Comparison of multigrid and machine learning-based Poisson solvers

Hadrien Godé[1], Carola Kruse[1], Richard Angersbach[2], Harald Köstler[2], Michaël Bauerheim[3], Ulrich Rüde[1,2]

1 CERFACS

2 FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

3 isae Institut Supérieur de l'Aéronautique et de l'Espace SUPAERO

Motivation

Poisson's equation: $\nabla^2\phi = f$

Electrostatic: elictric field potential linked to charge density function (ρ_f) and permittivity (ϵ)

$$\nabla^2\phi = -\frac{\rho_f}{\epsilon}$$

Fluid dynamics: pressure field linked to density (ρ) and velocity (U)

$$\nabla^2 P = -\rho\nabla.(U.\nabla U)$$

Plasma, material science, biophysics…

Motivation

What is the best solver for Poisson's equation ?

- Multigrid, conjugate gradient, Fourier, neural network… Response vary, but:
  - Historically: Multigrid method
  - Trend: Machine learning based methods (UNet)

- **Problem: unawareness/ legacy code**

Plan

1. Multigrid presentation
2. Unet presentation
3. Similarity & discrepancy
4. Direct comparison
5. Conclusion advantages and drawbacks

Multigrid presentation

Multigrid presentation: stencils representation

- With the finite difference method : $\nabla^2\phi = f$ in 2D becomes:

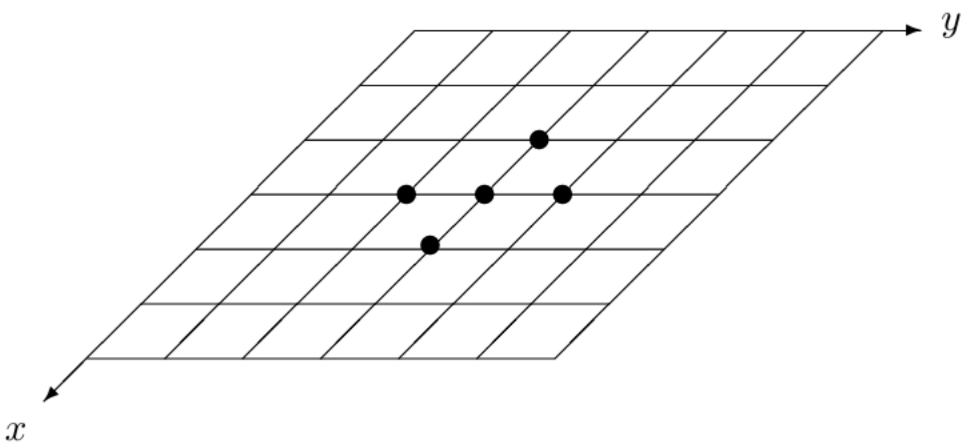
$$\frac{-\phi_{i-1,j} + 2\phi_{i,j} - \phi_{i+1,j}}{h_x^2} + \frac{-\phi_{i,j-1} + 2\phi_{i,j} - \phi_{i,j+1}}{h_y^2} = f_{i,j}$$

$$\frac{1}{h^2}\left(\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
& -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
& -1 & & -1 & 4 & -1 & & -1 & \\
& & -1 & & -1 & 4 & & & -1 \\
\hline
& & & -1 & & & 4 & -1 & \\
& & & & -1 & & -1 & 4 & -1 \\
& & & & & -1 & & -1 & 4
\end{array}\right)\phi_{i,j} = f_{i,j}$$

Multigrid presentation : stencils representation

$$\frac{1}{h^2}\begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix}\phi_{i,j} = f_{i,j}$$

- 5 point stencils $\boxed{\rightarrow}$ Solving Ax=B



Multigrid presentation : Main idea

$$Ax = f$$

$$Ax - f = r$$

Let u be the ground truth and v our guess:

$$e = u - v$$

$$Ae = r$$

Let assume there exists B an approximation of A, where $B^{-1}r$ is easy to compute and is used to obtain the next guess, then:

$$\tilde{e} = B^{-1}r$$

$$e_{k+1} = u - (v_k + \widetilde{e_k}) = e_k - \widetilde{e_k}$$

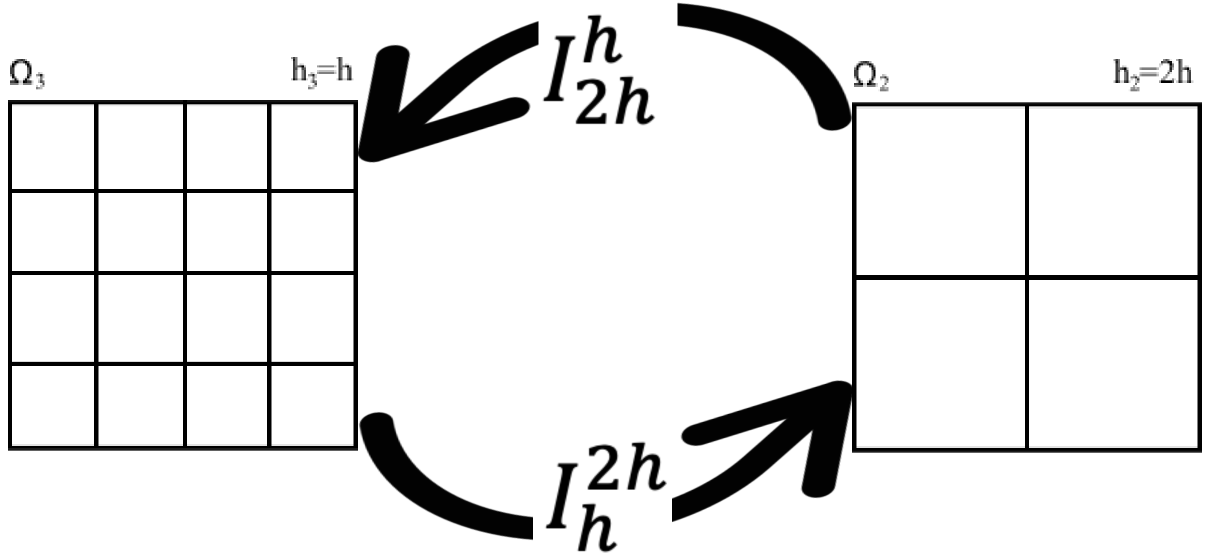
Multigrid presentation : The algorithm

First guess « v »: $e = 0$, example here: Jacobi (repeat ν times):

$$v_{i,j} \leftarrow \frac{1}{4}\left(v_{i,j-1} + v_{i,j+1} + v_{i-1,j} + v_{i+1,j} + h^2 r_{i,j}\right)$$

-> Slow convergence, notably for the smooth component of error

-> Solution compute an approximation on a coarser grid



Multigrid presentation : The algorithm

- Restriction to coarser grid (full weighting):

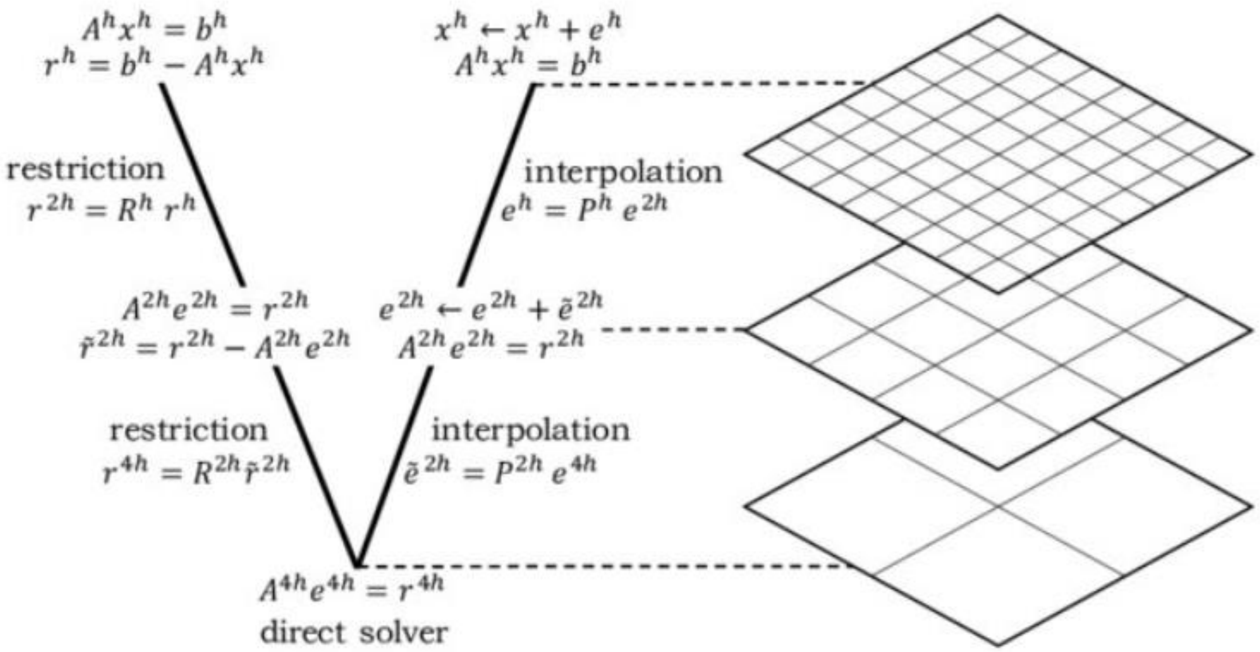
$$\underbrace{I_h^{2h} A_h I_{2h}^h}_{A_{2h}} e_{2h} = \underbrace{I_h^{2h}(f_h - A_h v_h)}_{r_{2h}}$$

- Correct the approximation on finer grid (bilinear interpolation):

$$v_h = v_h + I_{2h}^h v_{2h}$$

Multigrid presentation : The algorithm

1. Relaxation (Ae=r with initial guess e=0)
2. Restrict (interpolate) the residual to a coarser grid
3. If coarsest grid : (cheap computation) solve Ax=B
4. Else: recursively restart this algo at coarser grid
5. Prolongate/ correct to fine grid approximation
6. Relaxation (Ae=r with initial guess from coarsest grid)



Multigrid presentation : operation count

- Operation count per pixel:
  - Jacobi smoother: $v_j \leftarrow \frac{1}{4}(v_{i,j-1} + v_{i,j+1} + v_{i-1,j} + v_{i+1,j} + h^2 f_i)$ ➡ 5 additions and 1 multiplication
  - Compute and transfer the residual to coarser grid $\underbrace{I_h^{2h} A_h I_{2h}^h}_{A_{2h}} e_{2h} = \underbrace{I_h^{2h}(f_h - A_h v_h)}_{r_{2h}}$ ➡ 25/4 additions and 5/4 multiplication
  - Interpolate the correction to finer and addition to the previous approximation $v_h = v_h + I_{2h}^h v_{2h}$ ➡ 7/4 additions and 3/4 multiplication
  - Coarse grid solver ➡ depends, we have counted approximately 42 operations per pixel on our tests

Multigrid presentation : operation count

- At each level (except coarsest) operation count = 16 operations per grid point
- At coarsest: approx. 42 operations per grid point
- For a V cycle of depth 4 with 101x101 grid points:
  - Finest level: $10201 \times 16 = 163216$
  - Level 2: $2500 \times 16 = 40000$
  - Level 2: $625 \times 16 = 10000$
  - Coarsest grid: $144 \times 42 = 6048$
- Total number of operations: 219 624
- Total number of operations per grid points: 22

Multigrid presentation : operation count

- The V-Cycle is repeated until residual < threshold
- In our test for threshold = 1e-3 ➡ 4 iterations

Total number of operations per grid points: 88

UNet presentation

UNet presentation: Main idea

- Laplacian operator can be nondimensionalized:

$$\nabla^2_{\Delta} = \frac{1}{\Delta^2}\left(\frac{\partial^2}{\partial \bar{x}^2} + \frac{\partial^2}{\partial \bar{y}^2}\right)$$

- If training done at resolution Δ_{NN} and prediction is wanted a resolution Δ_{sim}:

$$\nabla^2_{sim} = \frac{\Delta^2_{sim}}{\Delta^2_{NN}} \nabla^2_{NN}$$

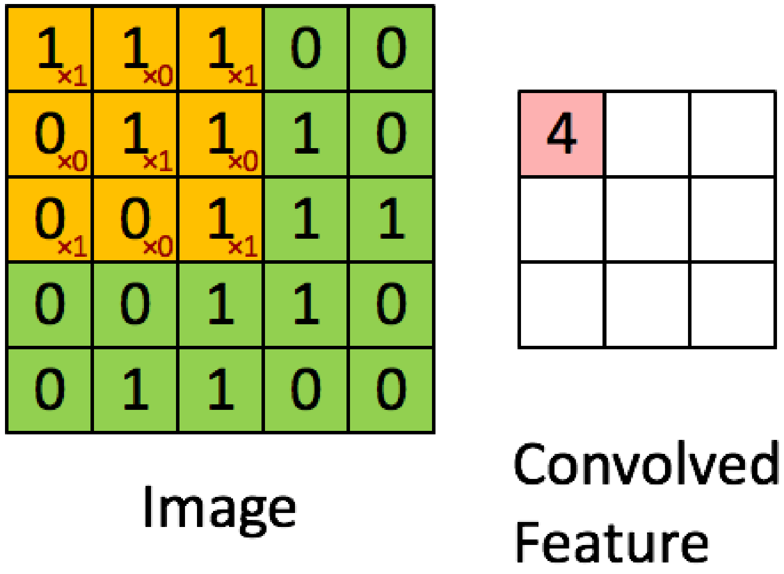
UNet presentation: Main idea

- Multiple tunable neurons that can learn complex functions
- non-convex optimization procedure is performed to update the neuron weights by minimizing a cost function (the loss function)
- Neural networks are denoted by f such that:

$$\phi_{out} = f(R_{in})$$

UNet presentation: the algorithm

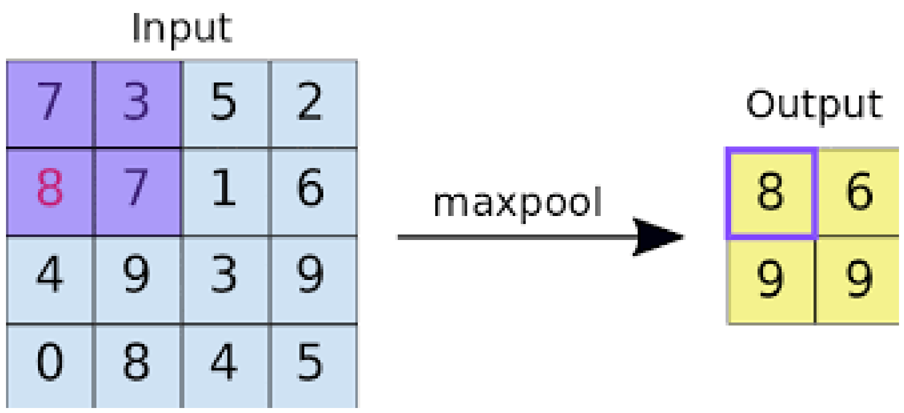
- Input: Right hand side (size $1, n_x, n_y$)
- Convolution to matrix of size $(n_{feature}, n_x - 2, n_y - 2)$



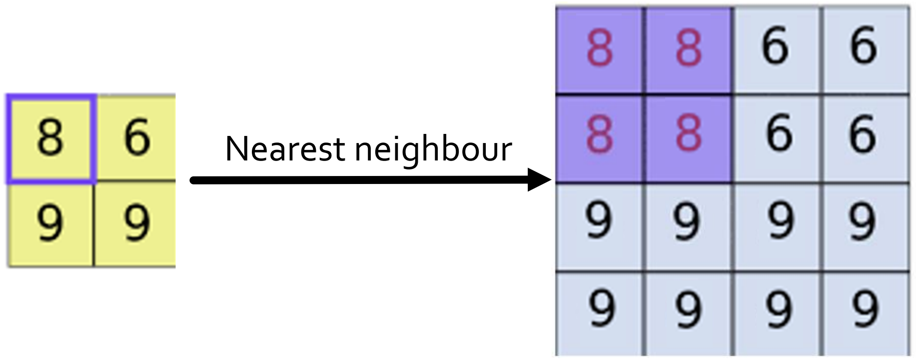
- ReLu (replace value by 0 if negative or keep value)

UNet presentation: the algorithm

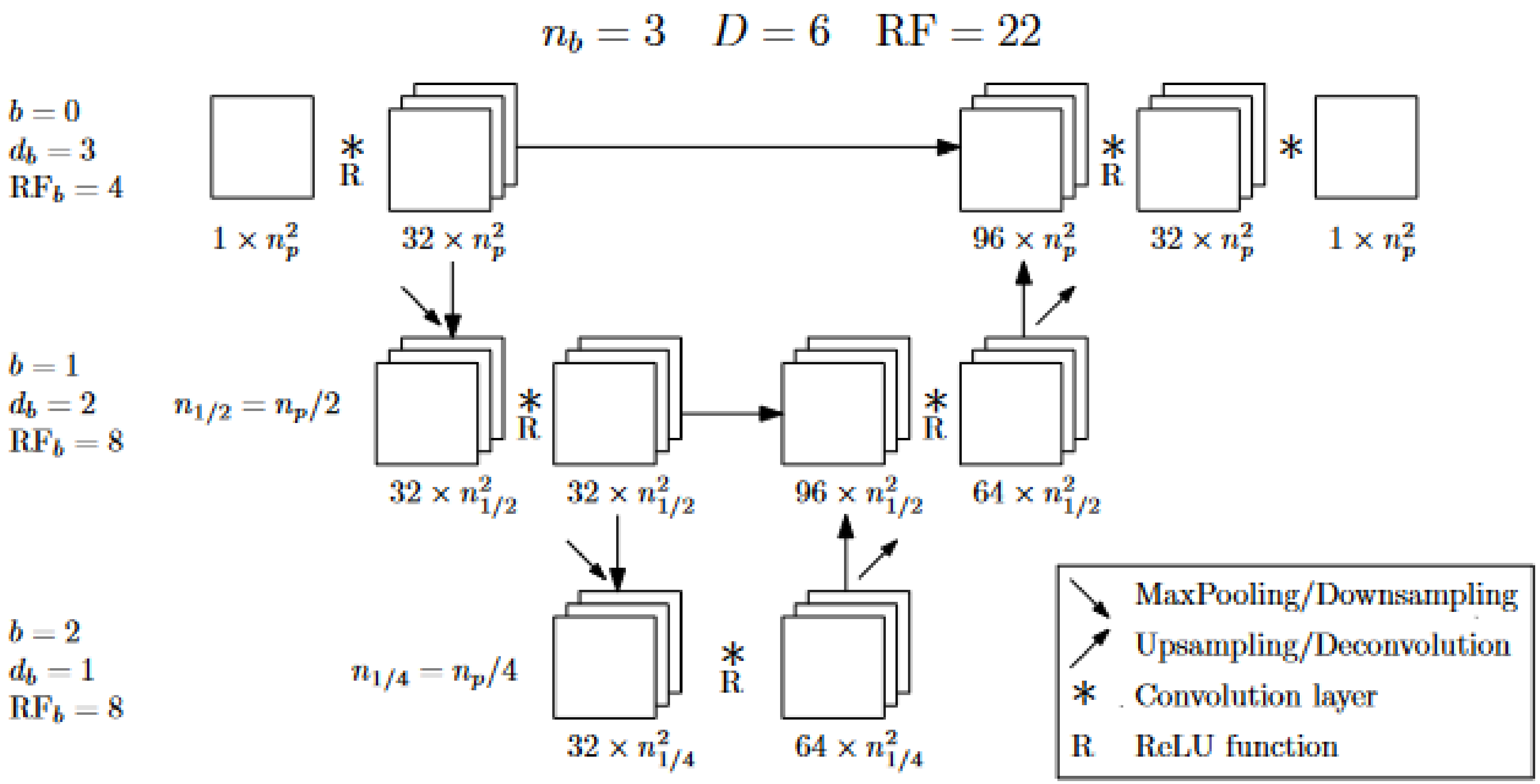
- Maxpooling:



- Upsampling: nearest neighbour



UNet presentation: the algorithm



UNet presentation: operation count

- Convolution: $Kernel_{size}^{2} \times input_{feature} \times output_{features} \times n_x \times n_y$ Operations
- ReLu: $n_x \times n_y$ operations
- Maxpool: $3 \times n_{x_{output}} \times n_{y_{output}}$
- Upsample: $output_{features} \times n_{x_{output}} \times n_{y_{output}}$

UNet presentation: operation count

Level	Level FLOP count
Level (	170 795 343 FLOP
Level 1	27 030 500 FLOP
Level 2	11 971 250 FLOP
Level 3	2 613 716 FLOP
Level 4	782 712 FLOP

Total:

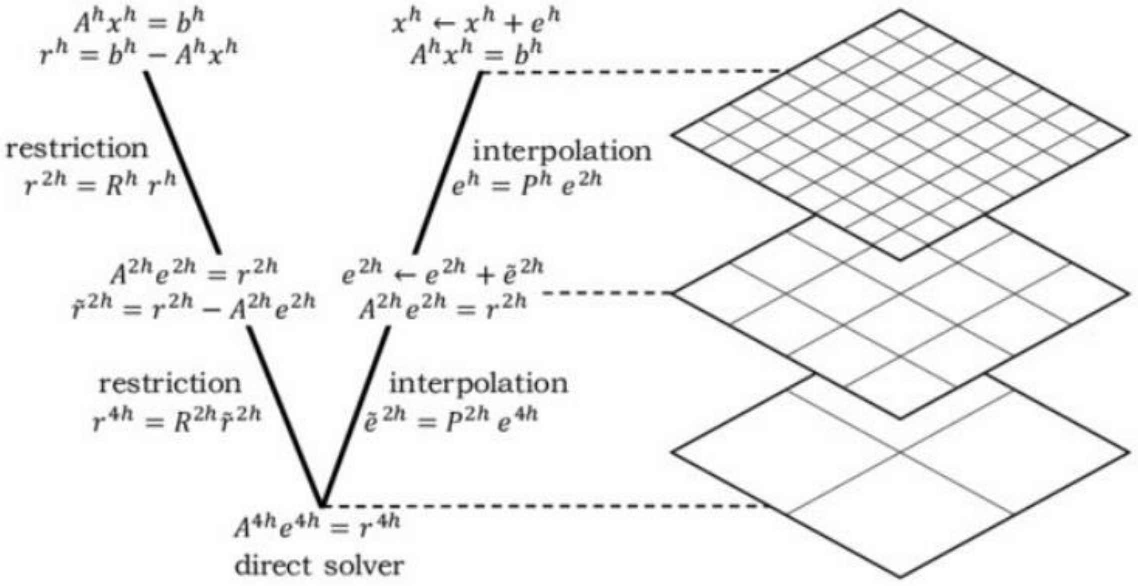
201 222 271 FLOP,

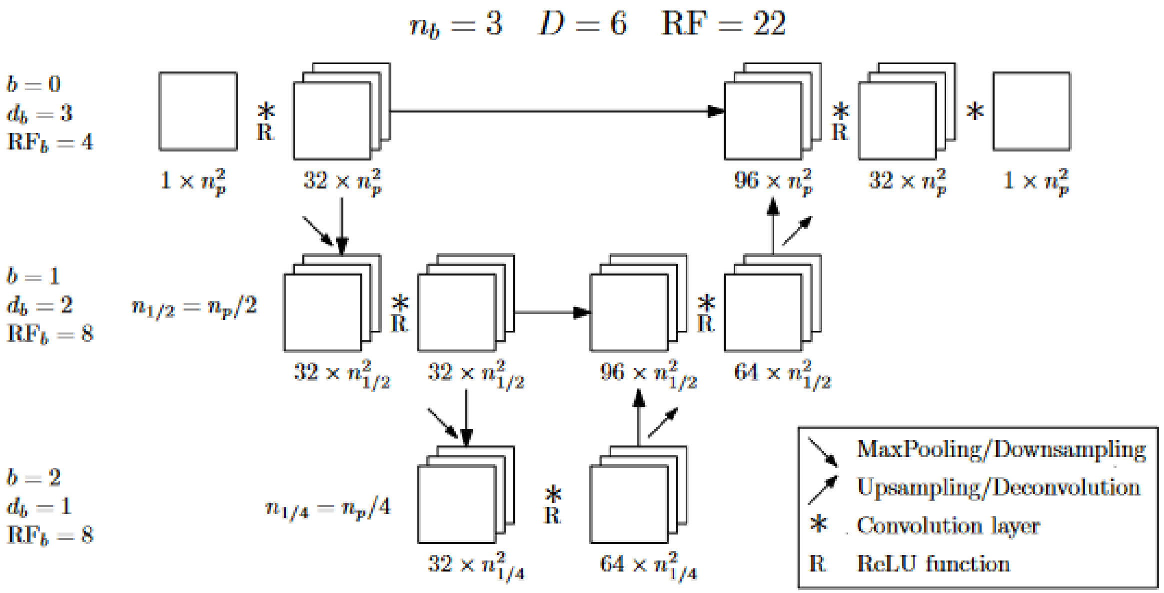
Or:

19 726 FLOP per pixel.

Similarity & discrepancy

Similarity & discrepancy: architecture comparison





Similarity & discrepancy: equivalence

- Restriction / Maxpooling
- Prolongation / Upsampling
- Convolution / Smoother

Similarity & discrepancy: difference

- Work on residual / work directly on data
- 2D matrix / Creation of a third axis for features
- Direct solver on coarsest / No direct solver on coarsest
- Iterative / Not iterative

Similarity & discrepancy: application

- Don’t use skip connection & ReLu
- Replace Maxpooling by average pooling
- Replace Upsampling by linear interpolation
- Use stencil as convolution kernel & slightly rewrite convolution step -> becomes a Jacobi smoother
- Use direct solver at deepest
- Work on residual rather than Right hand side

= We can define a MG V-cycle with pytorch

```python
def restrict_pytorch(u):
    u = F.avg_pool2d(u, 2)
    u[:, :, :, 0] = 0
    u[:, :, :, -1] = 0
    u[:, :, 0, :] = 0
    u[:, :, -1, :] = 0
    return u
```

```python
def prolongate_pytorch(u):
    u = F.interpolate(u, scale_factor=2, mode='bilinear', align_corners=True)
    u[:, :, :, 0] = 0
    u[:, :, :, -1] = 0
    u[:, :, 0, :] = 0
    u[:, :, -1, :] = 0
    return u
```

```python
def weighted_jacobi_smoother_pytorch(u, f, stencil, omega, iterations):
    h = 1 / np.shape(u)[-1]
    stencil = (1 / h**2) * stencil
    central_coeff = stencil[0, 0, 1, 1]
    for _ in range(iterations):
        u_conv = F.conv2d(u, stencil, padding=0)
        u_conv = F.pad(u_conv, (1, 1, 1, 1), "constant", 0)
        u = u + omega * (f - u_conv) / central_coeff
        u[:, :, :, 0] = 0
        u[:, :, :, -1] = 0
        u[:, :, 0, :] = 0
        u[:, :, -1, :] = 0
    return u
```

Direct comparison

Direct comparison: the test

- Test:

$$\nabla^2 u = -2\pi^2 \sin(\pi x)\sin(\pi y) \text{ on } \Omega$$

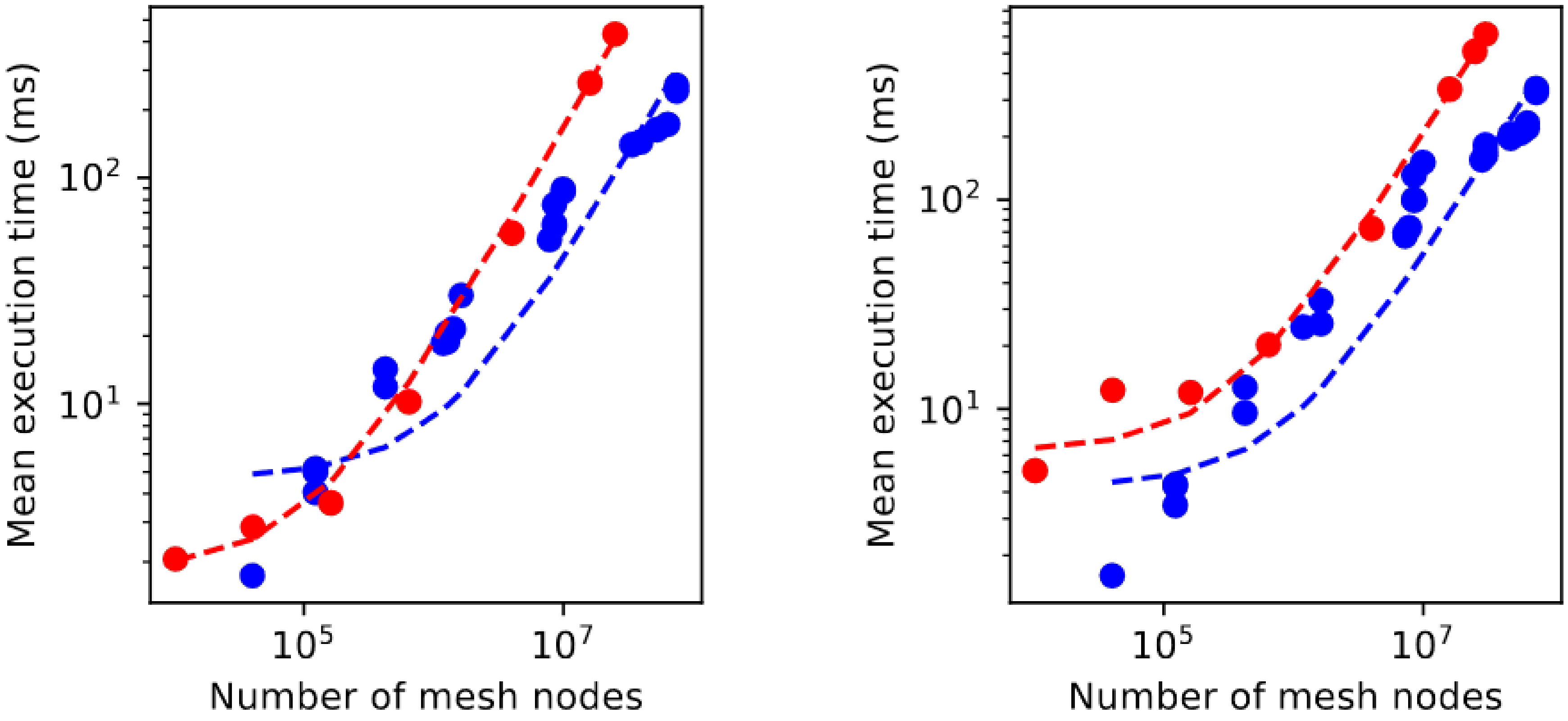
$$u = 0 \text{ on } \delta\Omega$$

$$\text{(exact solution: } u = -\sin(\pi x)\sin(\pi y)\text{)}$$

Residual threshold: 10^{-3}(UNet limit)

- Computer resource:
  - config_1: Bi-socket Intel node with 2 x 18-core Xeon Gold 6140 CPUs (2.3 GHz, 96 GB memory), interacting with 4 NVIDIA V100 32 GB GPUs (only one used in this study).
  - – config_2: Bi-socket AMD node with 2 x 64-core EPYC Rome 7702 CPUs (2 GHz, 512 GB memory), interacting with a single NVIDIA A100 40 GB GPU.

Direct comparison: mean execution time results



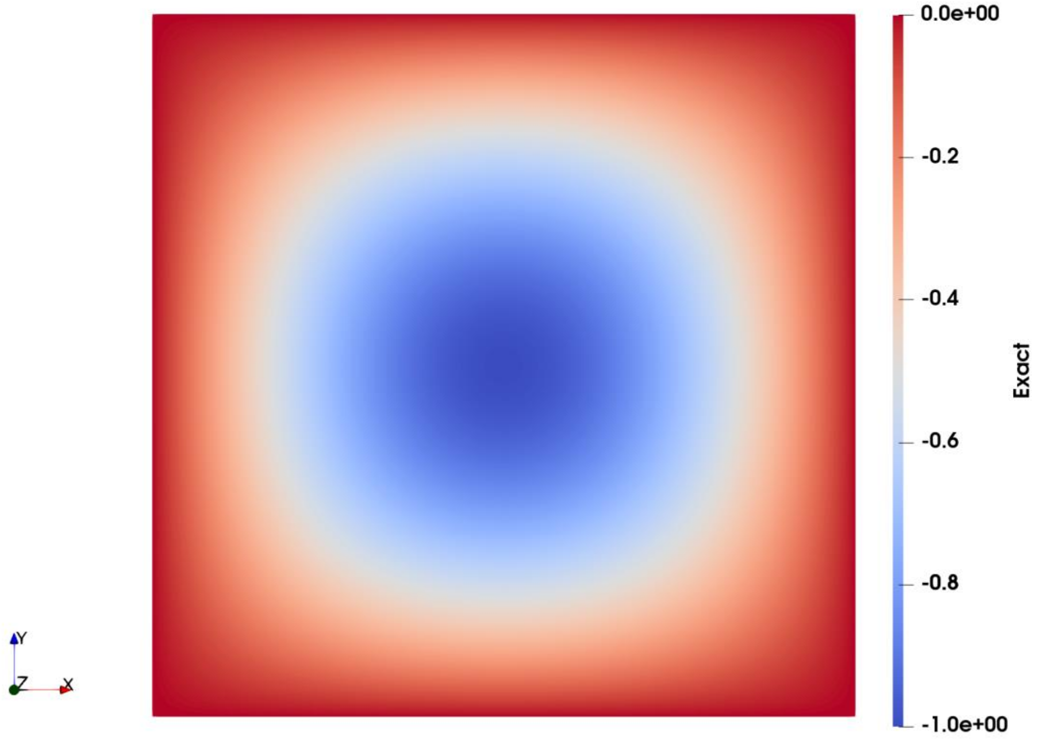
(a) On V100 GPU

(b) On A100 GPU

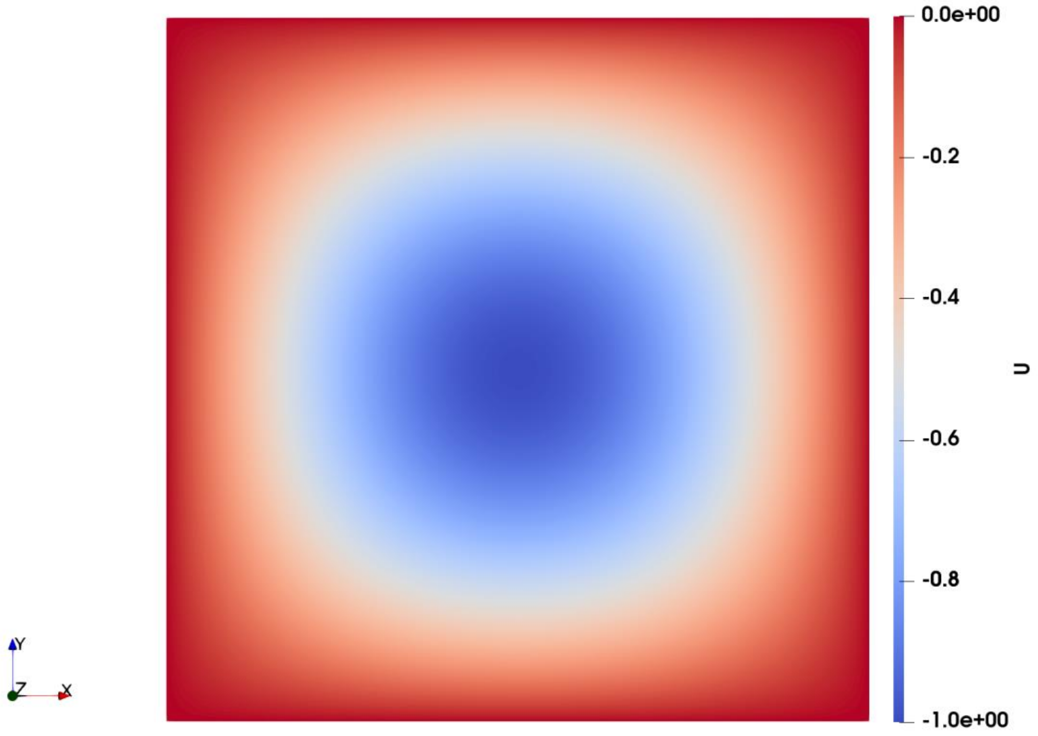
Multigrid V-Cycle results UNet results

Direct comparison: accuracy results

Exact solution

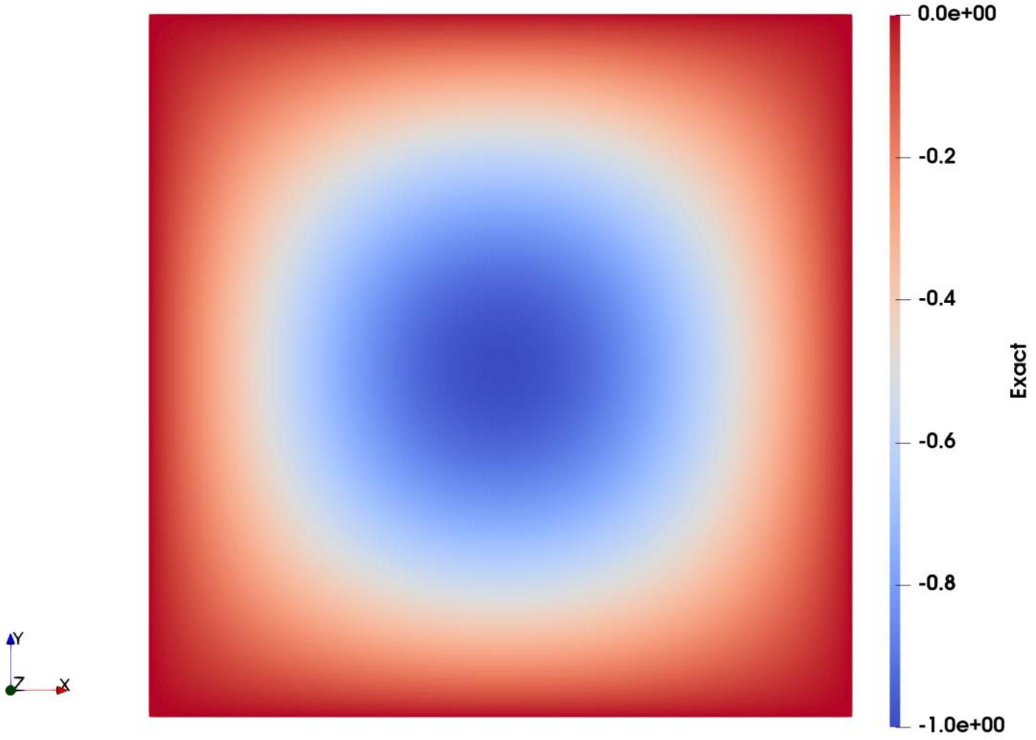


Multigrid result

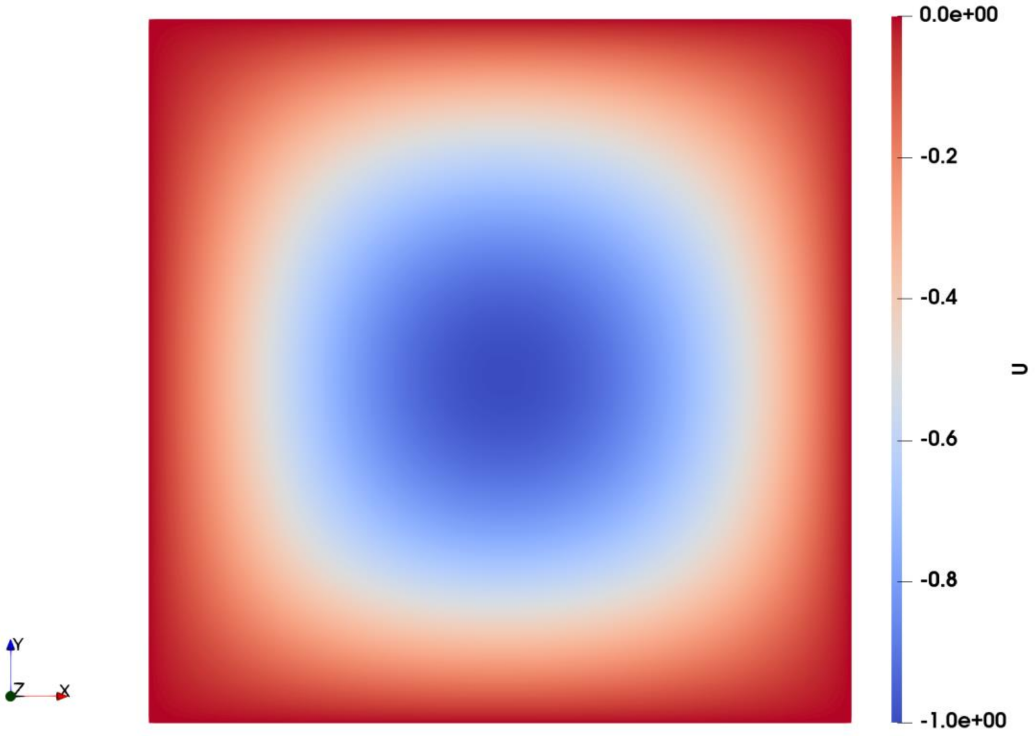


Direct comparison: accuracy results

Exact solution

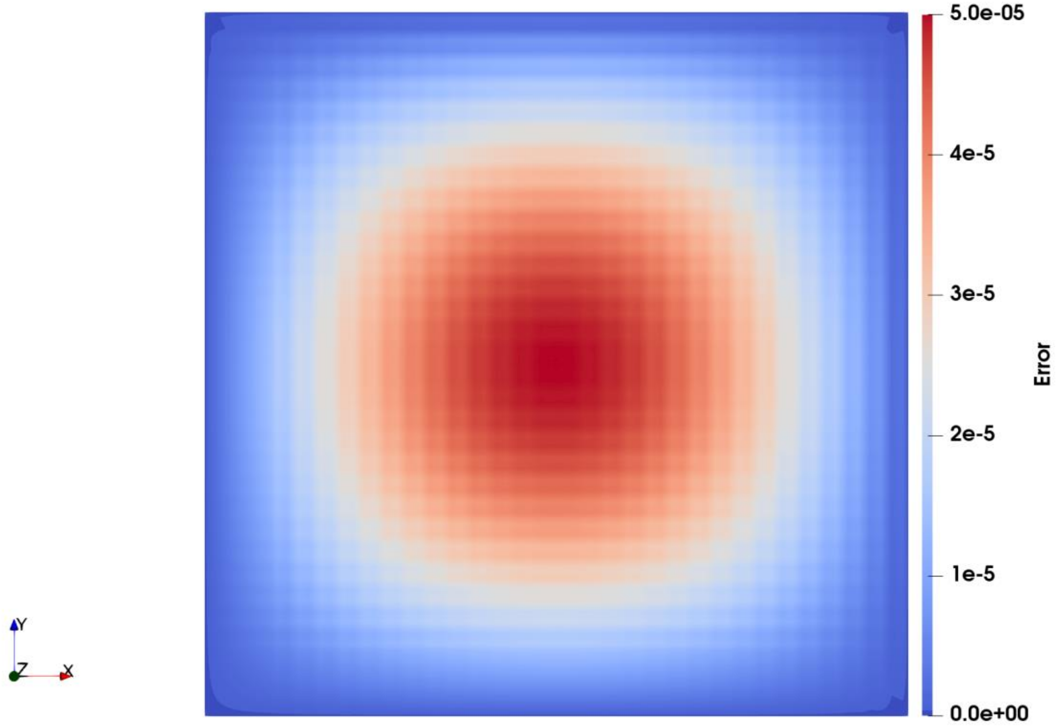


UNet result

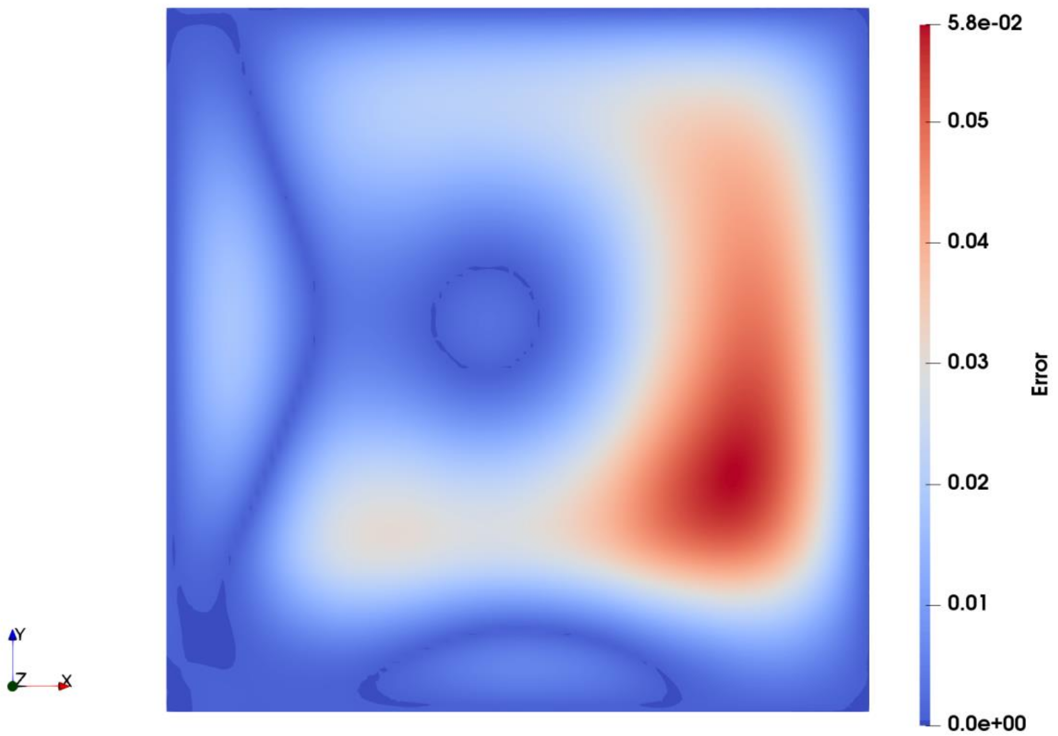


Direct comparison: accuracy results

Multigrid error abs(U-Exact)



UNet error abs(U-Exact)



Direct comparison: discussion

- For Poisson’s equation:
  - Multigrid faster
  - Less operation per grid points
  - Less IO operations
  - Controlable precision
  - Multigrid more accurate

- Using multigrid to solve Poisson’s equation is best !

Conclusion: advantages and drawbacks

Conclusion: advantages and drawbacks

Multigrid	Unet
+ Parallelizable	+Highly parallelizable
+ Linear scaling with the number of mesh nodes	+Linear scaling with the number of mesh nodes
+ Controllable precision (scale with –log10(residual))	+Well studied & developed
+Fast(est) on continuous isotropic problem	+ Applicable to a discontinuous problem
O Need complementary features for finite element method	+ Applicable to a anisotropic problem
O variety of architecture	O variety of architecture
- Less efficient on discontinuous problems	- Need for datasets & training
- Less efficient on anisotropic problems	- Limited precision