UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

# Parallelize a 3D Poisson problem

Numerical Methods Assignement
Supercomputing for Challenging Applications

**Daniela Wonneberger, Riccardo Selis**

---

**Summary:** In this project, we explored parallelization strategies for solving the 3D Poisson equation using a Preconditioned Conjugate Gradient (PCG) solver. Focusing on achieving efficient convergence across different mesh sizes, we applied preconditioners, such as SSOR and Additive Schwarz, to improve solver performance and stability. Sequential and parallel implementations were analyzed to identify key computational bottlenecks. We conducted strong and weak scaling analyses to assess the effectiveness of parallelization across varying thread counts. Additionally, we evaluated the impact of the relaxation parameter $\omega$ on convergence in SSOR, finding an optimal range for improved solver efficiency. Finally, we examined the effect of different right-hand side (RHS) configurations on convergence behavior, demonstrating that high-frequency RHS components are more readily dampened, whereas low-frequency modes persist, highlighting the significance of tailored preconditioning strategies.

## Contents

**1 Preliminar Timin Analysis of the Code** **2**
1.1 Further Profiling of the Code . . . 2
1.2 Optimization Approach and Parallelization Strategy . . . 2

**2 Parallelization of the Unpreconditioned Code** **3**
2.1 Implementations of the Parallel Functions . . . 3
2.2 Scalability Analysis of Unpreconditioned Code . . . 5

**3 Parallelization of the Additive Schwarz Preconditioner** **7**
3.1 Parallel Implementation of Functions . . . 7
3.2 Analysis of the results and Profiling of the Parallel Code . . . 9

**4 Advanced Tasks** **12**
4.1 Performance Analysis varying $\omega$ . . . 12
4.2 Paralleizing SSOR Preconditioner . . . 14

**5 Convergence Analysis with different Right-Hand Side** **15**
5.1 Experimental Setup . . . 15
5.2 Advanced Analysis on Conjugate Gradient Convergence's Properties . . . 17
5.3 Implications for High-Frequency and Low-Frequency Modes in Poisson Problem . . . 18
5.4 Final Analysis on Convergence Related to RHS Spectrum . . . 20

# 1. Preliminar Timin Analysis of the Code

## 1.1. Further Profiling of the Code

Before parallelizing the PCG solver, it is crucial to identify the main time-consuming parts of the algorithm. By breaking down the execution times for different preconditioners (id, ssor, and as), we can assess where computational resources are being spent—whether in preconditioning, matrix-vector multiplication, dot products, or vector updates. This helps guide optimization efforts for maximum performance improvement. Additionally, knowing all the partial times allows us to focus on specific functions and understand what needs to change to reduce computation time effectively. The data highlights that preconditioning methods, particularly `ssor` and `as`, significantly shift the computational burden towards preconditioning time. For both `ssor` and as, preconditioner time dominates, accounting for 65% or more of total execution, making it a key candidate for parallelization. In contrast, for the `id` method (without preconditioning), matrix-vector multiplication and dot product operations are the most time-consuming, indicating that optimizing these parts would yield the best results for parallelization. Thus, in preconditioned techniques, efforts should focus on parallelizing the preconditioner application, while non-preconditioned approaches should target matrix-vector and dot product operations.

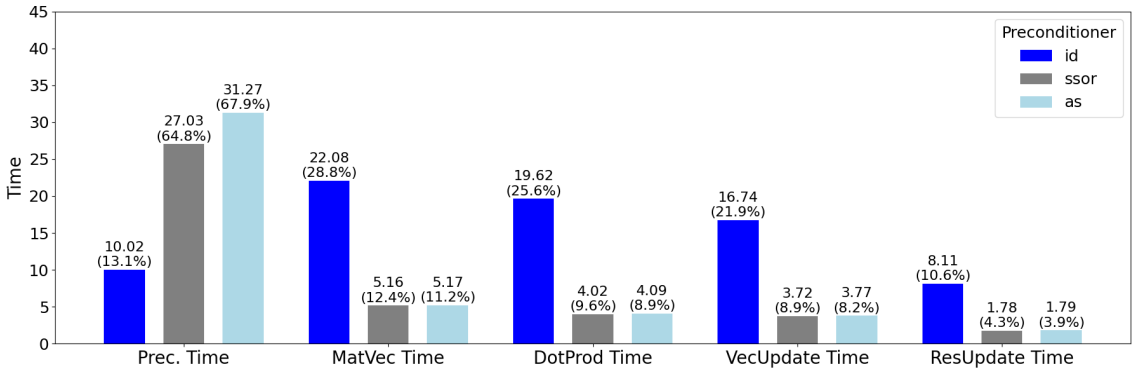


Figure 1: Execution times for different operations with `n = 200`, default parameters, and `rhs_1`. Times shown for three preconditioners: id, SSOR, and AS.

## 1.2. Optimization Approach and Parallelization Strategy

The optimization and parallelization of the 3D Poisson problem solver were approached systematically, beginning with a detailed time analysis to identify the most computationally expensive sections of the code. For both the preconditioned and unpreconditioned versions, it was clear that certain operations, particularly preconditioning (Prec) steps, matrix-vector multiplications, and dot products, were the primary contributors to runtime. Recognizing these bottlenecks was crucial for targeted optimization efforts. To gain a deeper understanding, each function in the code was analyzed individually, and parallelized incrementally using OpenMP directives. Each parallelization step was followed by rigorous timing experiments across varying thread counts, allowing for precise assessments of performance improvements.

This approach involved testing individual components, such as matrix-vector multiplication, dot products, and preconditioning, with different thread configurations to verify their scaling efficiency and ensure that parallelization was yielding tangible benefits. For each section, timing data provided insights into whether the code achieved significant speedup or encountered diminishing returns due to overheads from parallel execution. Additionally, complex portions of the code were examined to detect any remaining bottlenecks, such as memory access inefficiencies in functions like the copy operation, which could hinder parallel performance.

Based on these findings, an OpenMP parallelization strategy was applied to optimize each function independently, leveraging techniques such as task parallelism and loop-level parallelism. Through this methodical optimization and parallelization process, each section of the code was refined to improve overall solver efficiency on the Boada cluster.

# 2. Parallelization of the Unpreconditioned Code

## 2.1. Implementations of the Parallel Functions

### mul_poisson3d

The function `mul_poisson3d` computes the discrete Laplacian of a 3D grid, which is essential for solving the Poisson equation. It iterates over each grid point, calculating finite differences based on neighboring values and storing the result in `Ax`. To enhance performance, the function employs OpenMP for parallelization by applying the `#pragma omp parallel for` directive to the outermost loop over the z-dimension (`k`). This allows multiple threads to simultaneously process different slices of the grid, effectively utilizing multi-core processors. Consequently, the computation time is significantly reduced, enabling the function to scale efficiently for large problem sizes.
The `dot` function computes the dot product of two vectors `x` and `y` of size `n`, returning their scalar product. To parallelize this computation, the `#pragma omp parallel for reduction(+:result)` directive is used, distributing iterations across threads. This OpenMP reduction clause efficiently accumulates partial results from each thread, combining them into the final `result`. This approach accelerates the dot product calculation by leveraging multi-core architectures.

### parallel_copy

The `parallel_copy` function performs a parallelized memory copy operation for arrays `x` to `Ax` of size `n`. The function divides the array among available threads by calculating a `share` for each thread based on the array size and number of threads. Each thread copies its assigned segment of `x` into `Ax`, ensuring that boundaries are handled to avoid overlap and excess copying. If a thread's segment ends before the array boundary, it handles only its remaining elements using `memcpy`. This parallelized copy operation reduces memory duplication time on large arrays, effectively leveraging multi-threaded architecture.

### pc_identity

The `pc_identity` function acts as an identity preconditioner, copying the vector `x` into `Ax` without alteration. In parallel mode, it employs `parallel_copy`, which utilizes multi-threading to distribute the copy operation across available threads. Each thread handles a distinct segment of the vector, effectively lowering the workload per thread and accelerating the execution. The parallel approach thus optimizes performance by streamlining data transfer and reducing sequential memory operation delays.

### pcg

The `pcg` function is a parallelized Preconditioned Conjugate Gradient (PCG) solver used to iteratively solve a linear system $Ax = b$ with an optional preconditioner `Mfun`. Key components of this function are optimized for parallel execution to improve performance, particularly on large-scale systems. The parallelization strategy utilizes OpenMP to split computationally expensive tasks across multiple threads.
The residual computation `r[i] = b[i] - r[i]` leverages an `#pragma omp parallel for` directive to update each element of the residual vector independently. By dividing this task across threads, the update of each `r[i]` is executed concurrently, significantly reducing the computation time.
In the initial setup, the vector `p` is set equal to `z`. In parallel mode, this is achieved using `parallel_copy`, a custom function that distributes the copy operation among threads. This approach is advantageous for large vectors where a standard sequential copy would be slow, allowing the work to be distributed and reducing the time taken for initialization.
The search direction update `p[i] = z[i] + beta * p[i]` is also parallelized with

`#pragma omp parallel for`. This operation, which updates each element of the search direction vector `p`, benefits from the parallelization as each element's computation is independent. This approach improves efficiency by distributing the workload among threads, allowing faster completion of the search direction update step.
The operations `x[i] += alpha * p[i]` and `r[i] -= alpha * q[i]` are responsible for updating the solution vector `x` and the residual vector `r` in each iteration. Both are implemented with `#pragma omp parallel for`, allowing concurrent updates across threads. By parallelizing these updates, the function minimizes bottlenecks in solution and residual computation, enhancing overall performance. The total time for matrix-vector operations (MatVec), dot products, preconditioning, residuals, search direction, and solution updates is tracked separately to analyze performance improvements due to parallelization. These timing metrics help in identifying potential bottlenecks and optimizing further.

## 2.2. Scalability Analysis of Unpreconditioned Code

The analysis of the performance results obtained from the parallelized Preconditioned Conjugate Gradient (PCG) solver across different mesh sizes ($n = 100$, 150, 200, and 250) and varying thread counts offers insight into scaling efficiency and computational bottlenecks within the solver.

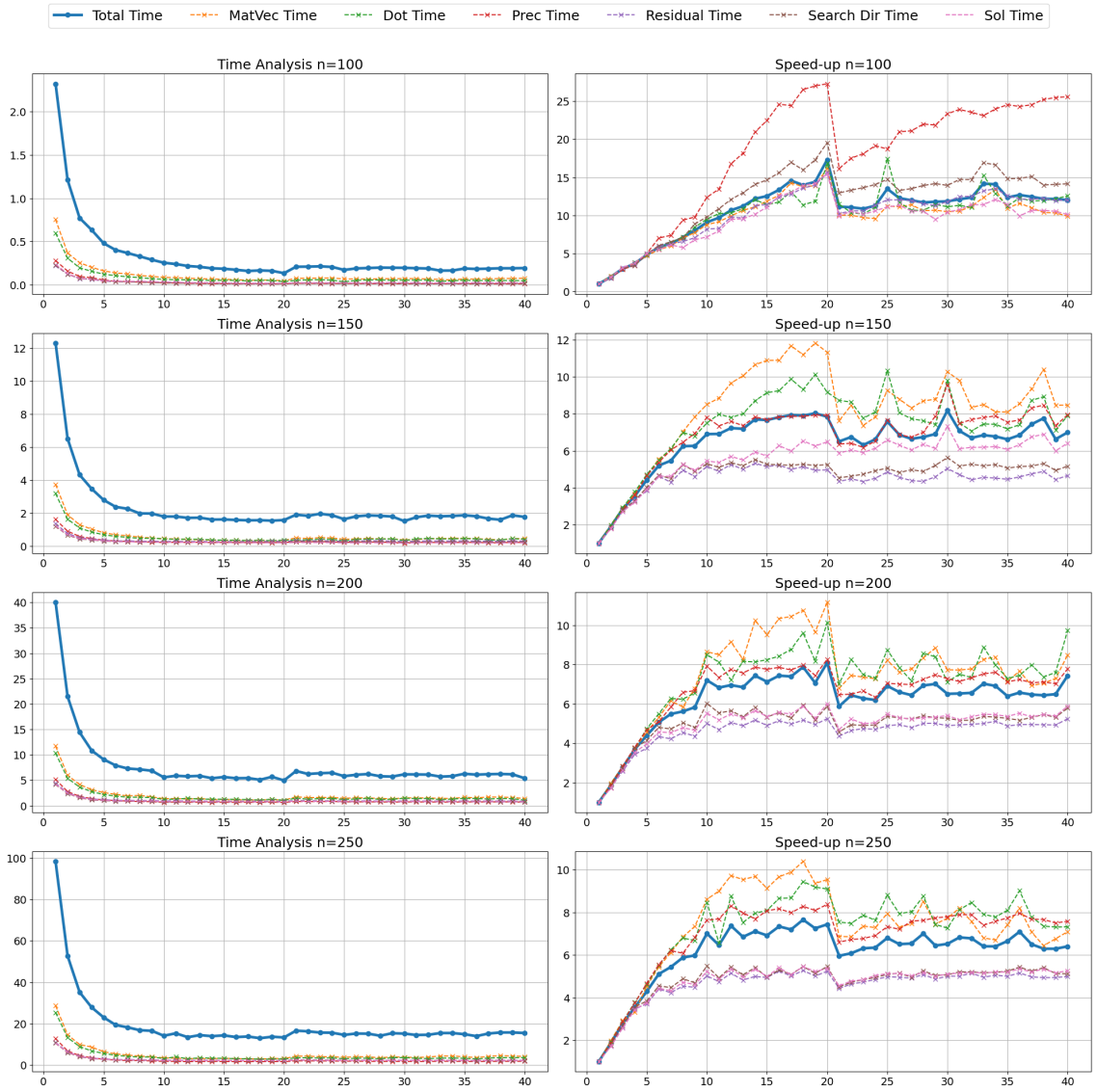


Figure 2: Time analysis and speed-up for various operations across different mesh sizes ($n = 100, 150, 200, 250$) using `rhs_1` and default parameters. Results show timing and scaling for Total Time and Partial times.

As thread counts increase, each mesh size initially shows a notable reduction in computation time, indicating effective parallelism. However, beyond a certain thread threshold, the benefits diminish, likely due to overheads that counteract further gains. For smaller grids like $n = 100$, computation time decreases significantly up to around 16-20 threads, after which gains plateau or even reverse, suggesting limited work per thread and increased communication overhead. Larger grids, such as $n = 250$, exhibit a more consistent improvement across thread counts, benefiting from better load balancing and larger computational domains.

Analyzing time components, Matrix-Vector Multiplication (MatVec) and dot products dominate computation time, especially in smaller grids. MatVec operations scale effectively up to 20 threads but encounter bandwidth limitations at higher counts. Dot products show similar trends but contribute less to the overall time than MatVec. The preconditioning step, particularly with the identity preconditioner, scales efficiently with minimal overhead, demonstrating its lower impact

on performance.
As thread count increases, synchronization and communication costs rise, particularly in residual calculations, which impacts scalability, especially for larger grids. Ultimately, the solver effectively uses parallel resources up to around 16-20 threads for smaller grids, while larger grids maintain scaling efficiency at higher thread counts, despite some limitations due to synchronization and bandwidth.

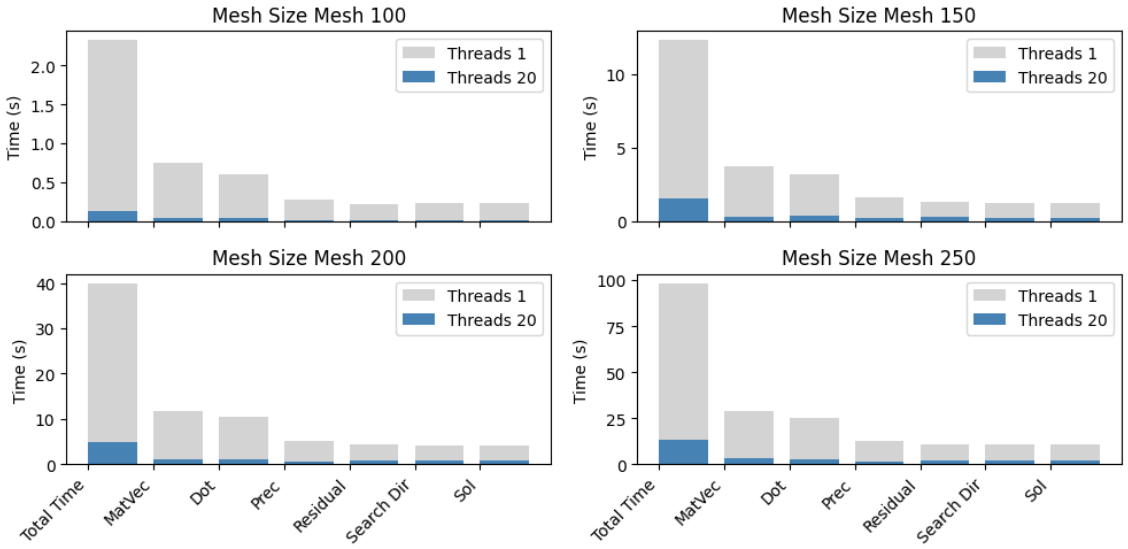


Figure 3: Comparison of execution times for key operations across different mesh sizes (100, 150, 200, 250) with 1 and 20 threads, using rhs_1 and default parameters.

In this weak scaling analysis, problem size was increased with the number of threads, adjusting mesh size to maintain comparable computational load per thread. As thread count rose from **1** to **40**, the problem size scaled from **100** to **342**, allowing us to assess solver behavior under consistently expanding workloads.
The total execution time increased with both problem size and thread count, highlighting parallel efficiency challenges due to data exchange and synchronization overhead. Matrix-Vector multiplication time rose significantly, from 0.68 seconds with 1 thread to 15.7 seconds with 40 threads, marking it as a major bottleneck impacted by memory bandwidth and data transfer latency. Dot product operations scaled relatively well but experienced time increases, reaching 11.84 seconds in the final iteration, indicating cache and inter-thread communication impacts as data volume expanded. Preconditioning and residual calculations also rose, though preconditioning remained below Matrix-Vector time, showing its lower sensitivity to scaling. Residual times grew moderately, impacted by synchronization demands with higher thread counts.

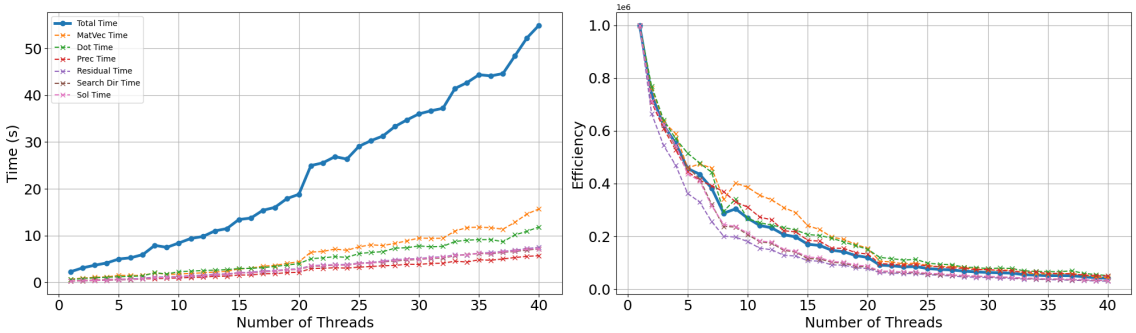


Figure 4: Weak scalability time and efficiency analysis for various operations as a function of thread count, using `rhs_1` and default parameters. Shows performance scaling and efficiency trends.

# 3. Parallelization of the Additive Schwarz Preconditioner

## 3.1. Parallel Implementation of Functions

### schwarz_get

The `schwarz_get` function manages the boundary conditions and local grid copying for the Schwarz preconditioner. The main task of this function is to transfer values from the global grid array `x` to a local array `x_local` for each subdomain, while ensuring that specific boundary slices are set to zero. This zeroing operation is crucial for defining boundary conditions across subdomains. Parallelization is implemented through `collapse` directive to merge the nested loops and distribute workload across threads efficiently. This strategy enhances data copying performance by balancing the computational load, especially for large subdomains, while zeroing the boundaries in parallel.

### schwarz_add

The `schwarz_add` function aggregates values from the local grid, `Ax_local`, into the global grid, `Ax`, over a specified subdomain. This accumulation is crucial in domain decomposition methods, allowing individual subdomain computations to be seamlessly integrated into the full solution. The parallelization approach collapses the three nested loops, distributing the workload across threads for each grid dimension simultaneously. This strategy maximizes concurrency, enabling each thread to handle non-overlapping portions of the domain and enhancing computational efficiency. By accelerating the addition process, this method significantly reduces aggregation time, especially on large grids, contributing to improved scalability and performance.

### memset

The `parallel_memset` function initializes a portion of memory, setting each element in a specified range to a given `value`. This parallel implementation divides the memory range `n` into nearly equal-sized segments assigned to each thread. By dividing `n` by the total number of threads, each thread is allocated a `share` of the memory space. The function ensures boundary conditions are met: if a thread's share exceeds the memory limit, it adjusts to prevent overflow. Parallelizing this initialization process reduces the time required to set large memory blocks, which is particularly beneficial in cases where zero-initialization or constant assignment is a repeated operation. This technique ensures efficient memory usage across threads.

### ssor_forward_sweep

The function `ssor_forward_sweep` performs a forward sweep of the Symmetric Successive Over-Relaxation (SSOR) method over a 3D grid, a core component of the solver's parallelization strategy. This function is designed with an efficient parallelization approach that utilizes OpenMP tasking for distributing the computational workload and ensuring dependencies are respected across multiple blocks in a three-dimensional grid.

To achieve this, the domain is divided into blocks, each of size `BS`, allowing for fine-grained parallelism. The function defines three block-level loops (in `i`, `j`, and `k`) that iterate over the grid and assign each block as an independent task. By using OpenMP's `collapse` directive in the sequential version, all three nested loops are combined, optimizing execution time in non-parallel runs. In the parallel version, the `task` directive is employed within an OpenMP `single` region to launch each block as a task, thus ensuring multiple threads execute different blocks concurrently.

The parallelization of `ssor_forward_sweep` relies heavily on defining task dependencies using the `depend` clause in OpenMP. The three dependency pointers, `dep_in_i`, `dep_in_j`, and `dep_in_k`, establish dependencies on neighboring blocks, ensuring that each block only begins its computation once its required neighboring data is ready. For instance, the pointer `dep_in_i` checks the status of the previous block along the `i`-axis, ensuring data integrity as tasks run concurrently.

Within each task, the code defines boundaries for each block to ensure computations remain within

the assigned domain. The loops then sweep through each `ii`, `jj`, and `kk` grid point within the block, updating values using neighboring points and applying the SSOR weight `w`. The structured use of blocks allows for scalable division of work between threads and provides each task with sufficient granularity to optimize memory and cache usage.
This strategy ensures efficient parallelization by dividing the computational load across multiple threads while maintaining the order of operations required by the SSOR method. By applying a dependency structure, the function minimizes idle times and maximizes resource utilization across threads, which is crucial for larger grid sizes where computational demands are high.

## ssor_backward_sweep

The `ssor_backward_sweep` function performs a backward sweep in the SSOR method and is **symmetric** to `ssor_forward_sweep`. It operates on a 3D grid, iterating over blocks in reverse order and applying a similar dependency-based parallelization strategy. In the parallel version, tasks are created for each block using OpenMP with `task` dependencies. Dependency pointers `dep_in_i`, `dep_in_j`, and `dep_in_k` ensure that each block respects the required dependencies for boundary data. Each block processes grid elements in reverse order and calculates updated values, using values from neighboring points and the relaxation factor `w`. This method maintains SSOR's sequential consistency while achieving parallel efficiency.

### ssor_diag_sweep

The `ssor_diag_sweep` function performs a diagonal sweep in the SSOR method. This step involves scaling each element of the 3D array `Ax` within the specified sub-domain bounds.
The scaling factor `6*(2-w)/w` is applied to every element, using the relaxation parameter `w`. In the parallel version, an `omp parallel for` directive is used to parallelize the outermost loop, distributing work across threads. This approach is efficient because each element's operation is independent, meaning no dependencies need to be managed. This diagonal scaling step is computationally lightweight compared to the forward and backward sweeps, thus benefiting from parallelization with minimal overhead.

## 3.2. Analysis of the results and Profiling of the Parallel Code

Analyzing the results across different mesh sizes ($n = 100$, $n = 150$, $n = 200$, and $n = 250$) and thread counts from 1 to 20 reveals significant trends in parallel scaling performance for each computational routine.

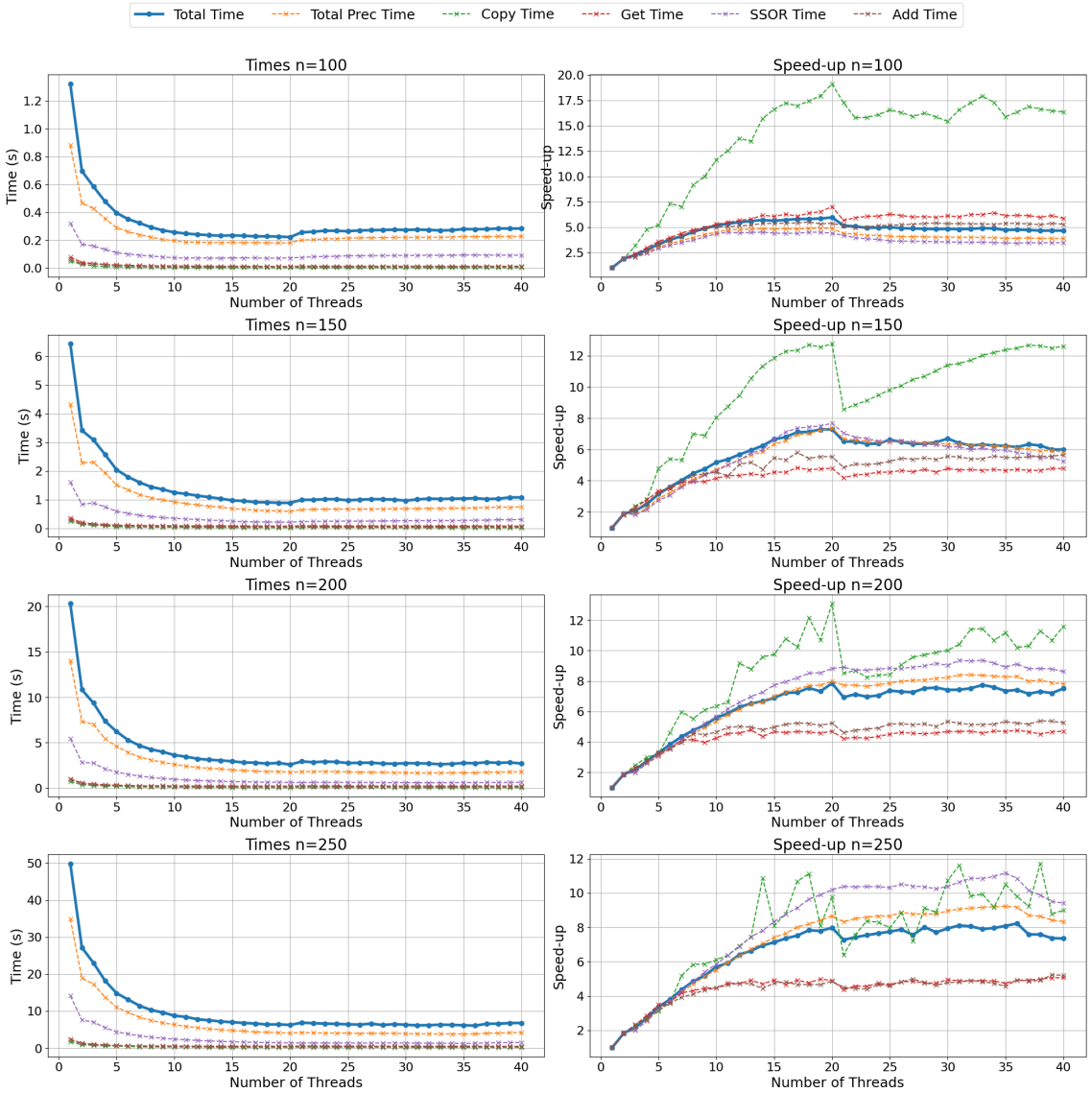


Figure 5: Execution time and speed-up analysis across various thread counts and mesh sizes (n = 100, 150, 200, 250), highlighting time spent in preconditioning, SSOR, and other operations. In the execution default parameters are used and `rhs_1` is applied.

The total execution time generally decreases with an increasing number of threads, reflecting the expected speedup from parallelization. However, the rate of improvement diminishes as the thread count increases, suggesting the influence of synchronization overhead and diminishing parallel efficiency at higher thread counts.
The SSOR (Symmetric Successive Over-Relaxation) preconditioning, a crucial part of the computation, exhibits considerable scaling with parallelization. This routine is split into forward, diagonal, and backward sweeps, with forward and backward sweeps demonstrating roughly similar time allocations. As threads increase, the SSOR sweep times reduce significantly, but they remain the most time-consuming part of the computation, emphasizing it as a primary computational bottleneck. This indicates that further optimizations in SSOR could yield significant performance gains.

The Matrix-Vector multiplication, while also benefiting from parallelization, shows a steady scaling but contributes less to the total execution time compared to SSOR. This pattern suggests that the multiplication routine is well-suited to parallel execution but is less critical in terms of optimization focus relative to SSOR.
The dot product operation shows good parallel scaling initially, with reduced times as thread count increases, but its contribution to overall computation is minor compared to SSOR and Matrix-Vector multiplication. This is expected, as dot products are typically bandwidth-bound rather than compute-bound.
Overall, the results highlight that while each routine benefits from parallelization, SSOR preconditioning remains the dominant factor in computation time across all mesh sizes. The decrease in incremental speedup with higher thread counts suggests bottlenecks due to memory bandwidth and inter-thread communication, particularly in larger meshes where data locality and cache efficiency are challenging. Future improvements could focus on optimizing SSOR further or adjusting task scheduling to reduce synchronization delays, particularly as the number of threads increases.

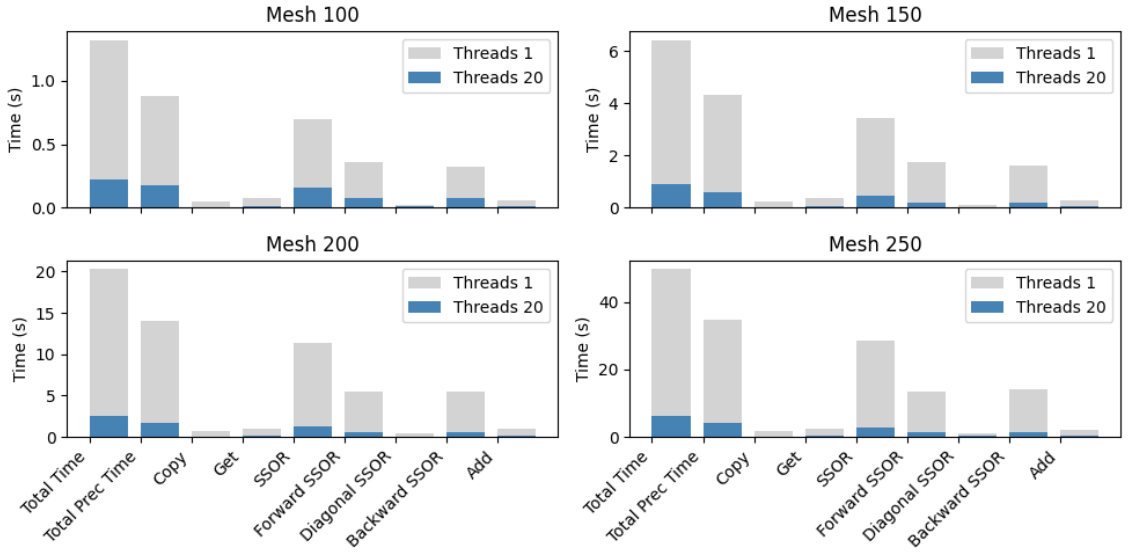


Figure 6: Comparison of execution times for various operations at mesh sizes 100, 150, 200, and 250, using 1 and 20 threads, focusing on SSOR components, using default parameter and `rhs_1`.

In this weak scaling study, we observe how performance is affected as the problem size and thread count are scaled together. Starting with a problem size of 100 and increasing to 342 while incrementally increasing threads from 1 to 40, we assess computational efficiency in the context of the **Additive Schwarz** preconditioner.

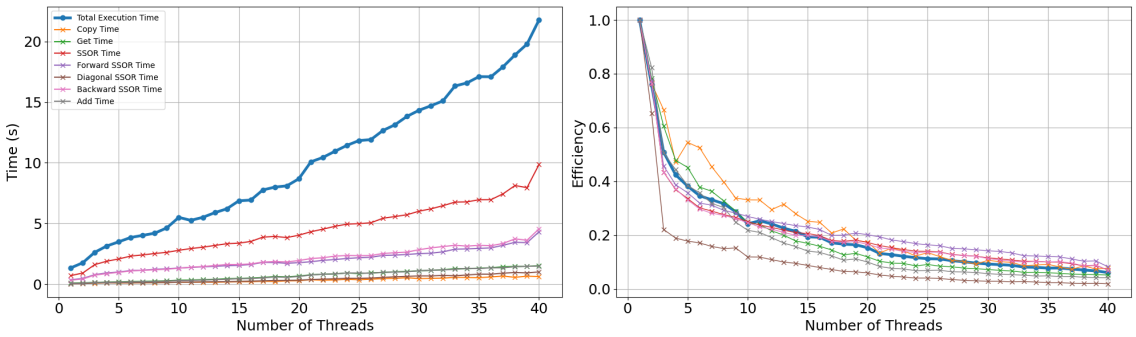


Figure 7: Execution time and efficiency across thread counts for various operations, highlighting SSOR components. Efficiency decreases with increased threads, indicating synchronization and memory bandwidth limitations.

The data indicates a steady increase in total time from 1.33 seconds for 1 thread to 21.78 seconds

for 40 threads. The dominant time contributors are Matrix-Vector multiplication and Dot product, which show a gradual increase with more threads, particularly as data transfer and synchronization overhead grow with problem size. The SSOR time, an essential component of the AS preconditioner, scales particularly poorly due to the reliance on inter-thread communication in forward and backward sweeps. Starting at 0.69 seconds for 1 thread, it rises disproportionately to 9.85 seconds for 40 threads, underscoring a synchronization bottleneck.
Other operations such as Copy, Get, and Residual time scale more efficiently but contribute less significantly to total time. Despite parallelization, the inherent need for sequential operations in some SSOR steps limits scalability, and future optimization could focus on further minimizing inter-thread dependencies to enhance scalability.

# 4. Advanced Tasks

## 4.1. Performance Analysis varying $\omega$

The Successive Over-Relaxation (SSOR) method is an iterative technique used to improve convergence rates by decomposing the matrix $A$ as $A = D - L - U$, where $D$, $L$, and $U$ are the diagonal, lower, and upper triangular parts of $A$, respectively. The SSOR preconditioner is then formulated as:

$$M_{\text{SSOR}} = (D - \omega L)D^{-1}(D - \omega U),$$

with $\omega$ as the relaxation parameter.
The choice of $\omega$ significantly affects the method's performance. For $\omega = 1$, SSOR reverts to the symmetric Gauss-Seidel method. Values $\omega < 1$ yield under-relaxation, often slowing convergence, while $\omega > 1$ provides over-relaxation, accelerating convergence but risking instability if $\omega$ is too high. Optimizing $\omega$ balances convergence rate and stability and depends on $A$'s spectral properties. In the **Additive Schwarz** context, SSOR is applied within each subdomain, with different $\omega$ values impacting the conditioning and convergence behavior locally, thus influencing the global solver performance.
To identify the optimal relaxation parameter $\omega$ for the SSOR method within the Additive Schwarz preconditioner, I conducted a systematic manual analysis. Initially, I explored a broad range of $\omega$ values from 0.1 to 1.9 with a step size of 0.1. This preliminary analysis aimed to observe the general behavior of the solver across various mesh sizes, identifying trends in convergence and the effectiveness of the preconditioner with SSOR as $\omega$ varied.

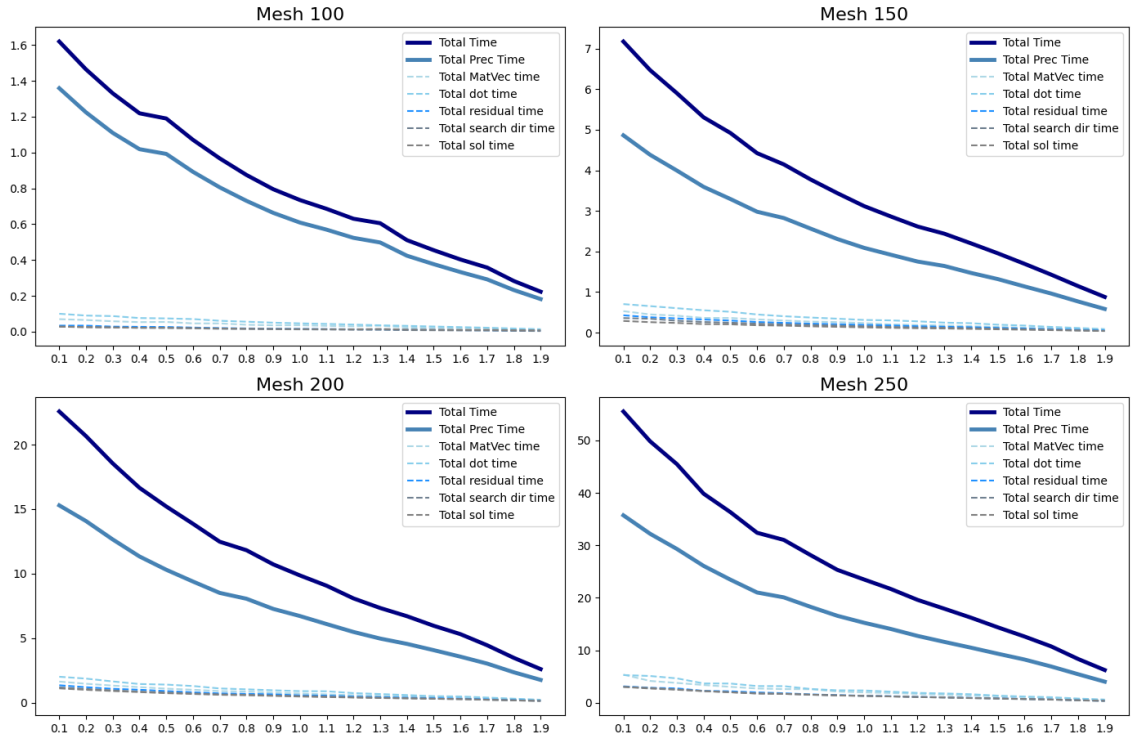


Figure 8: Execution times for different value of `n` for different value of relaxation parameter $\omega \in (0.1, 1.9)$ to analize the global trend. The other parameter are default, and `rhs_1` is used.

Following this initial exploration, I refined the analysis to focus on the range where $\omega$ approached optimal performance. Knowing that even minor adjustments in $\omega$ can significantly impact the solver's efficiency, I proceeded with a narrower range between 1.900 and 1.999, using a step size of 0.01. This involved running 100 simulations for each mesh size, capturing the sensitivity of the solver to these small changes in $\omega$. This approach allowed me to pinpoint a more precise value of $\omega$ that best balances convergence speed and stability. By systematically narrowing the search, I was able to enhance the solver's performance across different mesh sizes.

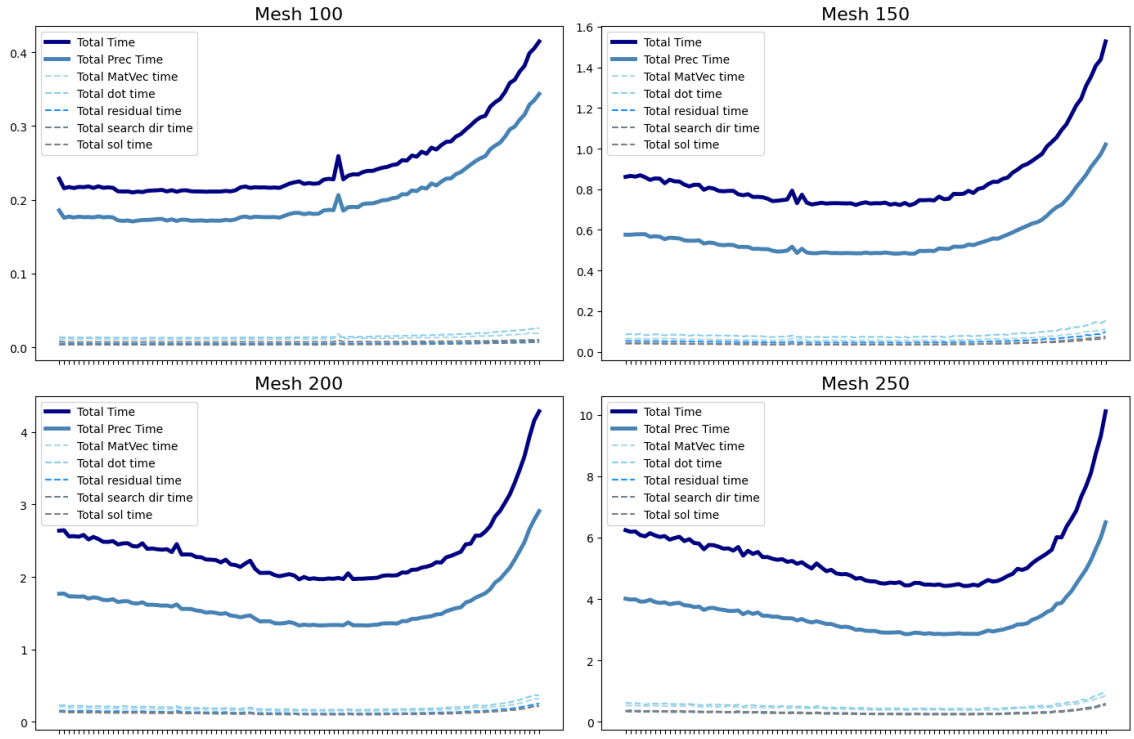


Figure 9: Execution times for different value of `n` for different value of relaxation parameter $\omega \in (1.900, 1.999)$ with step size `s=0.001` to analize the local behaviour. The other parameter are default, and `rhs_1` is used.

The results obtained from the omega variation analysis provide detailed insights into the SSOR preconditioner's sensitivity to omega adjustments. Increasing omega generally improves convergence up to a certain threshold, after which further increments lead to increased iterations and computation time due to numerical instability. The optimal range lies around values close to **1.92-1.94**, where the total number of steps and residual reduction balance to minimize computational costs.
Execution times, particularly for the SSOR phase, vary significantly across omega values, reflecting the preconditioner's dependency on omega to achieve efficient convergence. Notably, the backward and forward SSOR sweeps contribute substantially to overall runtime, confirming that fine-tuning omega within this range yields improved solver performance and reduced total processing times, especially for larger meshes where precise omega adjustments mitigate bottlenecks in iterative convergence. Overall, the study illustrates that omega tuning within a narrow band around the optimal range enhances the stability and efficiency of the iterative process, particularly for complex 3D meshes.

## 4.2. Paralleizing SSOR Preconditioner

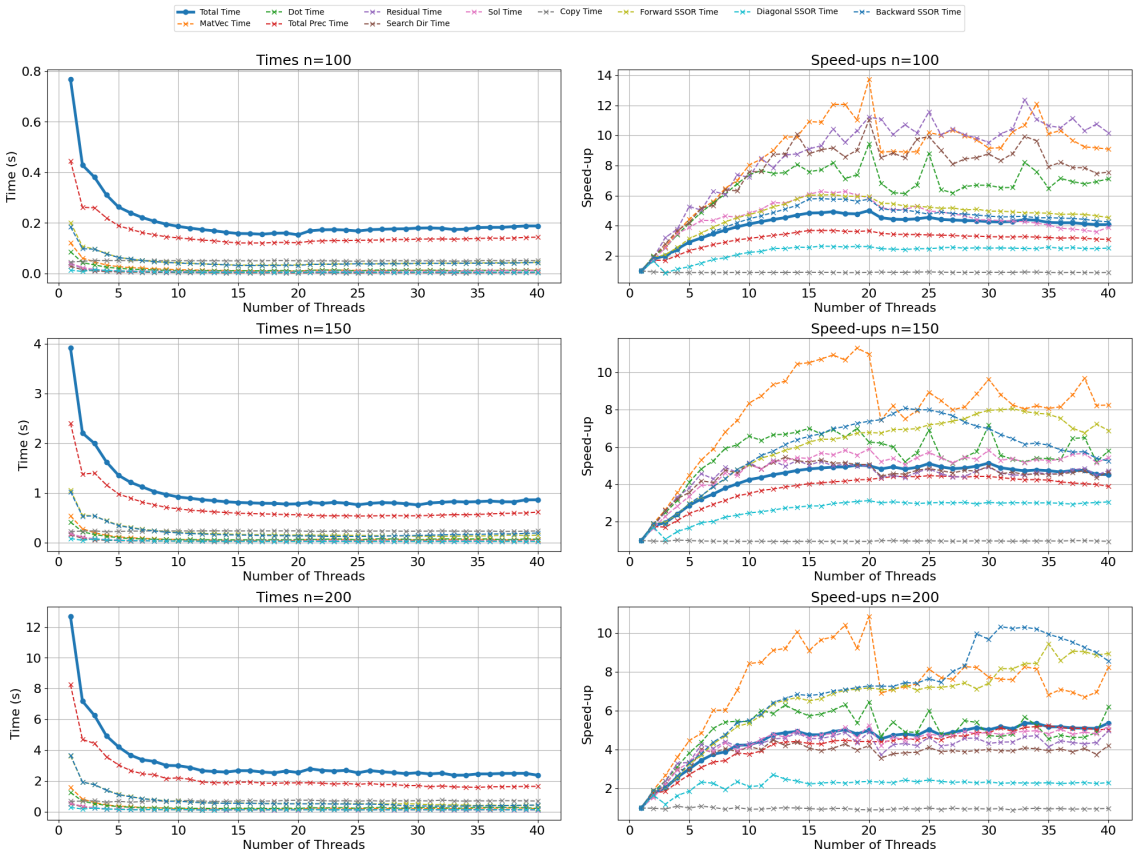


Figure 10: Execution time and speed-up analysis for various operations across thread counts at mesh sizes (n = 100, 150, 200). Highlights efficiency gains and performance trends.

This scalability test examines the effect of increasing thread count on runtime and performance for the SSOR preconditioned iterative solver across different mesh sizes (100, 150, and 200). The results demonstrate a clear decrease in computation time as thread count increases, highlighting the benefits of parallelization. For instance, significant speed-ups are achieved with 10 threads, but the benefits diminish beyond 20 threads due to parallel overheads, such as synchronization and memory contention. This trend indicates that optimal resource utilization occurs within the range of 10-20 threads, depending on the problem size.
The plots reveal a consistent reduction in specific operation times, particularly for preconditioning steps, where speed-ups range from 5x to over 10x at higher thread counts. In contrast, operations like search direction updates and residual computations show less sensitivity to parallelization, with lower relative speed-ups. These findings emphasize the importance of identifying optimal thread configurations to balance performance gains against computational overhead.

# 5. Convergence Analysis with different Right-Hand Side

## 5.1. Experimental Setup

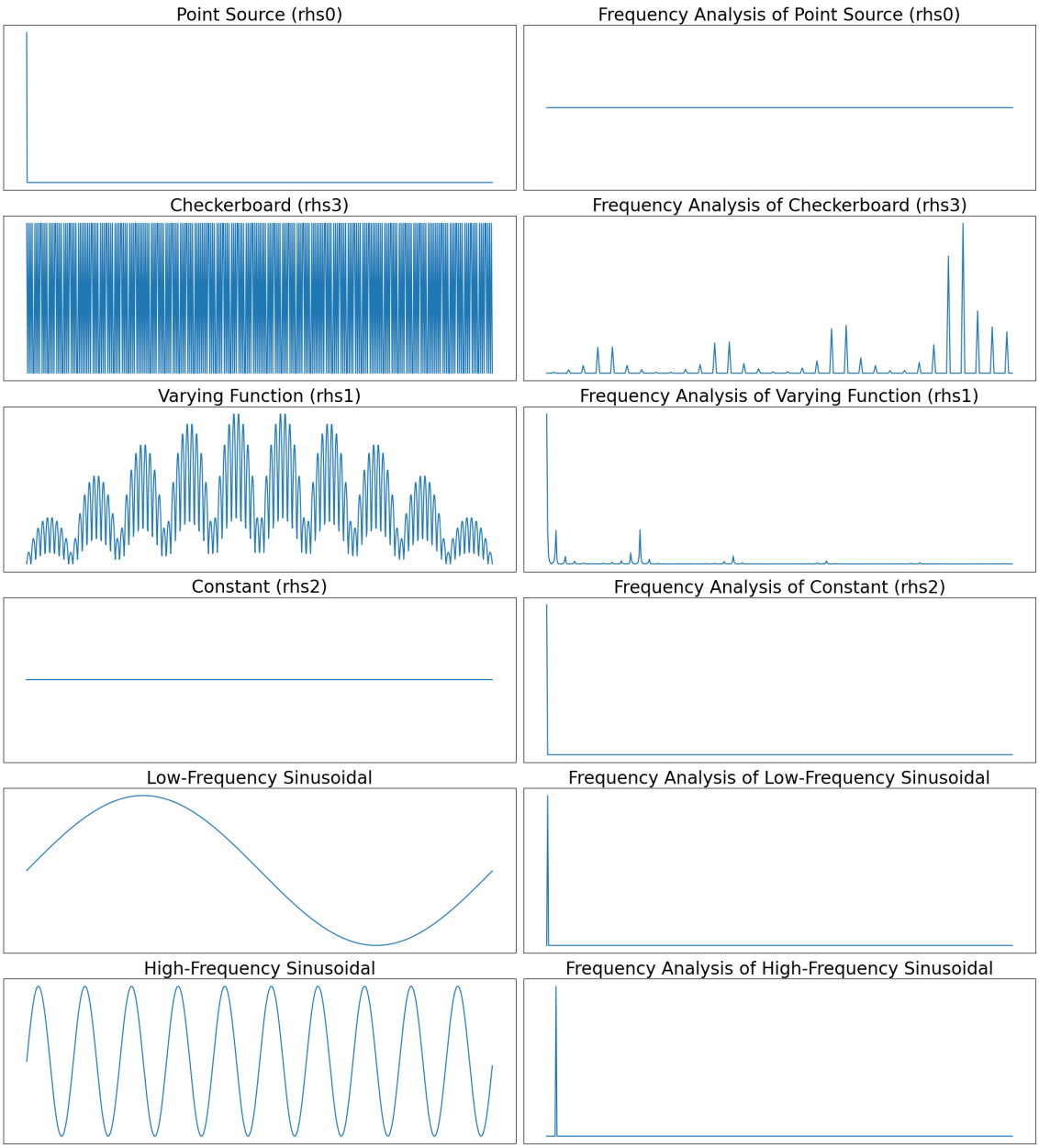


Figure 11: Qualitative analysis of different rhs functions and their frequency components. No scale is provided, focusing on identifying qualitatively high and low-frequency components.

In the final part of the project, we analyzed the impact of different right-hand side (RHS) configurations on the solver's performance, focusing on the frequency content of these RHS functions. The RHS functions were carefully chosen to represent various spatial patterns and frequency compositions, allowing us to investigate how different frequencies affect the convergence behavior of the solver.
The RHS configurations used in the analysis are as follows:

1. Point Source (`rhs0`):

$$b_i = \begin{cases} 1, & \text{if } i = 0, \\ 0, & \text{otherwise.} \end{cases}$$

This RHS represents a point source at the origin of the domain. It is analogous to a discrete delta function, which theoretically contains all frequency components equally. The broad frequency spectrum makes it ideal for examining the solver's response to inputs with extensive frequency content.

2. Smoothly Varying Function (`rhs1`):

$$b(x, y, z) = x(1 - x)\, y(1 - y)\, z(1 - z),$$

where $x, y, z$ are normalized spatial coordinates in the domain $[0, 1]^3$. This function varies smoothly across the domain and primarily consists of low-frequency components. It provides insight into the solver's performance when dealing with inputs that lack high-frequency content.

3. Constant Function (`rhs2`):

$$b_i = 1, \quad \forall i.$$

The constant RHS is uniform throughout the domain, containing only the zero-frequency (DC) component. It serves as a baseline case to understand the solver's efficiency with the simplest possible input, devoid of any spatial variation.

4. Checkerboard Pattern (`rhs3`):

$$b(i, j, k) = (-1)^{i+j+k},$$

where $i, j, k$ are the integer indices along each spatial dimension. This pattern creates an alternating sequence of $+1$ and $-1$ values in a regular, repeating fashion. The checkerboard RHS introduces higher-frequency components due to its rapid sign changes, allowing us to study the solver's handling of inputs with significant high-frequency content.

5. Low-Frequency Sinusoidal Function (`rhs_low_freq`):

$$b_i = \sin\left(2\pi m \frac{i}{N}\right), \quad m = 1,$$

where $N$ is the total number of discrete points and $m$ is the frequency multiplier set to a low value. This RHS introduces a single low-frequency sinusoidal wave, enabling us to observe the solver's effectiveness in processing inputs with specific low-frequency oscillations.

6. High-Frequency Sinusoidal Function (`rhs_high_freq`):

$$b_i = \sin\left(2\pi m \frac{i}{N}\right), \quad m = 10.$$

Similar to the low-frequency sinusoidal function but with a higher frequency mu ltiplier ($m = 10$), this RHS contains rapid oscillations corresponding to high-frequency components. It is used to assess how the solver copes with inputs dominated by high-frequency content.

For each RHS function, we performed a frequency analysis using the Fast Fourier Transform (FFT) to characterize their frequency compositions. By examining the spectral content of these RHS configurations, we can better understand the influence of different frequency components on the solver's convergence behavior. The varying spatial patterns—from constant to rapidly oscillating functions—provide a comprehensive set of test cases to evaluate the solver's performance across a spectrum of frequency inputs.

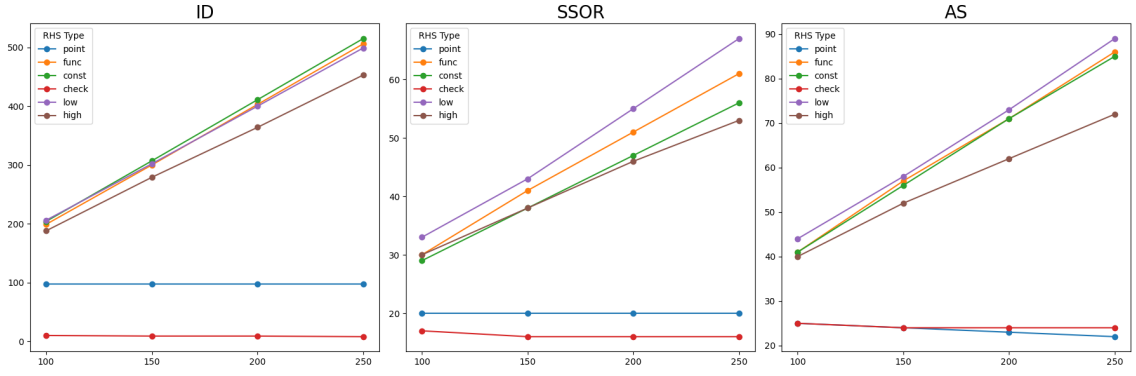


Figure 12: Comparison of execution times for different RHS types (point, func, const, check, low, high) across mesh sizes (100, 150, 200, 250) under ID, SSOR, and AS preconditioners.

Overall, the analysis shows that in solvers like PCG, high-frequency components are more easily reduced in early stages, while the smooth, low-frequency modes dominate the later iterations and slow down convergence. This highlights the need for preconditioning strategies that target these smooth modes, potentially improving performance across various applications where low-frequency components dominate the RHS. The analysis on the effect of different right-hand side (RHS) configurations using a preconditioned conjugate gradient (PCG) solver reveals significant insights into solver convergence behavior. Each RHS configuration exhibits a unique frequency profile, which directly influences convergence.

**High-frequency RHS**, like the checkerboard and high-frequency sinusoidal functions, typically see rapid initial error reduction, as high-frequency components are damped in the early PCG iterations. This rapid elimination of oscillatory components suggests that high-frequency content is less challenging for the solver, particularly when employing effective preconditioning strategies like SSOR or additive Schwarz, which optimize the convergence for such modes.

In contrast, **low-frequency RHS** setups, including the smoothly varying function and low-frequency sinusoidal, present a distinct challenge. The solver requires additional iterations to address these modes, as low-frequency errors persist longer in the iterative process. The results demonstrate this, with increased step counts observed for low-frequency RHS across different mesh sizes. The constant RHS also exemplifies low-frequency dominance, where the solver spends more iterations achieving residual reductions, emphasizing that smooth modes become the bottleneck for convergence.

## 5.2. Advanced Analysis on Conjugate Gradient Convergence's Properties

To understand the convergence behavior of the Conjugate Gradient (CG) method, it is essential to analyze how the error evolves at each iteration in terms of the eigenvalues and eigenvectors of the matrix $A$. Let $\{(\lambda_i, \mathbf{v}_i)\}_{i=1}^n$ denote the set of eigenpairs of the symmetric positive-definite matrix $A$, where $\lambda_i$ are the eigenvalues and $\mathbf{v}_i$ are the corresponding orthonormal eigenvectors satisfying $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$ and $\mathbf{v}_i^\top \mathbf{v}_j = \delta_{ij}$.

The initial error $\mathbf{e}^{(0)} = \mathbf{x}^{(0)} - \mathbf{x}^*$, where $\mathbf{x}^*$ is the exact solution and $\mathbf{x}^{(0)}$ is the initial guess, can be decomposed into the eigenvector basis:

$$\mathbf{e}^{(0)} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i, \tag{1}$$

where the coefficients $\alpha_i$ are the projections of the initial error onto the eigenvectors:

$$\alpha_i = \mathbf{v}_i^\top \mathbf{e}^{(0)} = (\mathbf{e}^{(0)}, \mathbf{v}_i). \tag{2}$$

This decomposition allows us to study how each component of the error along the eigenvector directions evolves independently during the iterative process.

The CG method minimizes the error at each iteration in the energy norm induced by $A$, known as the $A$-norm. The $A$-norm of a vector $\mathbf{e}$ is defined as:

$$\|\mathbf{e}\|_A = \sqrt{\mathbf{e}^\top A \mathbf{e}}. \tag{3}$$

This norm measures the error with respect to the quadratic form associated with $A$, providing a natural metric for convergence in problems involving symmetric positive-definite matrices. An important result in the analysis of the CG method is the bound on the $A$-norm of the error after $k$ iterations:

$$\|\mathbf{e}^{(k)}\|_A \leq 2\|\mathbf{e}^{(0)}\|_A \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k, \tag{4}$$

where $\kappa = \kappa(A) = \dfrac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of $A$, with $\lambda_{\max}$ and $\lambda_{\min}$ being the largest and smallest eigenvalues, respectively.

This inequality demonstrates that the convergence rate of the CG method is influenced by the condition number: a smaller $\kappa$ (i.e., better-conditioned matrix) leads to faster convergence. Using the initial error decomposition, the error at iteration $k$ can be expressed as:

$$\mathbf{e}^{(k)} = \sum_{i=1}^{n} e_i^{(k)} \mathbf{v}_i, \tag{5}$$

where $e_i^{(k)}$ represents the component of the error along the eigenvector $\mathbf{v}_i$ at iteration $k$.
In the CG method, the evolution of each error component $e_i^{(k)}$ is given by:

$$e_i^{(k)} = \phi_k(\lambda_i)\alpha_i, \tag{6}$$

where $\phi_k(\lambda)$ is a polynomial of degree $k$ that is determined by the method's iterates. This polynomial arises from the CG algorithm's property of minimizing the $A$-norm of the error over the Krylov subspace at each iteration. The polynomials $\phi_k(\lambda)$ constructed by the CG method are those that minimize the maximum error over the spectrum of $A$. Specifically, they are related to scaled and shifted Chebyshev polynomials of the first kind, which have optimal properties for minimizing the maximum deviation on an interval.
Using the properties of Chebyshev polynomials, we obtain the following bound on the error components:

$$|e_i^{(k)}| \leq 2 \left( \frac{\sqrt{\lambda_i} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_i} + \sqrt{\lambda_{\min}}} \right)^k |\alpha_i|. \tag{7}$$

This expression shows that the rate at which each error component decreases depends on the corresponding eigenvalue $\lambda_i$.
Analyzing the term within the parentheses:

$$\rho_i = \left( \frac{\sqrt{\lambda_i} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_i} + \sqrt{\lambda_{\min}}} \right), \tag{8}$$

we observe the following:

- **For large eigenvalues** ($\lambda_i \approx \lambda_{\max}$):
  - The numerator $\sqrt{\lambda_i} - \sqrt{\lambda_{\min}}$ is significantly larger than $\sqrt{\lambda_{\min}}$, but the denominator $\sqrt{\lambda_i} + \sqrt{\lambda_{\min}}$ is even larger.
  - Thus, $\rho_i$ is close to 1, but slightly less, making $\rho_i^k$ decrease slowly with $k$.
  - However, because $\lambda_i$ is large, the overall error component $|e_i^{(k)}|$ decreases rapidly due to the influence of $\lambda_i$ in the energy norm.
- **For small eigenvalues** ($\lambda_i \approx \lambda_{\min}$):
  - The numerator $\sqrt{\lambda_i} - \sqrt{\lambda_{\min}}$ approaches zero.
  - Therefore, $\rho_i$ is close to zero, and $\rho_i^k$ decreases rapidly.
  - However, since $\lambda_i$ is small, the error component $|e_i^{(k)}|$ does not diminish as quickly in the $A$-norm.

This apparent contradiction is resolved by considering the impact of $\lambda_i$ on the energy norm: larger eigenvalues amplify the error components in the $A$-norm, so reducing these components has a more significant effect on the overall error reduction.

## 5.3. Implications for High-Frequency and Low-Frequency Modes in Poisson Problem

In the context of the 3D Poisson problem:
For the 3D Poisson problem, the matrix $A$ represents a finite-difference discretization of the Laplacian operator $\nabla^2$ on a uniform $n \times n \times n$ grid with Dirichlet boundary conditions.
The **eigenvalues** $\lambda_{i,j,k}$ of the 3D Poisson matrix $A$ on an $n \times n \times n$ grid are given by:

$$\lambda_{i,j,k} = 6 - 2\cos\left(\frac{i\pi}{n+1}\right) - 2\cos\left(\frac{j\pi}{n+1}\right) - 2\cos\left(\frac{k\pi}{n+1}\right), \tag{9}$$

where $i, j, k = 1, 2, \ldots, n$. The form of these eigenvalues arises from the separability of the Laplacian operator in Cartesian coordinates. The eigenvalues has this properties:

- **Range:** The eigenvalues range approximately from 0 (for low-frequency modes) to 6 (for high-frequency modes).

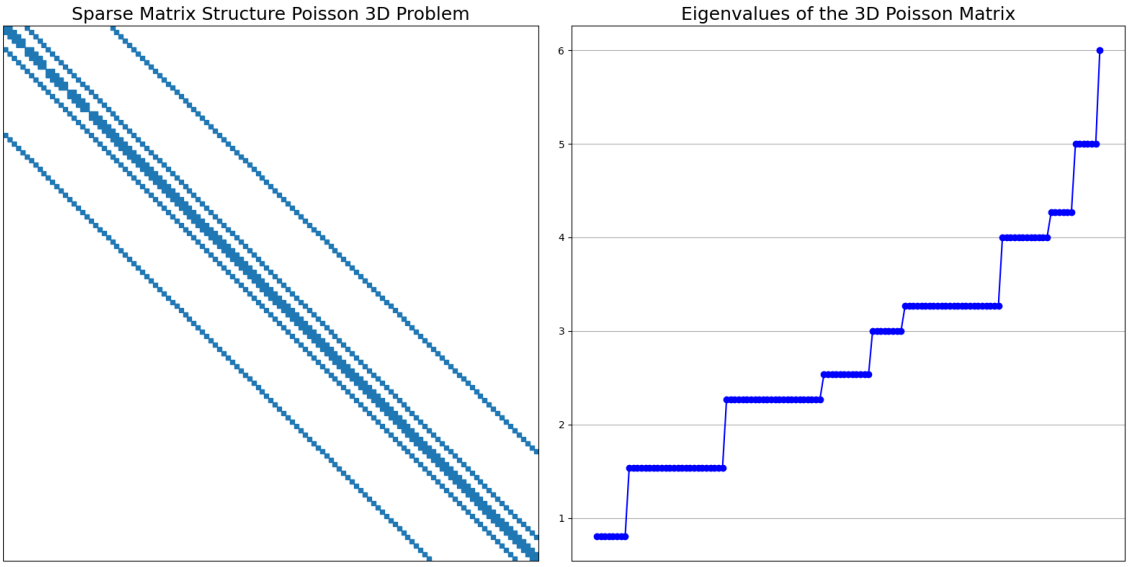


Figure 13: Visualization of the sparse matrix structure for the 3D Poisson problem (left) and the stepwise distribution of its eigenvalues (right). Studies are performed on small matrices to facilitate computation, hence the lack of scale

- **Spacing:** The eigenvalues are not uniformly spaced; they cluster near both ends of the spectrum. This clustering impacts the convergence properties of iterative solvers.
- **Symmetry:** The eigenvalue spectrum is symmetric with respect to frequencies in each dimension, corresponding to the separable, oscillatory nature of the modes.

The **eigenvectors** corresponding to each **eigenvalue** $\lambda_{i,j,k}$ are given by:

$$v_{x,y,z}^{(i,j,k)} = \sin\left(\frac{i\pi x}{n+1}\right)\sin\left(\frac{j\pi y}{n+1}\right)\sin\left(\frac{k\pi z}{n+1}\right), \tag{10}$$

where $x, y, z = 1, 2, \ldots, n$. The properties of eigenvectors are the following:

- **Sinusoidal Modes:** Each eigenvector represents a 3D discrete sine wave. This reflects the natural vibration modes of a cubic domain with Dirichlet boundary conditions.
- **Orthogonality:** The eigenvectors are orthogonal with respect to the standard inner product, an essential property for spectral decomposition methods.
- **Mode Numbers:** The indices $i, j, k$ specify the number of half-wavelengths that fit along each grid dimension, representing the spatial frequency of each mode.

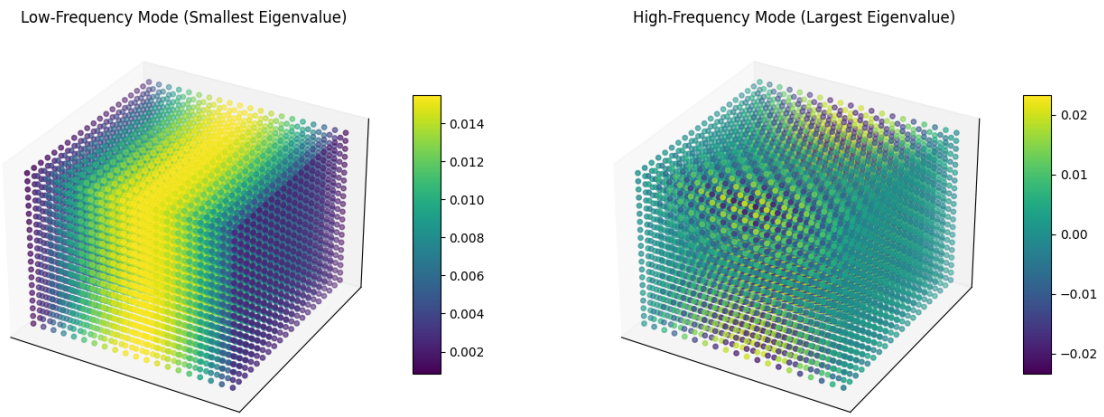


Figure 14: Visualization of low-frequency (left), related to lowest eigenvalue and high-frequency (right) eigenmodes related to highest eigenvalue for the 3D Poisson matrix. Low-frequency modes exhibit smooth variations, while high-frequency modes display rapid oscillations.

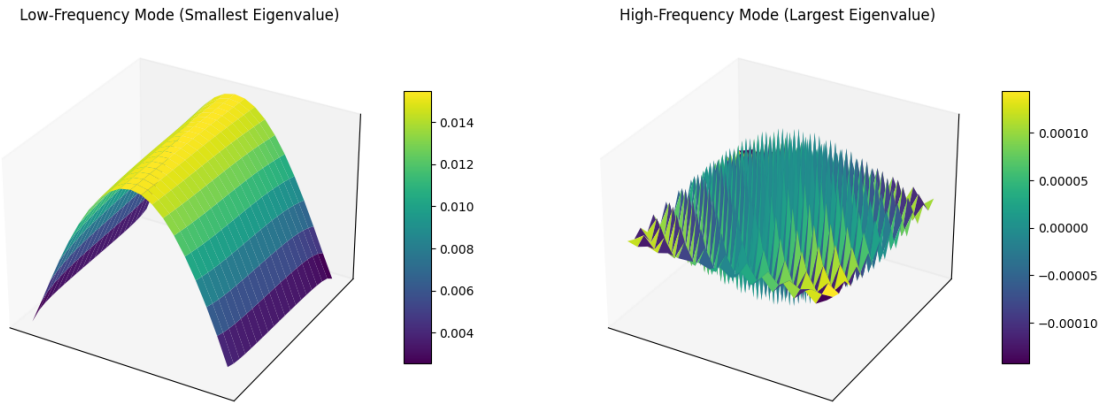


Figure 15: Visualization of low-frequency (left) and high-frequency (right) modes of the 3D Poisson matrix, corresponding to the smallest and largest eigenvalues. Plots show a 2D slice of the 3D grid.

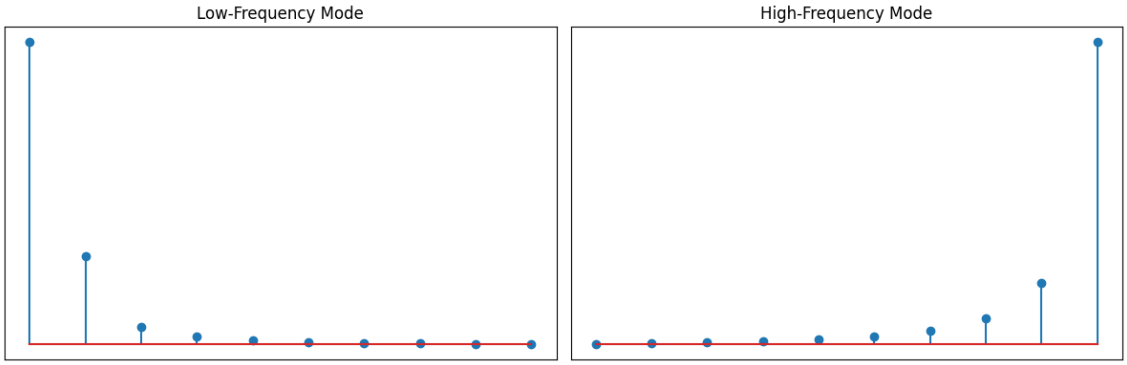


Figure 16: Frequency spectrum of the eigenmodes corresponding to the smallest eigenvalue (left, low-frequency) and largest eigenvalue (right, high-frequency) of the 3D Poisson matrix, showing dominant components.

## 5.4. Final Analysis on Convergence Related to RHS Spectrum

In conclusion, the analysis of the 3D Poisson matrix and its eigenstructure provides significant insights into the convergence behavior of iterative solvers like the Preconditioned Conjugate Gradient (PCG) method. The spectral decomposition shows that high-frequency components, associated with larger eigenvalues, are reduced more efficiently in the early stages of the PCG iterations. This rapid convergence for high-frequency modes suggests that oscillatory components in the right-hand side (RHS) are less problematic for the solver, particularly when effective preconditioners like SSOR or additive Schwarz are used. Conversely, low-frequency modes, linked to smaller eigenvalues, pose greater challenges as they dominate the error in later stages, slowing down overall convergence. This behavior underscores the importance of preconditioning strategies targeting smooth, low-frequency modes to enhance convergence rates. These findings highlight that in applications where the RHS exhibits low-frequency dominance, tailored preconditioning can significantly improve the solver's performance, reducing computational costs and enhancing efficiency in solving Poisson-type problems.