# Fast Poisson solver: a hybrid AI-enhanced approach

We now turn to the central objective of this internship: studying an hybrid - AI-enhanced multigrid solvers through learning techniques to define the best smoothing operators. While this concept may not be immediately familiar, this section aims to clarify it step by step. We begin with a theoretical overview of the underlying problem, followed by the presentation and discussion of the experimental results.

## 2.1 Theoretical and methodological framework

### 2.1.1 Problem statement and classical numerical approaches

#### Addressed problem

The goal is to solve the Poisson equation in 2D with homogeneous Dirichlet boundary conditions, a very common problem in physics and engineering — such as hydraulics, diffusion or electrostatic. The equation is expressed as follows :

$$\begin{cases} -\Delta u = f & \text{in} \quad \Omega = ]0,1[^2 \subset \mathbb{R}^2, \\ u = 0 & \text{on} \quad \partial\Omega. \end{cases}$$

Applying the finite difference method on an uniform Cartesian mesh leads us to the following equation :

$$\frac{4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2} = f_{i,j},$$

where $u_{i,j}$ the unknown solution at the vertex $(i, j)$ of the grid (see Figure 2.1).

Once this equation is established, the next step is to solve it numerically. To do so, we begin by discretizing the domain. Specifically, we divide the unit square $[0, 1] \times [0, 1]$ into a uniform grid with spacing $h = 1/N$.

Each grid point will represent a value of the unknown function $u$, and as we are under Dirichlet conditions, we only solve for the interior points, which are $(N - 1)^2$ in total. The Laplacian at a point is then approximated by a weighted combination of the solution at the

same vertex and its four neighbors. We refers to it as five-point stencil, shown below :

$$\frac{1}{h^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

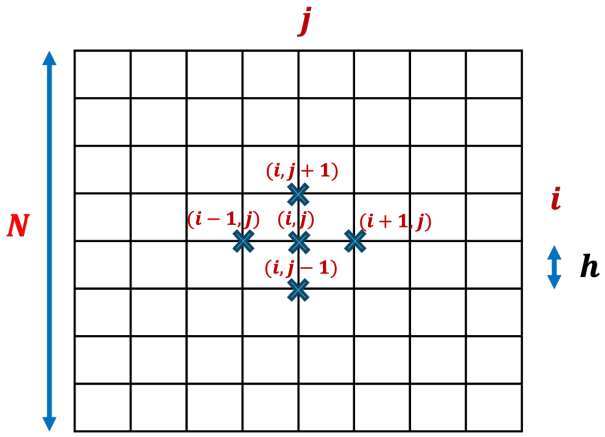



Figure 2.1: Uniform Cartesian grid

Now that we have a linear equation for each interior point, we can, with the problem as a full linear system, collect all the unknowns $u_{i,j}$ and the values on the right-hand side $f_{i,j}$ into a single vector $u$ and $f$, ordered lexicographically (i.e., first varying the index $i$). The resulting system is written in matrix form as:

$$Au = f,$$

where :

- $u$ the unknown vector of size $(N-1)^2$;
- $f$ the right-hand side.

For facilities, let us note $n = (N-1)^2$. Next, we have $A$ a sparse, symmetric, positive-definite matrix of size $n \times n$ defined as follows:

$$A = \frac{1}{h^2}\begin{bmatrix} T_n & I_n & 0 & \cdots & 0 \\ I_n & T_n & I_n & \ddots & \vdots \\ 0 & I_n & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & T_n & I_n \\ 0 & \cdots & 0 & I_n & T_n \end{bmatrix} \in \mathbb{R}^{n^2 \times n^2} \quad \text{with} \quad T_n = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 4 & -1 \\ 0 & \cdots & 0 & -1 & 4 \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Each block $T_n$ corresponds to the interaction of points within the same row of the grid, and the identity matrices $I_n$ represent couplings between neighboring rows.

## Formalization of classical solvers

Now that the problem has been introduced, we turn to the question of how to solve it numerically.

The linear system $Au = f$ has long been at the core of scientific computing, and over the years, numerous solution methods have been developed. In this section, we focus on fixed-point iterative methods, in particular the Jacobi and Gauss–Seidel algorithms.

This choice is not arbitrary. While these techniques are known to have slow convergence they have numerical features that play a fundamental role in the understanding and construction of multigrid algorithm. Although we have not yet introduced multigrid techniques directly, these solvers will provide essential insight into the mechanisms of smoothing and error propagation.

The goal of these iterative methods is to approximate the solution $u$ of the linear system $Au = f$. To do so, we start from an initial guess $u^{(0)}$ and apply the following iterative update scheme, computing the new iterate $u^{(k+1)}$ from the current one $u^k$:

$$u^{(k+1)} = u^{(k)} + M(f - Au^{(k)}) \tag{2.1}$$

where $M$ is a non singular matrix chosen to improve convergence by attenuating the error at each step. This matrix $M$ is the key factor over the method rate convergence. The best $M$ we might dream of is $M = A^{-1}$ as the iterative scheme would converge in two iterations. Of course, it is not feasible in practice but some approximation of $A^{-1}$ might considered. In the sequel we denote two different scenario:

- $M = \omega \times D^{-1}$, $0 \leq \omega \leq \frac{2}{\rho(D^{-1}A)}$ being the range for convergence[1] and $D$ the diagonal matrix of A [1];

- $M = \omega \times (L+D)^{-1}$, $0 \leq \omega \leq 2$ being the range for convergence, $D$ the diagonal matrix and $L$ the lower triangular matrix of A [2]

Although these methods are straightforward to implement without forming explicitly the inverses, they are rarely used to solve linear systems due to their slow convergence rates, as illustrated in Figure 2.2. However, they are commonly used as preconditioner in more advanced iterative schemes, where their ability to efficiently damp high-frequency error components becomes particularly valuable.

[1] We display in appendix page 54 the practical optimal value for weighted Jacobi

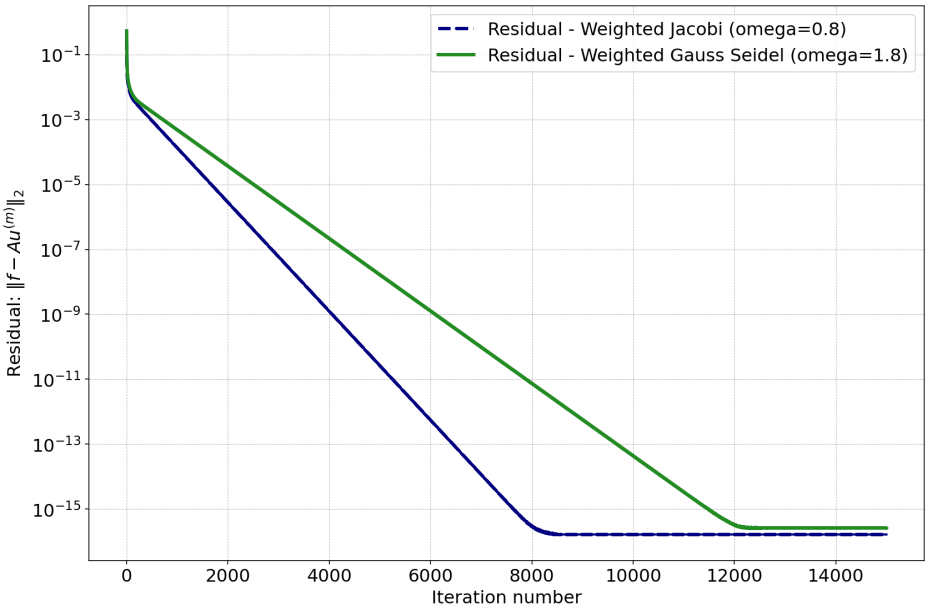



Figure 2.2: Convergence rate of Jacobi and Gauss Seidel as direct solvers

## Limitations of classical methods

To understand why these methods are not particularly effective at reducing the error, we need to revisit the expression of the error itself. At each iteration $k$, the error $e^{(k)}$ can be decomposed as a linear combination of the eigenvectors $\phi_i$ of the system matrix $A$:

$$e^{(k)} = \sum_{i=0}^{n-1} \alpha_i^k \phi_i,$$

where :

- $\phi_i$ are orthonormal eigenvectors of $A$, given by the following formula :

$$\phi_i(j) = sin\left(\frac{nik}{\pi}\right) sin\left(\frac{nj\ell}{\pi}\right), j = 1, .., n$$

- $\alpha_i^k$ represent the contribution for each eigenmode in the error, given by the following formula:

$$\alpha_i^k = \phi_i^T \left(u_{ref} - u^{(k)}\right)$$

Each eigenvectors $\phi_i$ represents a frequency component of the error. The highest index (i.e, $i$ large) corresponds to the high frequency (highly oscillating vector entries), while the lowest index (i.e.,$i$ small) corresponds to the low frequency (smooth vector entries).

What has been shown through the years is that the fixed-point methods, such as the one presented above, are very effective to reduce the high frequency errors, while being poor on the smooth ones.

A way to realize this phenomenon, is to look at how $a_i^k$ evolve along the iteration. In order to do that, we start with a random vector, that will be our true solution (i.e, $u_{ref}$), we compute the right-hand side by multiply A and $u_{ref}$, and select the initial guess $u^{(0)} = 0$.

Then, we apply both Jacobi and Gauss Seidel iterative methods, with they optimal weight. We report in Figure 2.3 the convergence history of the error correspondents.

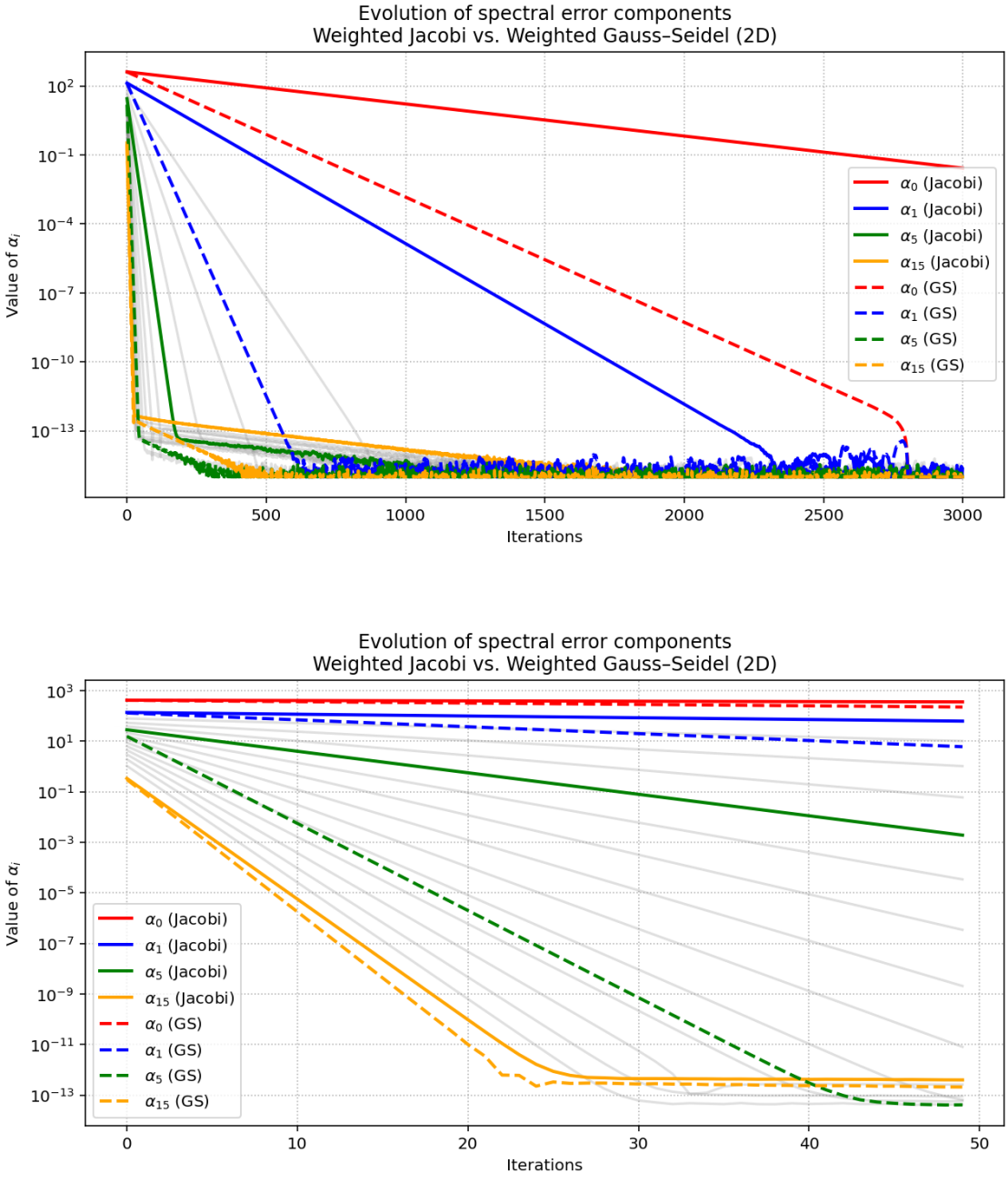



Figure 2.3: Illustration of the spectral error components through four index, with an overall overview (figure at the top) and a zoom on the fifty first iterations (figure at the bottom)

These two graphs clearly illustrate the behavior described above. High-frequency components of the error — corresponding to large values of $i$ — are rapidly damped, often reaching machine precision within 20 to 40 iterations. In contrast, low-frequency components (small $i$) converge much more slowly, requiring up to 3000 iterations to achieve the same level of accuracy.

Thus, after a few iterations, the error vector $e^{(k)}$ has its high-frequency components effectively eliminated, while low-frequency components persist. This observation is precisely what motivated the development of multigrid methods. The key idea behind multigrid is to transfer the error equation, $Ae^{(k)} = r^k = f - Au^{(k)}$ problem to a coarser grid, where the remaining low-frequency errors appear as higher-frequency components. These transformed components can then be efficiently reduced using the same fixed-point methods, which are more effective at attenuating high frequencies.

### 2.1.2 Multigrid and reformulation using convolutions

#### Formalization of the multigrid solver

In this section we focus on the multigrid method, and more specially on the V-Cycle [2]. The multirgrid V-Cycle uses a concept of level in a grid hierarchy, where each level corresponds to a specific mesh size. We start by the finest one (level $L$) down to the coarsest (level 0). The algorithm can be written as follows:

**Algorithm 1:** Multigrid V-cycle

Apply $\mu_1$ **pre-smoothing steps** on the current grid.

Compute the **residual** of the current approximation.

**Restrict** the residual and operator $A$ to the coarser grid.

**if** *the current level is the coarsest grid* **then**
    **Solve** the error equation directly.
**else**
    Recursively call the V-cycle on the next coarser grid.

**Interpolate** the correction back to the fine grid.

**Correct** the solution: update $u \leftarrow u +$ correction.

Apply $\mu_2$ **post-smoothing steps**.

[2] We present here the V-Cycle, the most used algorithm in the litterature, and the one I used for my intership. See [3] for more variant

The structure of the algorithm can also be illustrated graphically on a 2 level example, which helps to clarify the origin of the term "V-Cycle."

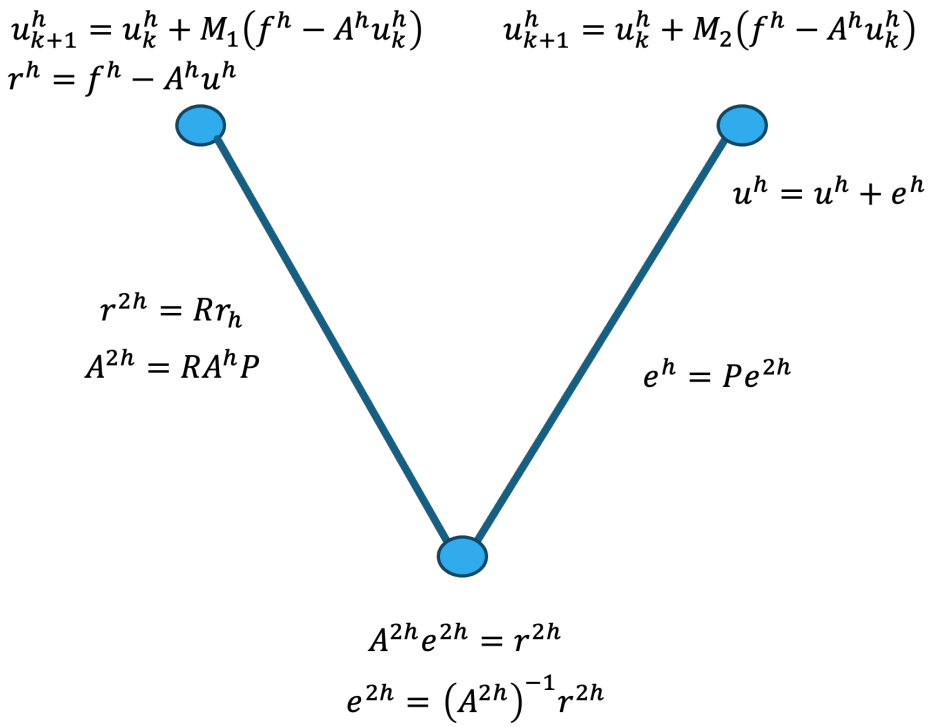



Figure 2.4: Illustration of the V-Cycle algorithm.

Having introduced the overall structure of the V-cycle through Algorithm 1 and its schematic illustration in Figure 2.4, we now proceed to describe each step of the method in more details.

In particular, we aim to formalize how the multigrid operations are carried out in the standard linear algebraic framework — using matrix-vector products and explicitly defined operators at each level. This will not only provide a precise understanding of the algorithm's mathematical foundation, but also prepare the ground for its reformulation using convolutions.

**Step 1 and 10: Applying pre and post smoothing steps**

The steps 1 and 10 are correlated, as we applied on both smoothing steps. The numbers of steps are defined a priori, usually between 1 and 2. We denote $\nu_1$, the number of steps for the pre-smoothing and $\nu_2$, the number of steps for the post-smoothing.

We use the fixed point methods presented before, either weighted Jacobi or Gauss-Seidel. As mentioned earlier, these two methods have a different optimal weight, weight that will be the same for all levels.

**Step 2: Computation of the residual**

Two mains measures are defined of the deviation of an observed value from its "true value", the error and the residual. In the case of the multigrid algorithm, where the goal is to solve a PDE, the error is the difference between the true solution and the approximation (i.e, $e = u_{ref} - \tilde{u}$), and the residual the difference between the right-hand side and the result of plugging your approximation into the system ($r = f - A\tilde{u}$).

This two quantities are related, indeed, by combining both definitions:

$$Ae = A(u - \tilde{u}) = Au - A\tilde{u} = f - A\tilde{u} = r \Rightarrow Ae = r,$$

However, we prefer to use the residual, since we don't need the exact solution, as is usually unknown.

**Step 3: Restrict the residual and operator A:**

Restriction appears when we have to bring the data from the fine grid to the coarse grid (during the descent phase of the V-Cycle). A way to understand it better is to show an example of a restriction in 1D:

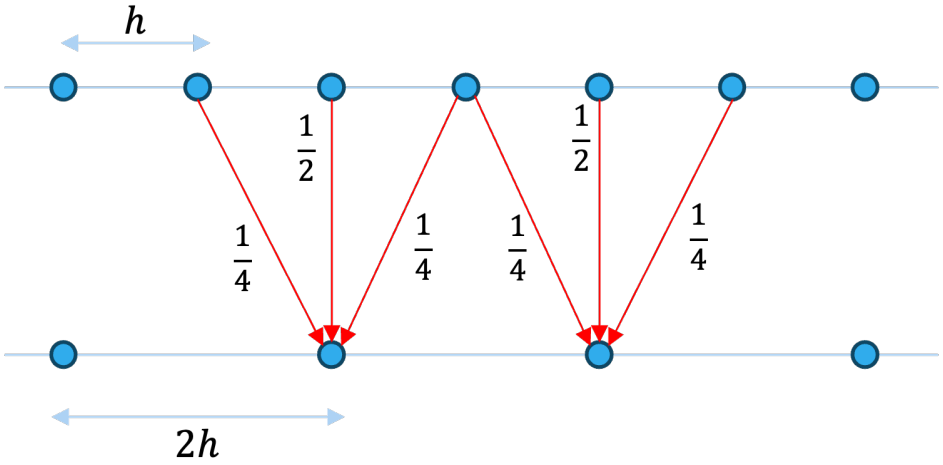



Figure 2.5: 1D full-weighting restriction.

In 1D, the full-weighting restriction operator uses a simple 3-point stencil to aggregate fine grid values into a coarse one. It is defined as:

$$\frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix},$$

Which leads to the following matrix:

$$R^{(1d)} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 2 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 2 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & 2 & 1 \end{bmatrix}$$

This representation means that each coarse point is obtained by a weighted average of three consecutive fine points:

$$u_i^{2h} = \frac{1}{4}\left(u_{2i-1}^h + 2u_{2i}^h + u_{2i+1}^h\right).$$

To extend this idea to 2D, we apply the same principle in both directions, which conduct to a $3 \times 3$ stencil obtained as the tensor product of the 1D stencil with itself:

$$K_R = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

Each coarse-grid value is computed as a weighted average over a $3 \times 3$ block of fine-grid points, centered around the corresponding coarse location. Graphically, this leads to the following illustration:

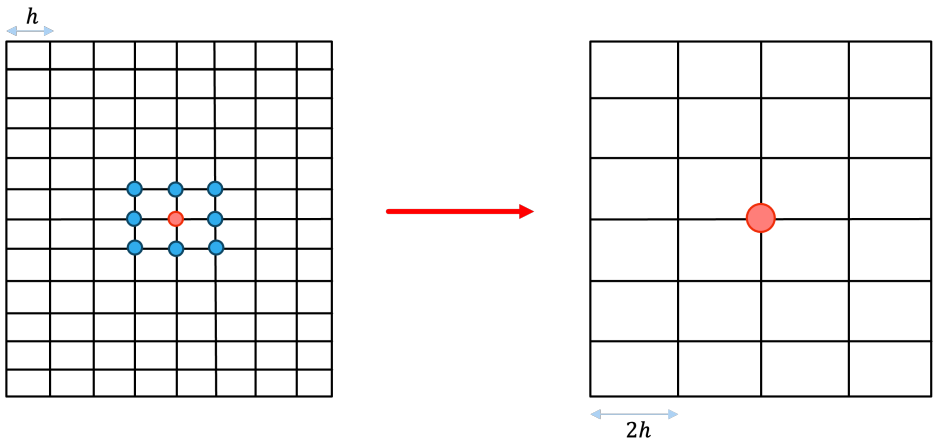



Figure 2.6: 2D full-weighting restriction.

While we use the matrix representation in 1D (meaning $R^{(1d)}$), we prefer to use a stencil when we increase the dimension, as the matrix becomes too large to have a clear representation. We will see further how to restrict the residual using the stencil.

**Step 4 to 7: Solve or recursively called**

These steps are the key components of the multigrid algorithm, where we either solve the error equation directly if the coarsest grid is reached or we recursively apply the V-Cycle and we start at step 1.

For the solving part, we can use a direct solver due to the fact that we are on the coarsest grid. Indeed, the size of the matrix $A$ are usually between 16 and 32 on that grid, which makes the direct solution easy to compute.

**Step 8: Interpolating the correction**

Let's now focus on the study of the prolongation operator. At the opposite of restriction, the prolongation goes from a coarse grid to the fine one (during the ascent phase). As for the restriction, we shown an example in 1D, for a better understanding:

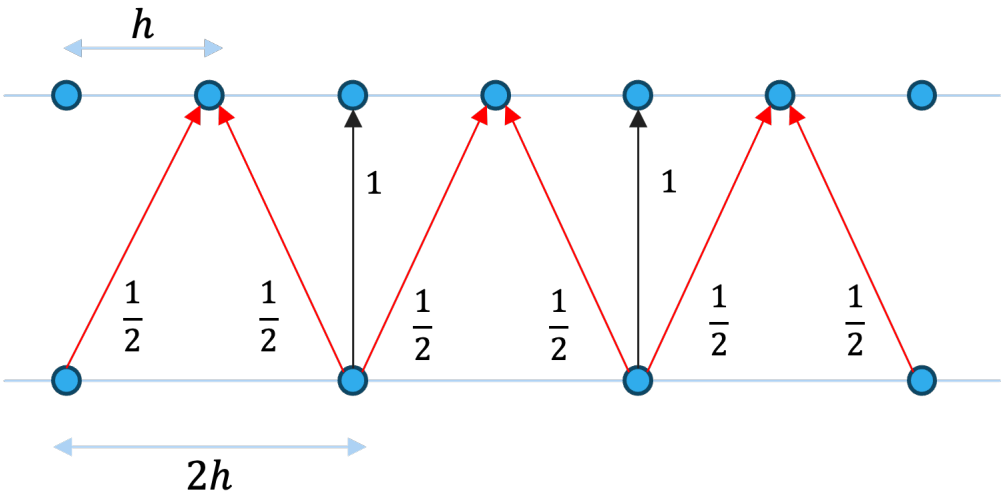



Figure 2.7: 1D linear interpolation.

For the prolongation, we used the linear interpolation operator that uses a simple 3-point stencil to aggregate coarse grid values into a fine one. It is defined as:

$$\frac{1}{2}\begin{bmatrix}1\\2\\1\end{bmatrix}$$

And the matrix associated $P^{(1D)}$:

$$P^{(1D)} = \frac{1}{2}\begin{bmatrix} 1 & & & & & \\ 2 & & & & & \\ 1 & 1 & & & & \\ & 2 & & & & \\ & 1 & \ddots & & & \\ & & \ddots & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & 2 & & \\ & & & 1 & & \end{bmatrix}.$$

This representation means that each fine point is obtained by a either a weighted average of two consecutive fine points or by the same point that belongs to the two succesive grids:

$$u_i^h = \begin{cases} u_{\frac{i}{2}}^{2h} & i \text{ even}, \\ u_{\frac{i-1}{2}}^{2h} + u_{\frac{i+1}{2}}^{2h} & \text{i odd}. \end{cases}$$

Again, to extend this idea to 2D, we apply the same principle in both directions, this leads to the following $3 \times 3$ stencil:

$$K_P = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}.$$

We can illustrate the stencil by:

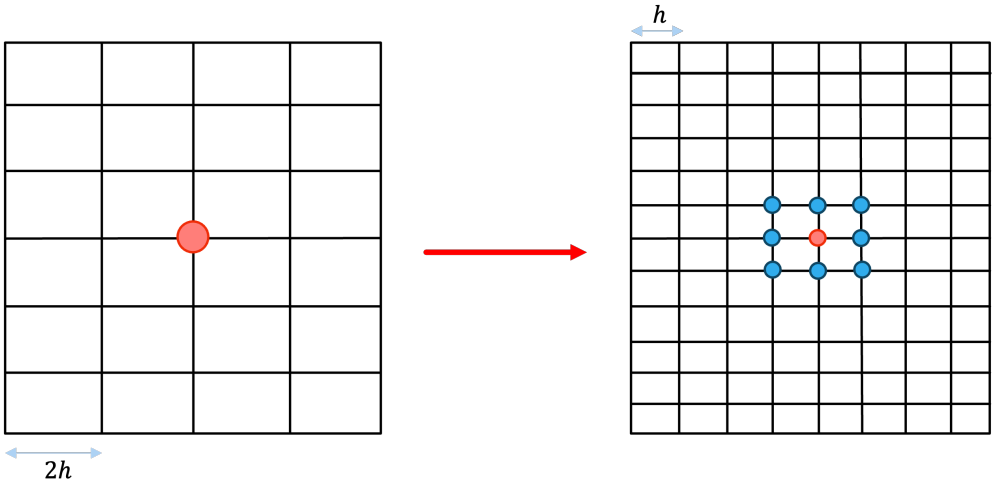



Figure 2.8: 2D linear interpolation.

Again, we prefer to use the stencil for 2D computation, explained more in details further down.

While this linear algebra formulation is standard and theoretically grounded, it comes with significant practical limitations. In particular, explicitly storing and multiplying large sparse matrices can be costly on high-resolution grids. Moreover, as we seen, the multigrid algorithm revolves around applying operators like $A$, restriction $R$, prolongation $P$, and smoothing $M$ to update or transfer information across the grid levels.

A crucial observation is that these operators are all local: they act on each grid point using information from a close neighborhood. We had a preview of this locality, especially when we talked about the restriction and prolongation operator, but what is important now is that we can extend this stencil representation to all steps of the algorithm 1.

This insight opens the door to a reformulation of the multigrid method using convolutional operators instead of sparse matrix multiplications. Not only does this provide computational advantages (especially on GPUs), but it also allows for learning optimized kernels to accelerate convergence.

We are now going to describe how each step of the multigrid algorithm — residual computation, smoothing, restriction, and prolongation — can be expressed using convolutions and stencils.

## Reformulation using convolutions

### Residual:

We recall the expression of the residual:

$$r^k = f - Au^{(k)} \tag{2.2}$$

Here, we tend to replace the product $Au^{(k)}$, using the locality of the operator A. Indeed, as we said earlier, the Laplacian at a point is approximated by a weighted combination of its four neighbors and itself, referring as to five-point stencil, that we note $K_A$:

$$K_A = \frac{1}{h^2} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

Then, equation (2.2) can be expressed as:

$$r = f - \tilde{u} * K_A,$$

with $*$ the convolution operator[3].

[3]When using a stencil in a convolution operator, we prefer to denote this previous as a **convolution kernel**. Otherwise, when it represent the finite difference operator, we denote it as **stencil**.

**Restriction and prolongation operator:**

We already seen the stencil representation of the restriction and prolongation. We can now replace both restriction of the residual $r^h$ and prolongation of the correction $e^{2h}$ steps using convolution:

$$r^{2h} = Rr^h \Rightarrow r^{2h} = r^h * K_R$$

and

$$e^h = Pe^{2h} \Rightarrow e^h = e^{2h} * K_P.$$

To summarize, we have successfully reformulated the key multigrid operators — namely the system matrix $A$, the restriction operator $R$, and the prolongation operator $P$ — as local convolutional stencils. However, one crucial operator remains: the smoother $M$. Can it also be expressed as a convolution?

This question is particularly important, as the smoother plays a central role in reducing high-frequency errors at each level of the multigrid hierarchy. In the following section, we explore how this reformulation can be achieved only in the case of Jacobi iterations, where $M$ retains the locality required for a stencil-based interpretation.

## Classical smoother as a convolution: the Jacobi case

As we said before the reformulation of the smoother M as a convolutional operator is only applicable in the case of Jacobi-type iterations, where $M = D^{-1}$. In contrast, the Gauss–Seidel method involves solving triangular systems through $M = (L+D)^{-1}$, which introduces a global coupling between variables. As a result, this operator is no longer local and cannot be represented as a convolution.

Thus, in the case of Jacobi-type iterations, what should be the structure and expression of this kernel? To find it, we start by the expression of $D$, the diagonal matrix of $A$:

$$D = \mathrm{diag}(A) = \frac{1}{h^2}\begin{bmatrix} 4 & 0 & 0 & \dots & 0 \\ 0 & 4 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \dots & 4 \end{bmatrix}$$

which leads to:

$$D^{-1} = h^2\begin{bmatrix} \frac{1}{4} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{4} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \dots & \frac{1}{4} \end{bmatrix}.$$

The matrix $M$, can then be represented by the following $3 \times 3$[4] stencil:

$$K_M = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \omega\frac{h^2}{4} & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

As we are focusing on the weight, we can take out the parameter $\frac{h^2}{4}$, which leads to:

$$u^{(k+1)} = u^{(k)} + \frac{h^2}{4}(f - u^{(k)} * K_A) * K_M. \tag{2.3}$$

Having established that the smoothing operator $M$ can be reinterpreted as a local convolution kernel $K_M$, we now turn to the core idea of this work: replacing the hand-crafted stencil with a data-driven one.

Rather than fixing the weights of the stencil analytically, we propose to learn them directly from data, using a training procedure that optimizes convergence. This leads to a fully adaptive smoother that maintains locality and efficiency while being tailored to the problem at hand.

[4]The stencil can as well be represented by a $1 \times 1$ stencil, we will see further down why this size

The following section details the learning strategy and implementation used to train $K_M^\ell$ at each level of the multigrid hierarchy.

### 2.1.3 Learning and implementation

#### Learning a Data-Driven Smoother

As discussed before, we propose to learn a *convolutional stencil* — denoted $K_M^\ell$ at level $\ell$ — that serves the role of the smoother. This learned kernel replaces the classical weights with trainable parameters, optimized to accelerate convergence while preserving the overall structure of the multigrid algorithm.

The result is a new operator $K_M$, interpreted as a lightweight, local convolutional filter. It is optimized via gradient descent over many training examples and is specifically tailored to enhance high-frequency error reduction — one of the most critical aspects of efficient multigrid methods.

We explore several strategies for parameterizing the smoothing stencil $K_M^\ell$, each corresponding to a different level of flexibility in the learning process.

**Level-wise weighted Jacobi.** In the first approach, we aim to improve the classical Jacobi method by learning an optimal relaxation weight for each level of the multigrid hierarchy. Rather than using a fixed scalar weight across all levels, a value that corresponds to the optimal smoothing factor for classical Jacobi relaxation (as shown in Appendix A), we introduce a distinct, trainable scalar per level, denoted $\omega^{(l)}$, where $\ell$ refers to the level index.

This results in a level-wise weighted Jacobi smoother that preserves the simplicity and locality of the original method while enabling data-driven adaptation at each resolution. We denoted the kernel as $K_\omega^{(l)}$:

$$K_\omega^{(\ell)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \omega^{(\ell)} & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The goal of this approach is to enhance convergence speed beyond that of standard Jacobi smoothing, without significantly increasing computational cost or model complexity. As we will see in the following experiments, learning per-level weights can yield faster convergence and better residual reduction compared to fixed-weight Jacobi, and serves as a solid foundation for more flexible strategies.

**Learning a Fully Adaptive Smoother.** In the second approach, we go beyond Jacobi-type smoothers. We now treat all weights in a $3 \times 3$ convolution kernel as trainable parameters, enabling the model to discover an entirely new smoothing pattern, optimized for convergence and adapted to the multigrid framework. Formally, at each level $\ell$ we learn a convolutional kernel $K_M^{(\ell)}$ of the form:

$$K_M^{\ell} = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \\ \omega_4 & \omega_5 & \omega_6 \\ \omega_7 & \omega_8 & \omega_9 \end{bmatrix}.$$

This technique allows the model to also take into account the neighboring values (above, below, left, right and diagonals), he doesn't just consider how far one point is from the solution, but how its local neighborhood behaves.

However, this comes at a cost: this representation has more parameter (9 vs 1), which leads to a slower convergence, especially when we increase the number of V-Cycles.

**Generalization learning.** As we have, in the previous strategies, one stencil per level, the generalization of our learned stencils to further levels is not possible. This reflection leads to another study case: one stencil for the descent phase and one for the ascent phase.

# 2.2 Training strategy

## 2.2.1 Loss function

In this section, we are going to focus on the formulation of the learning, and specifically describe both datasets and more importantly the loss function that should be minimized during the training process.

For the dataset, we generate the random solution vector $u_{ref}$ following an uniform law for its entries, and with this solution, we compute the right-hand side by: $f = Au_{ref}$, generated once for a given grid size $N$.

To train the multigrid solver, we define a composite loss function that accumulates contributions of the first subsequent iterates $u^{(\ell)}$ computed by the algorithm. In particular, we monitor the residual at each of the first iterations of the multigrid cycle, and weight them according to balance their contribution to the loss function to minimize.

In our approach, we define two distinct types of loss functions, depending on the trainning objective and the available reference data: the error-based loss $L_{err}$ and the residual-based loss $L_{res}$.

**The error-based loss:** This loss relies on the availability of a reference solution $u_{ref}$. In our case, as we generate randomly the true solutions, we can easily compute the error, representing the distance between the current approximation $u^{(m)}$ and the reference solution $u_{ref}$.

We define the loss function as follows:

$$L^{err} = \sum_{i=1}^{m} \beta_i^{err} \frac{\|u_{ref} - u^{(i)}\|_2}{\|u_{ref}\|_2}$$

where $\beta_i^{err} \in \mathbb{R}_+$ are scalar weights that are crucial for balancing the contribution of each term in the sum, ensuring that all individual errors contribute equally to the overall objective function.

They are defined in a pre-processing phase from a first random solution $u_{ref}$

$$\begin{cases} \beta_1^{err} = 1 & \text{if} \quad i = 1 \\ \beta_i = \dfrac{\|u_{ref} - u^{(1)}\|_2}{\|u_{ref} - u^{(i)}\|_2} & \text{if} \quad 2 \leq i \leq m \end{cases}$$

**The residual-based loss.** We can although consider another loss function that does not relies on the ground truth solution but rather on the scaled norms of the residual associated with he sequence of iterates defined as $\|f^i - Au^{(i)}\|$. In that context the loss function is defined as follows:

$$L^{res} = \sum_{i=1}^{m} \beta_i^{res} \frac{\|f^\star - Au^{(i)}\|_2}{\|f^\star\|_2} \tag{2.4}$$

where $\beta_i^{res} \in \mathbb{R}_+$ are defined in the same line as before, that is,

$$\begin{cases} \beta_1^{res} = 1 & \text{if} \quad i = 1 \\ \beta_i^{res} = \dfrac{\|f^\star - Au^{(1)}\|_2}{\|f^\star - Au^{(i)}\|_2} & \text{if} \quad 2 \leq i \leq m \end{cases}$$

with $f^\star$ a first random right-hand side used in a processing phase to compute the weights.

Using a weighted sum over multiple cycles rather than only the final one has a practical benefit: it encourages the model to perform well throughout the cycle sequence, not just at the end. This prevents overfitting to the first few iterations or relying solely on long-term convergence. In experiments, this also helps avoid plateaus: if the loss stagnates too early, it may indicate that the learned smoother is not effective across the whole range of iterations.

### 2.2.2 Implementation and framework

The implementation of the framework will be using PyTorch Lightning. This open-souce Python library provides a high-level interface for Pytorch, that organizes Pytorch code to decouple research from engineering, thus making deep learning experiments easier to read and reproduce.

All experiments were conducted on Kraken, the in-house GPU cluster of Cerfacs. This high-performance computing environment provides access to modern NVIDIA GPUs, which allow us to efficiently train our convolution-based smoothers and perform large-scale multi-grid simulations.

The use of GPU acceleration is particularly beneficial for handling the large number of batched operations involved in training and inference, significantly reducing computation time compared to CPU-based implementations.

## 2.3 Experimental results and evaluation

This second part refers to the experimental results. It starts with individual learned kernels for each level, followed by a kernel for the descent and a kernel for the ascent, which are shared by all the levels. For all the numerical experiments we use $\mu_1 = \mu_2 = 1$ as number of pre and post-smoothing iteration.

### 2.3.1 Experiments with distinct convolution kernels per level

In this section, we report numerical experiments in which a distinct convolution kernel is learned for each level of the multigrid hierarchy. Specifically, a $1 \times 1$ kernel corresponds to a level-wise optimized Jacobi smoother, while a $3 \times 3$ kernel enables learning a more general, spatially localized, level-dependent smoother.

#### Experimental parameters

Before turning to the actual experiments, we need to define the key parameters and notations used throughout the different configurations studied:

- $N$ is the **1D grid resolution** on the finest level, i.e., the number of interior points along each spatial dimension. The total number of grid points associated with unknowns (excluding boundaries) is thus $(N-1)^2$;
- $L$ represents the **number of multigrid levels**, including the finest grid at level $\ell = 0$ and coarser grids down to the coarsest level $\ell = L - 1$;
- $K^{\ell}_{\omega\downarrow}$ and $K^{\ell}_{\omega\uparrow}$ denote the **learned relaxed Jacobi smoothers** respectively used during the **descent** and the **ascent** of the V-cycle at level $\ell$. These act as data-driven analogues of the weighted Jacobi method in which the optimal weight is computed analitically [4];
- $K^{\ell}_{M\downarrow}$ and $K^{\ell}_{M\uparrow}$ refer to the **full convolutional smoothers** (i.e., $3 \times 3$ kernel), also learned separately for the descent and ascent phases at level $\ell$. These operators replace the traditional fixed kernel by learned local convolution masks and allow for a richer spatial interaction;
- $m$ denotes the numbers of multigrid cycles made for the training. It's the number of time the V-Cycle is repeated, the larger is $m$, the most accurate is the solution $u^{(m)}$, refers to Figure 2.9 for better understanding. Even though it represents the same quantity, we use the word "cycle", when we refer to the training part, and "iteration", when we run the inference.
- $\nu_1$ and $\nu_2$ representing respectively the number of pre-smoothing and post-smoothings steps.

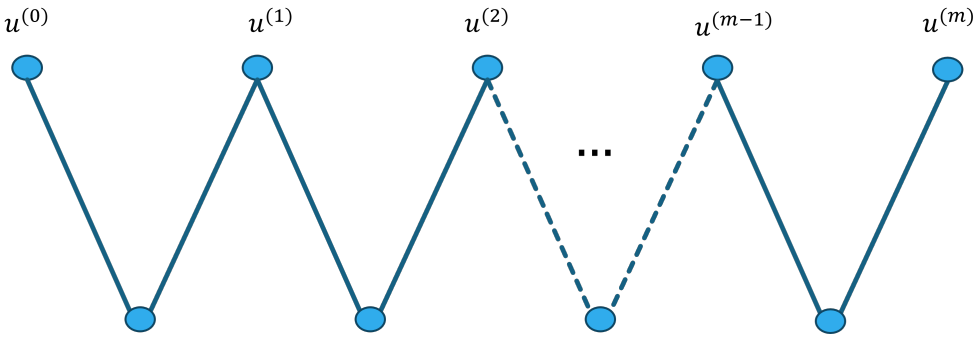



Figure 2.9: Illustration of the $m$ - We compute here $m$ multigrid cycles

For the following experiments and different strategies, we keep the same learning parameters, $\boldsymbol{N = 64,\ L = 4}$ **and** $\boldsymbol{\nu_1 = \nu_2 = 1}$ , which are the fixed parameters and for the learned smoothers and the parameter $m$, they will depend on the strategies used.

## Level-wise weighted Jacobi

The level-wise weighted Jacobi learning is the simplest strategy that we consider, but despite this easiness, it is a good starting point to understand the role of the weighted factor.

Indeed, in multigrid method using weighted Jacobi as the smoother, the weight is usually fixed for all levels, set to 0.8 (for Jacobi) and 1.9 (for Gauss-Seidel), denote as analytical weight. The core goal with this method, is to see how far we get from this optimal value, in a level-free scenario. Let us start with the first test.

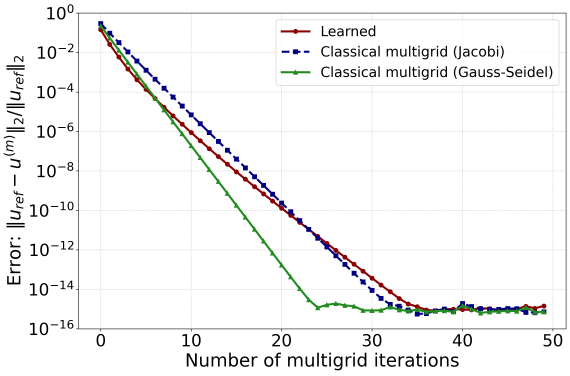



Figure 2.10: Learned $K_\omega^\ell$, with $m = 1$ and the error-based loss

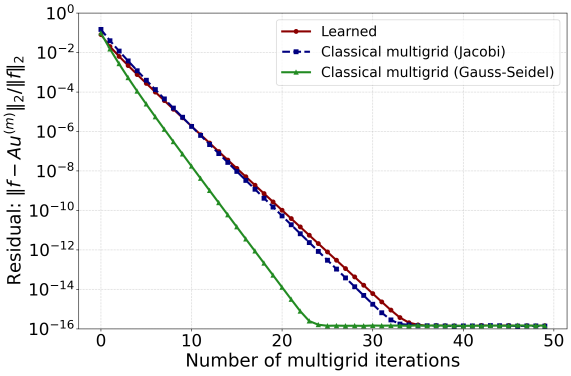



Figure 2.11: Learned $K_\omega^\ell$, with $m = 1$ and the residual-based loss

For the rest of this report, we will have:

- Classical Jacobi - Blue Curve (Using the optimal analytical weight, i.e, $\omega = \frac{4}{5}$ [4])
- Classical Gauss-Seidel - Green Curve (Using the optimal analytical weight, i.e, $\omega = 1.9$)
- Learned - Red Curve (Using the learned convolutional smoother $K_\omega^\ell$)

In Figure 2.10 and 2.11, we can remark that the learned smoothers perform best in the early iterations, outperforming the Jacobi method during that time, being very distinct in the error-based loss. However, the curve do not keep the same rate until the end, and finish by being slower than the two other methods.

These results can be easily explained, as we learn for $m = 1$, meaning that we only learn for and from the first cycle from the multigrid algorithm, which leads the kernel to be effective just in the early iterations. An idea to overcome this problem is to let the model learn from more cycles, i.e, $m > 1$.

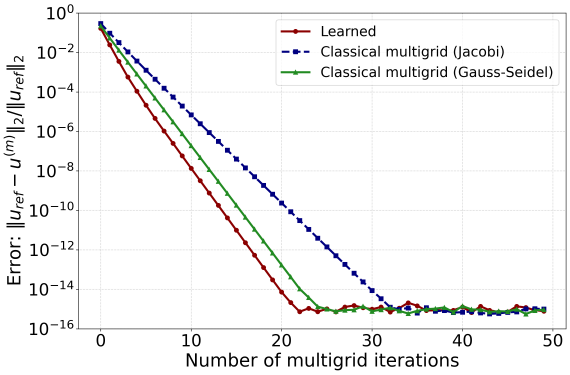



Figure 2.12: Learned of $K_\omega^\ell$, with $m = 5$ and the error-based loss

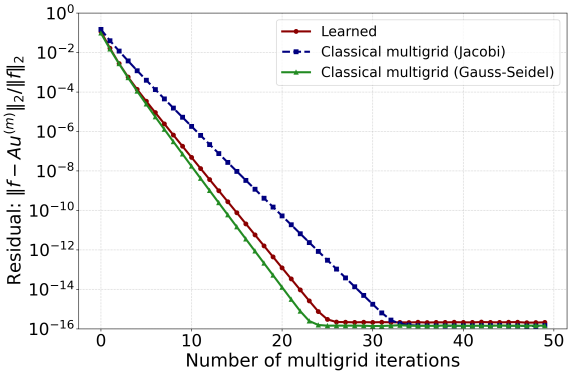



Figure 2.13: Learned of $K_\omega^\ell$, with $m = 5$ and the residual-based loss

In Figure 2.12 and 2.13, $m = 5$ is used for the training and the learned smoothers perform very well through all the iterations, successfully outperforming the analytical optimal Jacobi. However, stays slower than Gauss-Seidel method.

At this point we already have a first success: our learned kernels $K_\omega^\ell$ converge faster than the classical Jacobi methods, but still slower than the Gauss-Seidel ones. In order to improve the situation we might let the model learn on even more cycles. This is what is reported in Figure 2.14 and 2.15, where we set $m = 10$ for the training.

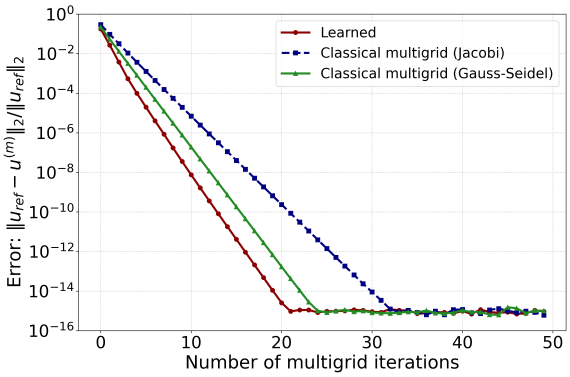



Figure 2.14: Learned of $K_\omega^\ell$, with $m = 10$ and the error-based loss

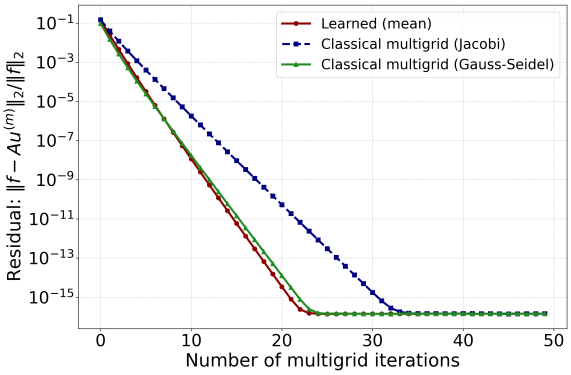



Figure 2.15: Learned of $K_\omega^\ell$, with $m = 10$ and the residual-based loss

Here we remark that the two sets of graphs for $m = 10$ are essentially the same than for $m = 5$; we do not have any upgrade between them two curves. At $m = 5$, the gradient of the loss function with respect to the stencil parameters is already very low due to the fast convergence rate of the multigrid method, which means that the code will not be able to optimize even more. The more we increase it, the more we have a low gradient.

Increasing even more the numbers of cycle $m$ in the definition of the loss function will then have no impact on the convergence rate of multigrid equipped with our learned kernelsbut will increase their training costs. We can here conclude about the first step: center-only learning. The goal of this step was to upgrade the classical Jacobi method, used with one fixed weight along the levels, and we depict in Table 2.1, the learned relaxation parameters on the four levels

| | Level $\ell =$ | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| Descent | 2.20 | 0.41 | 0.70 | 1.35 |
| Ascent | 0.73 | 1.69 | 1.80 | 0.57 |

Table 2.1: Learned Jacobi relaxation parameters

**Conclusion.** We proposed an approach that enables to find the best weight for each level. Doing that, we already have a faster convergence than the classical multigrid with analytic optimal relaxed Jacobi as a smoother. Our learned smoothers lead to convergence that are very similar to the one of multigrid using relaxed Gauss-Seidel, but has the advantage of being less computing demanding.

However we face two questions. Can we learned smoothers that enable to outperform Gauss-Seidel? We will see how the full-kernel learning address this challenge. The second concern is related to the generalization. Indeed, we train our weights for a specific number of levels ($= 4$), which leads to an impossibility to increase the numbers of them if one want to solve larger Poisson problems.

## Learning a Fully Adaptive Smoother

Let us first start by the more flexible approach, where we learn a fully adaptive smoother, to evaluate whether it yields a tangible improvement in convergence rate. We recall the expression of the kernel $K_M^\ell$, where each weight of the kernel at each level of the multigrid hierarchy is a trainable parameter:

$$K_M^\ell = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \\ \omega_4 & \omega_5 & \omega_6 \\ \omega_7 & \omega_8 & \omega_9 \end{bmatrix}.$$

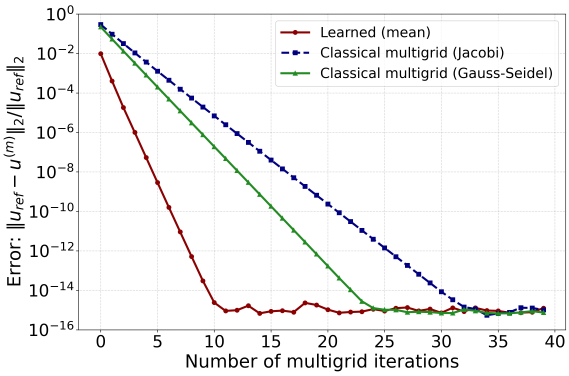



Figure 2.16: Learned $K_M^\ell$, with $m = 1$ and the error-based loss

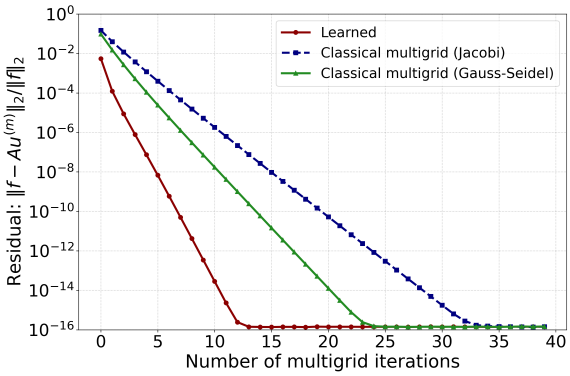



Figure 2.17: Learned $K_M^\ell$, with $m = 1$ and the residual-based loss

We display in Figure 2.16 and 2.17 the convergence history of multigrid with the learned smothers ($3 \times 3$ kernels) using $m = 1$ in the definition the loss function. It can be seen that they already outperform both optimal Jacobi and optimal Gauss-Seidel smoothers, reaching the machine accuracy with less than 15 iterations.

We report in Figure 2.18 and 2.19 (Figure 2.20 and 2.21) the performances of mutigrid with the trained smoothers for $m = 5$ (respectively $m = 10$). We can observe that the parameter $m$ does not have as similar impact as for trained Jacobi. The value $m = 1$ already gives good performance that are not improved by considering larger values. These latter are more expensive to train.

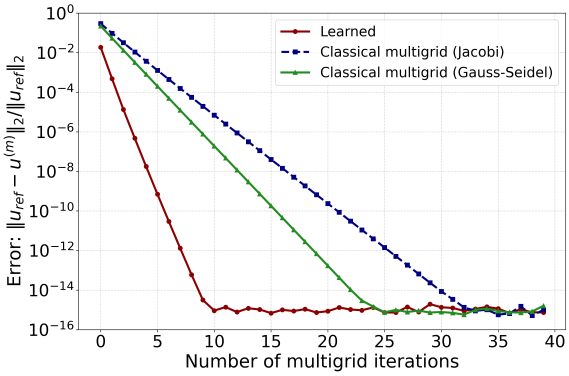



Figure 2.18: Learned of $K_M^\ell$, with $m = 5$ and the error-based loss

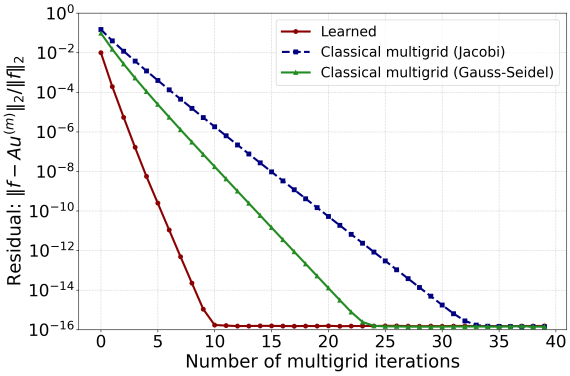



Figure 2.19: Learned of $K_M^\ell$, with $m = 5$ and the residual-based loss

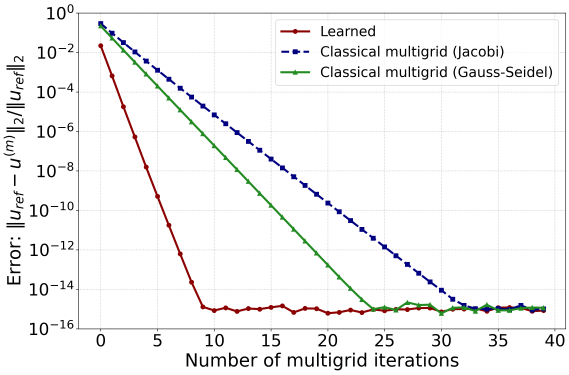



Figure 2.20: Learned of $K_M^\ell$, with $m = 10$ and the error-based loss

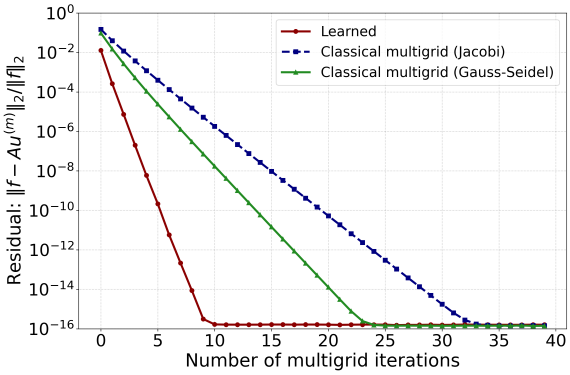



Figure 2.21: Learned of $K_M^\ell$, with $m = 10$ and the residual-based loss

Below, we plot the learned kernels that we obtained for the convergence history displayed in Figure 2.19:

| | **Descent phase** | **Ascent phase** |
|---|---|---|
| $K_M^{(4)} =$ | $\begin{bmatrix} 0.011 & 0.069 & 0.011 \\ 0.069 & 0.296 & 0.069 \\ 0.011 & 0.069 & 0.011 \end{bmatrix}$ | $\begin{bmatrix} 0.053 & 0.096 & 0.053 \\ 0.096 & 0.303 & 0.096 \\ 0.053 & 0.096 & 0.053 \end{bmatrix}$ |
| $K_M^{(3)} =$ | $\begin{bmatrix} -0.052 & 0.145 & -0.052 \\ 0.145 & 0.132 & 0.145 \\ -0.052 & 0.145 & -0.052 \end{bmatrix}$ | $\begin{bmatrix} 0.196 & 0.266 & 0.196 \\ 0.266 & 0.466 & 0.266 \\ 0.196 & 0.266 & 0.196 \end{bmatrix}$ |
| $K_M^{(2)} =$ | $\begin{bmatrix} 0.007 & 0.119 & 0.007 \\ 0.119 & 0.231 & 0.119 \\ 0.007 & 0.119 & 0.007 \end{bmatrix}$ | $\begin{bmatrix} 0.062 & 0.184 & 0.062 \\ 0.184 & 0.447 & 0.184 \\ 0.062 & 0.184 & 0.062 \end{bmatrix}$ |
| $K_M^{(1)} =$ | $\begin{bmatrix} -0.073 & 0.254 & -0.073 \\ 0.254 & -0.019 & 0.254 \\ -0.073 & 0.254 & -0.072 \end{bmatrix}$ | $\begin{bmatrix} 0.492 & 0.535 & 0.492 \\ 0.534 & 0.465 & 0.534 \\ 0.494 & 0.534 & 0.493 \end{bmatrix}$ |

Figure 2.22: Learned stencils $K_M^{(\ell)}$ from level 4 to level 1. Left: descent phase; Right: ascent phase.

**Conclusion.** Even thought the difference between $m = 5$ and $m = 10$ is not flagrant, we have in Figure 2.20 and Figure 2.21 the best convergence rate. Moreover, the learned stencils are all symmetric and positive definite, keeping the same property as classical smoother. They all also very different, with no obvious trends emerging, i.e, Figure 2.22.

However, this fast convergence has no utility if we cannot use it on different meshes. Indeed, the optimization part takes quite a while, and we cannot afford to repeat this optimization on larger meshes. The idea is now to optimize not one kernel per level but one for the ascent and descent phase, in order to generalize any mesh size.

### 2.3.2 Generalized approach

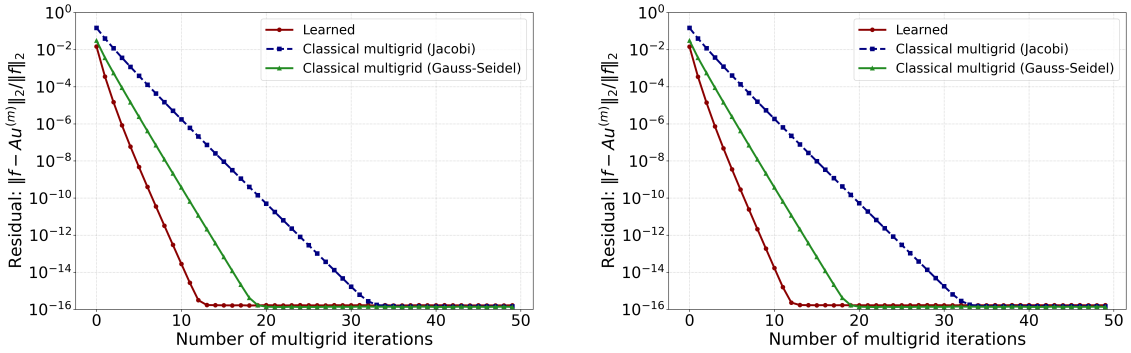
As mentioned previously, we have a fixed number of kernels, one per level. In our case, the training has been made using the following parameters: $N = 64$ and $L = 4$. Meaning, even if we can increase the mesh size, we cannot do the same with the number of levels $L$.

The following section will focus on the generalization over larger mesh sizes and level numbers. We start by training a kernel for both down and up sweep, by finishing on a overall kernel. The training is performed with $N = 64$, $L = 4$ and $m = 5$ and the loss function based on the residual norm.

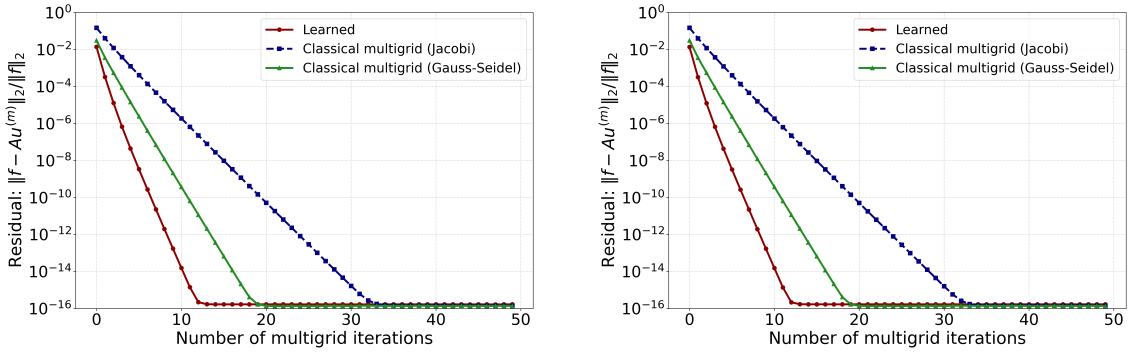
We refer to the learned kernel as $K_{M,\downarrow}$ and $K_{M,\uparrow}$, and we display their values below:

$$K_{M,\downarrow} = \begin{bmatrix} 0.010 & 0.056 & 0.021 \\ 0.056 & 0.281 & 0.056 \\ 0.021 & 0.056 & red0.010 \end{bmatrix} \qquad K_{M,\uparrow} = \begin{bmatrix} 0.029 & 0.068 & 0.011 \\ 0.068 & 0.36 & 0.068 \\ 0.011 & 0.068 & 0.029 \end{bmatrix} \tag{2.5}$$

These two kernels have the same structure as before, with the highest value at the center, lower in the four directions, and even smaller in both diagonals. The graphs associated with these kernels are depicted in Figure 2.23. We recall that we train for the kernels for $N = 64$, $L = 4$ and $m = 5$, and then generalize to $N = 128/L = 5$, $N = 256/L = 6$ and $N = 512/L = 6$.




(a) $N = 64$ and $L = 4$ (b) $N = 128$ and $L = 5$




(c) $N = 256$ and $L = 5$ (d) $N = 512$ and $L = 6$

Figure 2.23: Performance with two learned smoothers $K_{M,\uparrow}$ and $K_{M,\downarrow}$ trained using $N = 64$, $L = 4$ and $m = 5$

**Conclusion.** Two main observations can be made. First, as it could have been expected, the convergence rate is slightly slower than in the previous cases where we had one specialized smoother at each level both moving up and down the hierarchy. If we compare the performance of the trained smoothers in Figure 2.23(a) and Figure 2.19, it can be seen that the final accuracy is obtained with a few extra iterations. However, multigrid with the generalized smoothers do still outperform both Jacobi and Gauss Seidel methods.

The second observation is that the objective is reached and that the single $K_{M,\uparrow}$ and $K_{M,\downarrow}$ enable the generalization. The convergence histories reported in the various graphs in Figure 2.23 show that increasing the mesh size and the number of levels do not change at all the convergence rate of multigrid with the learned smoothers. This is indeed a unique numerical property of the multigrid methods. Regardless of the specific mesh size $h$ or the dimension of the problem, the number of iterations to converge will be the same. This enables us to design models that generalize well and do not need to re-learn the same properties at every resolution.

To finish, we can have a study about the learned kernels. One of the idea behind the generalization, was to say that the kernels $K_{M,\downarrow}$ and $K_{M,\uparrow}$ were a mean of the learned kernel use for the fully adaptive smoother. However, if do compute the mean using the Figure 2.22, we obtain the following kernels:

$$K_{M,\text{mean},\downarrow} = \begin{bmatrix} -0.026 & 0.147 & -0.026 \\ 0.147 & 0.160 & 0.146 \\ -0.026 & 0.146 & -0.026 \end{bmatrix} \qquad K_{M,\text{mean},\uparrow} = \begin{bmatrix} 0.201 & 0.270 & 0.201 \\ 0.270 & 0.420 & 0.270 \\ 0.201 & 0.270 & 0.201 \end{bmatrix} \tag{2.6}$$

which are not the same as in equation 2.5. This difference suggests that the generalized kernels $K_{M,\downarrow}$ and $K_{M,\uparrow}$, as defined in Equation 2.5, are not simple averages of the fully adaptive stencils learned at each level, but rather represent optimized operators trained in a different context.

Several factors may explain this difference:

- The fully adaptive smoother is trained to optimize convergence locally at each level, whereas the generalized kernel aims to perform uniformly across levels.
- Averaging can smooth out critical variations, especially when the learned stencils adapt to grid resolution or error components differently.

### 2.3.3 Learning at once the smoother and the grid transfer operator

In multigrid the smoother application is followed by the restriction operator when going down the grid hierarchy and preceded by the prolongation when going up. In this section, we investigate the possibility to learn the two operators at once, instead of simply learning the smoothers as we did so far.

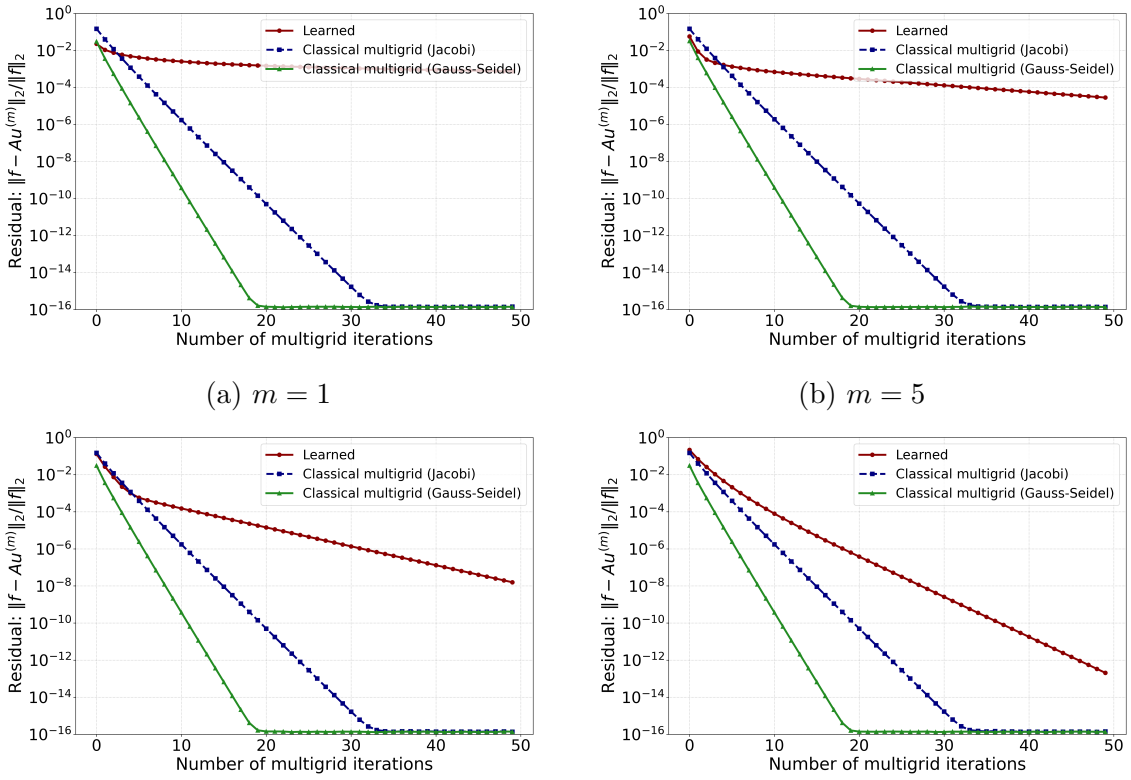
The first decision to make concerns the size of the convolution kernels that will replace the product of two operators. For example, combining a $1 \times 1$ Jacobi kernel with a $3 \times 3$ transfer operator, either preceding or following it, results in an effective $3 \times 3$ convolution kernel. That is what we refer to as case A: learning a $3 \times 3$ kernel, that mimics the level-wise weighted Jacobi, with a classical grid transfer.

Therefore, when learning a $3 \times 3$ kernel, one would expect the resulting multigrid method to exhibit numerical behavior similar to that of a learned Jacobi-based scheme. Similarly, if one aims to replace the full adaptive smoother combined with a grid transfer operator; that is, the product of two $3 \times 3$ convolution kernels, then it is reasonable to consider learning a single $5 \times 5$ convolution kernel instead. That is our case B: learning a $5 \times 5$ kernel, that mimics the full adaptive smoother, combined with a grid transfer.

In the sequel we denote these kernels respectively as $K^{\alpha}_{M+R,\downarrow}$ and $K^{\alpha}_{M+P,\downarrow}$, with $\alpha$ the with of the kernel. We denote them as **Transfer-Smoothing Kernels$^{\boldsymbol{\alpha}}$, $\boldsymbol{TSK^{\alpha}}$**.

## Case A: Learning a $3 \times 3$ kernel

This kernel is trained within the same learning framework as before, i.e, loss function based on the residual norm, $N = 64$, $L = 4$, $\nu_1 = 1$ and $\nu_2 = 2$, with the goal of preserving convergence efficiency while simplifying the algorithm. We experiment with $m = 1$, 5, 10 and 20.




(a) $m = 1$ (b) $m = 5$

(c) $m = 10$ (d) $m = 20$

Figure 2.24: Learned TSK$^3$, with the residual based loss, trained on $N = 64$, $L = 4$ and several values of $m$.

As shown in the Figure 2.24, increasing $m$ improves the effectiveness of the learned smoother, however, the learned kernel always converges slower than classical Jacobi. In appears that increasing the number of multigrid cycles $m$, enables the learned kernel to closely approximate the behavior of classical Jacobi methods, which aligns with the intended goal sense. However, he does not simplify the algorithm, as we have to compute $m > 20$ to have a fitting close to the classical Jacobi method.

This indicates that the $3 \times 3$ stencil lacks the expressive power required to handle both roles effectively. For reason that we do not fully understand trying to replace a $3 \times 3$ constant kernel followed by a learned $1 \times 1$ by a a single $3 \times 3$ kernel does not work or at least would require more investigations.

## Case B: Learning a $5 \times 5$ kernel

In light of this limitation, we extended the stencil size to $5 \times 5$, thereby increasing its capacity to capture broader spatial interactions and potentially encode both smoothing and restriction behaviors. The resulting kernel contains 25 trainable parameters, allowing for more complex and effective filtering.

We first trained this single $5 \times 5$ stencil with $m = 1$ multigrid cycle, to observe its early impact on convergence.

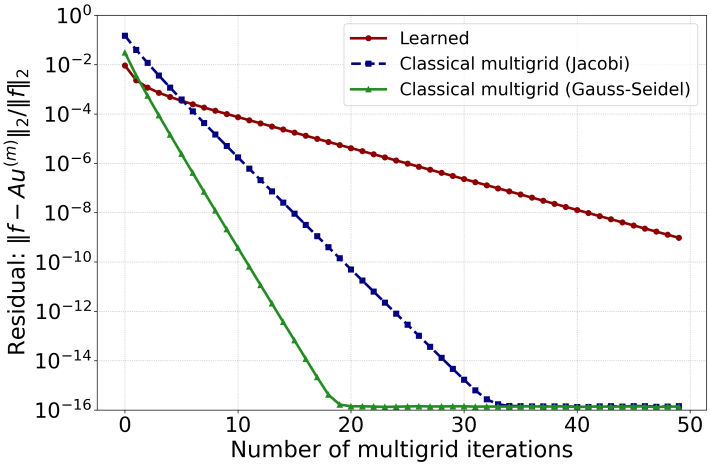



Figure 2.25: Learned TSK$^5$, with $m = 1$ and the residual based loss.

As illustrated in Figure 2.25, the performance improves in the very early iterations, with a moderate reduction in residual per iteration. Nonetheless, the convergence remained suboptimal over multiple iterations.

To enhance convergence, we increased the number of training cycles $m$, which allows the stencil to optimize over a longer sequence of updates and better generalize its dual function.

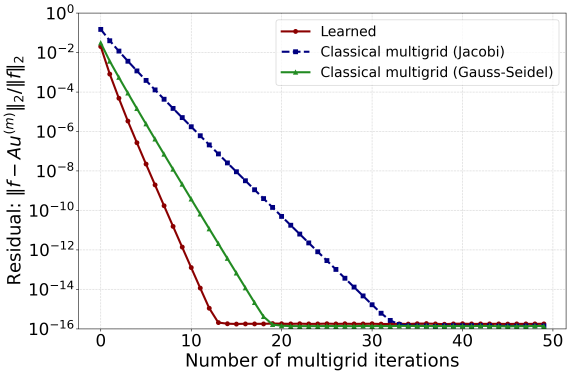



Figure 2.26: Learned TSK$^5$, with $m = 5$ and the residual based loss

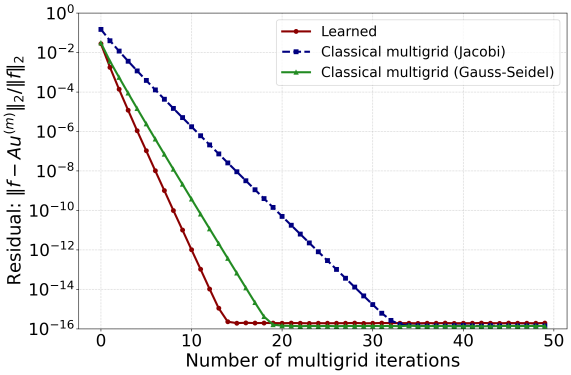



Figure 2.27: Learned TSK$^5$, with $m = 10$ and the residual based loss

As shown in Figure 2.26 and 2.27, this leads to a significant improvement, bringing performance closer to that reported in Figure 2.23 where we only learned the smoothers.

**Conclusion.** These experiments suggest that jointly learning smoothing and grid transfer operations is possible, but requires a larger stencil and sufficient training depth. This opens up new research directions, such as exploring adaptive stencil sizes, regularization techniques to control parameter growth, or applying similar strategies to the prolongation operator. Ultimately, this line of work could lead to a fully unified, learnable multigrid solver with minimal hand-designed components.

# Conclusion

My time at CERFACS has been an exceptionally both collegial and intellectually stimulating atmosphere. From my first day, I found myself immersed in a culture of curiosity and mutual support: whether browsing the extensive library of books and articles, consulting with experts, or attending the rich thesis defenses and conferences such as the "Sparse Days", there was always an opportunity to expand my knowledge. Regular seminars and conferences brought together leading researchers, fostering vibrant discussions and invaluable networking.

The training courses offered by CERFACS (GIT introduction, advanced scientific programming, effective English presentations) proved both free of charge and immediately applicable, enhancing my practical skills and accelerating my professional growth. In short, I have thoroughly enjoyed every aspect of life at CERFACS, its collaborative spirit, abundant resources, and commitment to continuous learning made this internship an unforgettable experience.

Throughout this internship, I developed and evaluated a hybrid multigrid Poisson solver enhanced by learned convolutional smoothers. First, I implemented classical iterative solvers (Jacobi, Gauss–Seidel) and confirmed their complementary strengths and limitations in damping high- and low-frequency errors. I then reformulated each multigrid component—residual computation, restriction, prolongation, and smoothing—as local convolutional kernel.

Then, I focused on learning optimal relaxation operators at each level of the multigrid hierarchy. In the level-wise setting, I trained scalar-weighted Jacobi kernels to adaptively dampen high- and low-frequency errors, demonstrating significant acceleration compared to classical fixed-$\omega$ schemes. I then extended this approach to a full-stencil representation ($3\times3$ convolutional filters), enabling the solver to learn anisotropic and spatially varying corrections. This per-level flexibility yielded dramatic improvements in convergence rate and robustness, as the learned stencils captured local coupling patterns that scalar weights alone could not.

Building on these level-specific successes, I explored a further generalization: training a single pair of descent and ascent kernels shared across all levels. Beyond merely smoothing, I co-learned the restriction and prolongation operators themselves, parameterizing them as convolutional filters and optimizing their weights together with the smoothing kernel. This unified approach achieved mesh-size invariant convergence, maintaining uniform error reduction even as the grid was refined. By learning both transfer operators and smoothing in one shot, the solver's architecture becomes remarkably compact, highly scalable, and directly transferable to new meshes and boundary conditions without retraining per level.

These trainings, level-wise weighted Jacobi kernels and fully adaptive $3 \times 3$ convolutional filters used both error-based and residual-based losses. Comparing loss-on-error versus loss-on-residual revealed that while both objectives yield faster convergence than analytical weights, optimizing directly on the residual norm often produces more robust behavior across iterations. In the"full $3 \times 3$" case, the learned kernels outperformed classical smoothers by reaching machine-precision residuals in under ten V-cycles, demonstrating the clear advantage of data-driven kernels.

To conclude, future work can explore larger convolutional kernels (e.g. $5 \times 5$), joint learning of smoothers and transfer operators, and extension to more complex PDEs and boundary conditions. Integrating uncertainty quantification into the training loop and deploying the solver on HPC clusters with mixed-precision and GPU-optimized convolutions will further strengthen performance and applicability.