of the GMRES algorithm using MPI. Specifically, three key operations are parallelized:

1. **Matrix–Vector Multiplication**
2. **Vector Norm Computation**
3. **Modified Gram–Schmidt Orthogonalization**

The code is located in `parallel-GMRES.cpp`. Like the serial version, this parallel code builds an orthonormal basis for the Krylov subspace and solves a small least-squares problem at each iteration, but it does so by distributing data and computations across MPI processes.

---

2. Key Components

1. Data Distribution

- The matrix A of size $n \times n$ is conceptually partitioned by rows among the MPI processes. Each rank `irank` handles a contiguous block of rows.
- Likewise, vectors of length n are partially owned by each process, with a local segment of size $\lfloor n/\mathrm{isize} \rfloor$ (plus a remainder if n is not divisible by the number of processes).

2. **Parallel Matrix–Vector Product**

```
std::vector<double> matVecProduct(const std::vector<std::vector<double>> &A,
                                  const std::vector<double> &v)
{
    int irank, isize;
    MPI_Comm_rank(MPI_COMM_WORLD, &irank);
    MPI_Comm_size(MPI_COMM_WORLD, &isize);
    int n = A.size();
    int n_local = n / isize;
    int my_n = n_local;
    if (irank == isize - 1) {
        my_n += n % isize;
    }
    int my_start = irank * n_local;

    // Local portion w
    std::vector<double> w(my_n, 0.0);
    for (int i = 0; i < my_n; i++) {
        for (int j = 0; j < n; j++) {
            w[i] += A[i + my_start][j] * v[j];
        }
    }

    // Gather local segments into a global vector
    std::vector<double> w_all(n, 0.0);
    MPI_Allgather(&w[0], my_n, MPI_DOUBLE,
                  &w_all[0], my_n, MPI_DOUBLE,
                  MPI_COMM_WORLD);
    return w_all;
}
```

- Each process multiplies its local rows of A by the global vector v.
- Results are combined via `MPI_Allgather` into a full result vector w_{all}.

3. **Parallel Norm Computation**

```
double norm2(const std::vector<double> &v)
{
    int irank, isize;
    MPI_Comm_rank(MPI_COMM_WORLD, &irank);
    MPI_Comm_size(MPI_COMM_WORLD, &isize);
    ...
    // Each rank sums squares of its local portion

    double sum = 0.0;
```

```
    for (int i = my_start; i < my_end; i++) {
        sum += v[i] * v[i];
    }

    double sum_all;
    MPI_Allreduce(&sum, &sum_all, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    return std::sqrt(sum_all);
}
```

- Each process computes the partial sum of squares of its local elements of v.
- A global reduction via `MPI_Allreduce` aggregates these partial sums into `sum_all`, from which the L2 norm is computed.

4. **Parallel Modified Gram–Schmidt**

```
void gramSchmidt(const std::vector<std::vector<double>> &V,
                 std::vector<double> &w,
                 std::vector<std::vector<double>> &H,
                 int k)
{
    // For each existing basis vector q_i, compute the dot product in parallel
    // and update w. Dot products are summed with MPI_Allreduce.
}
```

- Each rank computes a local dot product with its portion of w and the current basis vector q_i.
- `MPI_Allreduce` is then used to obtain the global dot product.
- The vector w is updated locally, then `MPI_Allgather` synchronizes the new w among all ranks.

---

3. The Parallel GMRES Algorithm

1. **Initialization**
   - Each process obtains the global dimension n and calculates its local row partition.
   - The vector b is read globally, and we compute $\beta = \|b\|$ in parallel.
   - The first column of V is set to b/β.
2. **Krylov Subspace Construction**
   - For iteration $k = 0, 1, \ldots, m-1$:
     1. **Matrix–Vector Product**: $w = Av_k$ (parallel)
     2. **Modified Gram–Schmidt**: orthonormalize w against existing columns of V using the parallel dot product and update steps.
     3. **Check**: $\|w\|$ (computed via `norm2`) for near-zero. If it's below `tol`, break.

4. Normalize w to form v_{k+1}.

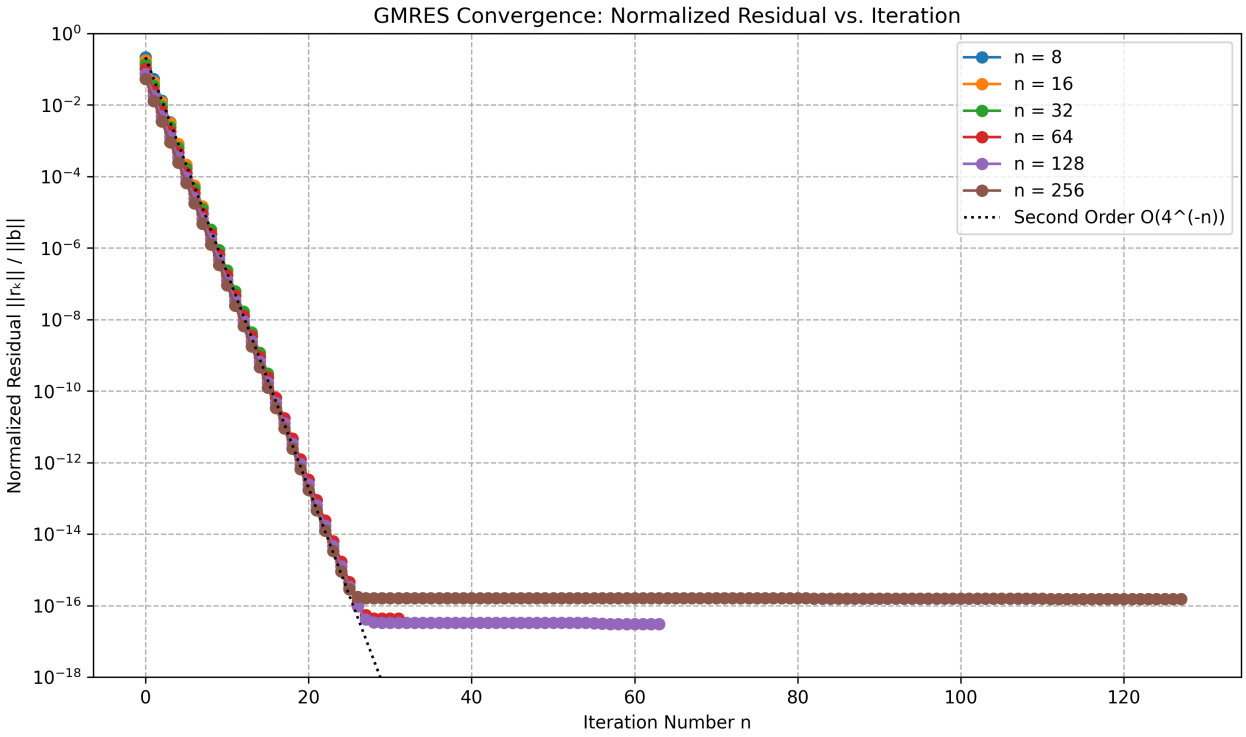
3. **Givens Rotations & Residual**
   - As in the serial code, we apply Givens rotations to maintain an upper-triangular form in the small Hessenberg matrix H.
   - The global residual $\|r_k\|$ is derived from the updated vector g. We store the ratio $\|r_k\|/\|b\|$ in `residualHistory`.

---

4. Correctness Testing

1. **Comparison with Serial GMRES**
   - We compare final solutions and residual histories from the parallel version with those from the serial code for the same matrix A and vector b. They should match closely (within floating-point rounding differences).
2. **Residual Plots**
   - As with the serial code, we record $\|r_k\|_2/\|b\|_2$ at each iteration and use a semi-log plot to visualize convergence. We can confirm the same rapid decrease in residual, verifying that the parallel approach is mathematically consistent.

     We get the plot2, as follows:



3. **MPI-Specific Checks**
   - Check that the partial sums in dot products and norms are aggregated properly with `MPI_Allreduce`.

---

5. Performance Considerations and Stop Criteria

1. **Large-Scale Problems**
   - For small n, communication overhead (e.g., `MPI_Allgather`, `MPI_Allreduce`) can dominate, so parallel GMRES may not yield significant speedup.
   - **Recommendation**: I tried to use larger n (e.g., 1024–2048–4096) to see meaningful parallel gains. As n grows, the computational cost of local matrix–vector products becomes larger relative to the communication overhead, and parallelization can reduce total runtime.

     ```
     iter 2048: 2.70715e-16
     Residual data written to gmres_residuals.txt.
     ./serial-GMRES  95.58s user 0.22s system 99% cpu 1:35.85 total

     iter 2048: 1.29311e-16
     Residual data written to gmres_residuals_paralell.txt.
     mpirun -n 4 ./parallel-GMRES  79.49s user 0.43s system 396% cpu 20.135
     total
     ```

     When running with a matrix dimension of $n = 2048$:

     - **Serial GMRES**:
       The serial version completed in approximately **95.6 seconds** (total time of 1:35.85).
     - **Parallel GMRES with 4 Processes**:
       The parallel version achieved a total runtime of about **20.1 seconds**. This represents a significant speedup compared to the serial run (roughly a 4.7x improvement).

     For small to moderate problem sizes, the cost of communication (using `MPI_Allgather`, `MPI_Allreduce`, etc.) can dominate the overall runtime. In our tests, using 4 processes provided a good balance between computation and communication, yielding a substantial speedup over the serial version.

2. **Stopping Criterion**

   The code has a commented line:

   ```
   // if (H[k + 1][k] < tol * 2e-7)
   // {
   //     break;
   // }
   ```

   And we use this stopping criterion:

```
double res = std::fabs(g[k + 1]);
        residualHistory.push_back(res);

        // Absolute residual check
        if (res < tol * 2e-7)
            break;
```

We prefer checking `std::fabs(g[k + 1])` as our stopping criterion because it provides a direct measure of the residual norm after all the transformations (Givens rotations) have been applied. This approach is more reliable in determining when the iterative process has converged to a sufficiently accurate solution. The commented-out alternative, which checks `H[k + 1][k]`, does not capture the full effect of the rotations and may lead to premature or delayed termination of the iterations.

In this implementation, the residual is compared against a scaled tolerance, `tol`$\times 2 \times 10^{-7}$. The base tolerance `tol` is typically set relative to machine precision (e.g., 1×10^{-8}), and the scaling factor 2×10^{-7} is used to account for rounding errors and other numerical effects that may prevent the residual from decreasing indefinitely. This adjusted threshold ensures that once the residual is on the order of machine precision, further iterations are unlikely to yield a meaningful improvement.

Benefits:

- **Efficiency**: By stopping the iterations once the residual is sufficiently small, we avoid unnecessary computation and reduce the overall runtime.
- **Numerical Stability**: The threshold accounts for numerical round-off errors, ensuring that the algorithm does not attempt to achieve a level of accuracy beyond what is numerically feasible.
- **Practical Convergence**: In practice, when the residual $\|r_k\|$ drops below this threshold, the approximate solution x_k is already as accurate as needed for most applications.

---

6. Example Execution and Results

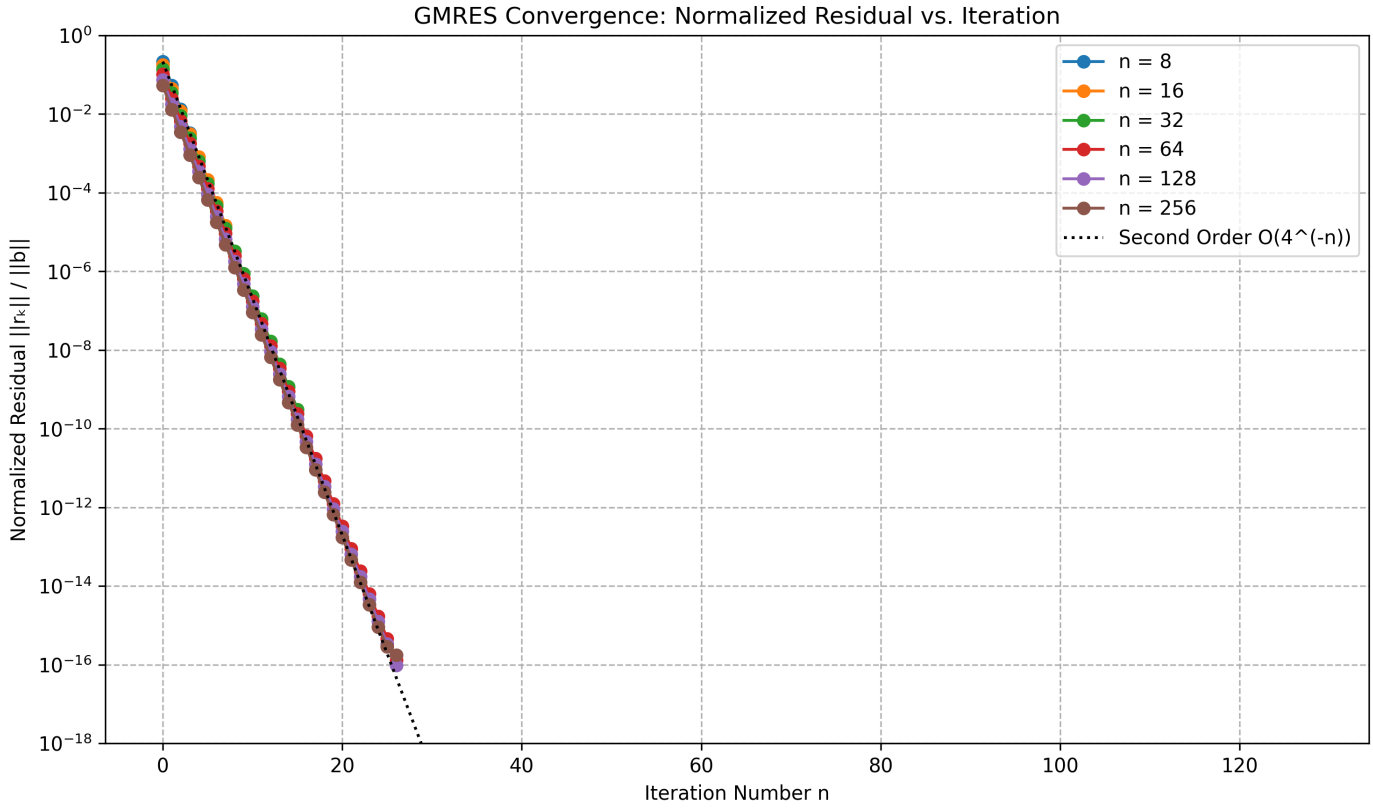
After compiling, we can run the code:

```
mpirun -np 4 ./parallel-GMRES
```

And then we can make the plot

```
yujinhan@Yujins-MacBook-Pro A2 % python3 plot2.py
```

Get the `plot_parallel2.png` as follows:



Conclusion

- **Parallel GMRES** extends the serial method by distributing data and computations among MPI processes.
- **Key Steps** (Matrix–Vector multiplication, Norm, Orthogonalization) are parallelized via collective operations such as `MPI_Allreduce` and `MPI_Allgather`.
- **Correctness** is verified by comparing final solutions and residual curves with the serial code.
- **Performance** depends on balancing the cost of local computations and communication overhead. For bigger n, we generally see better parallel speedups.