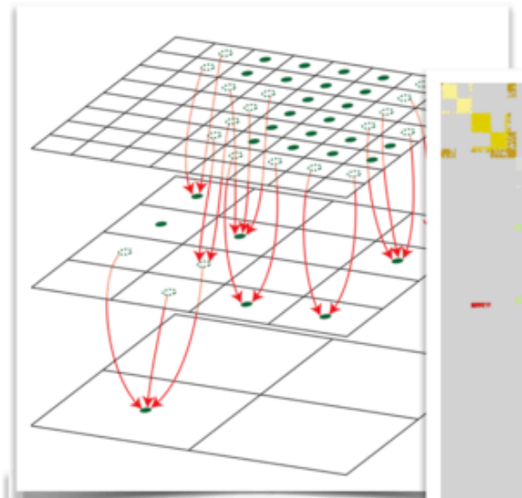

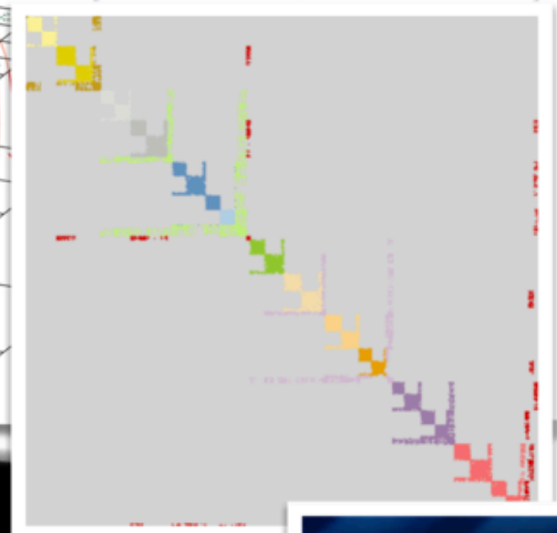

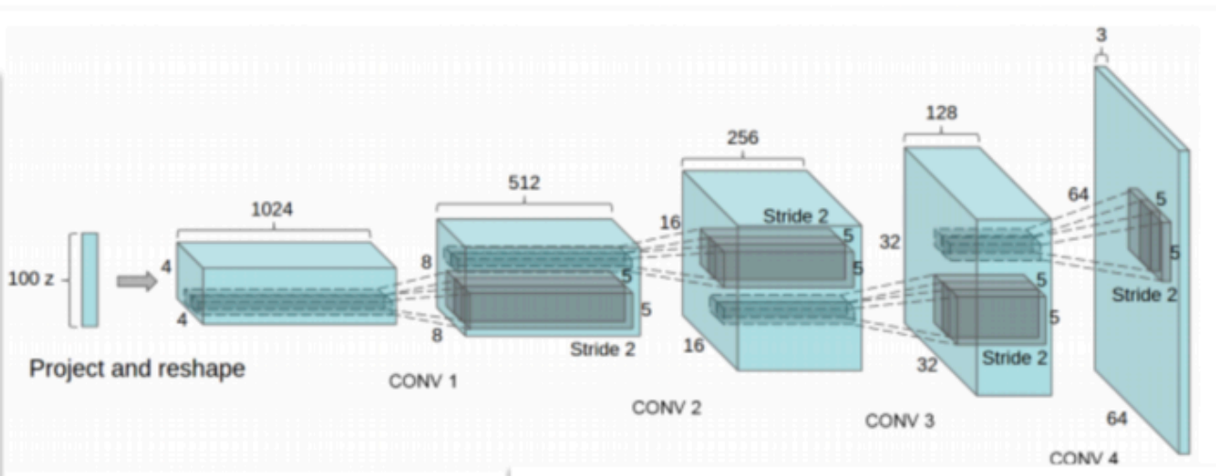

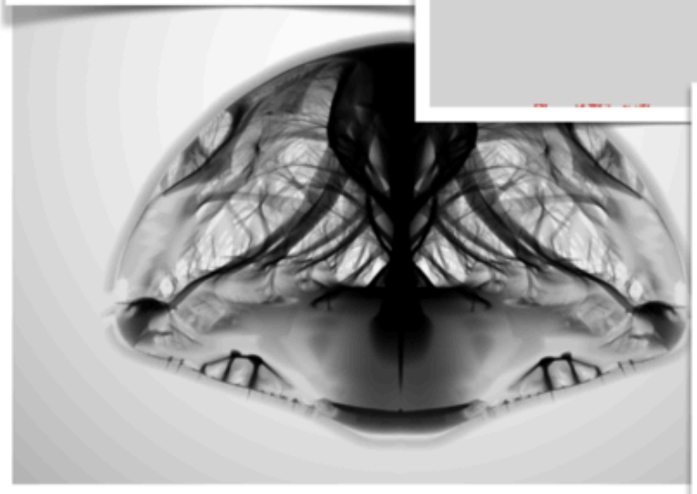

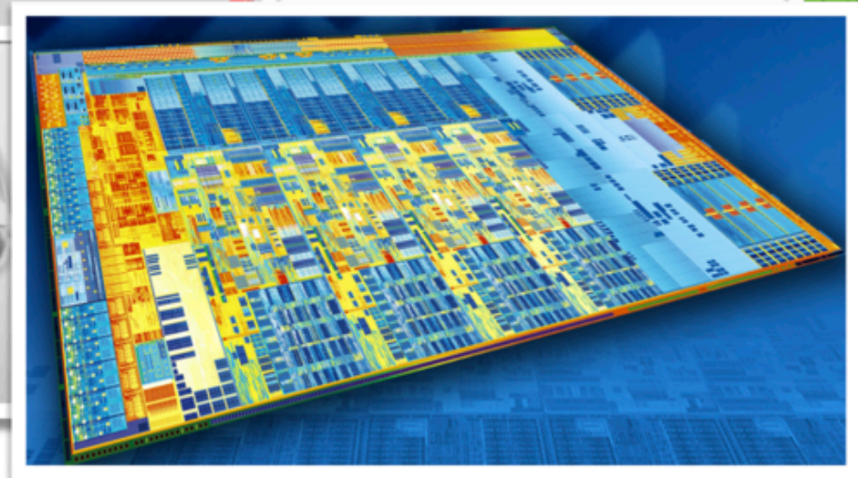

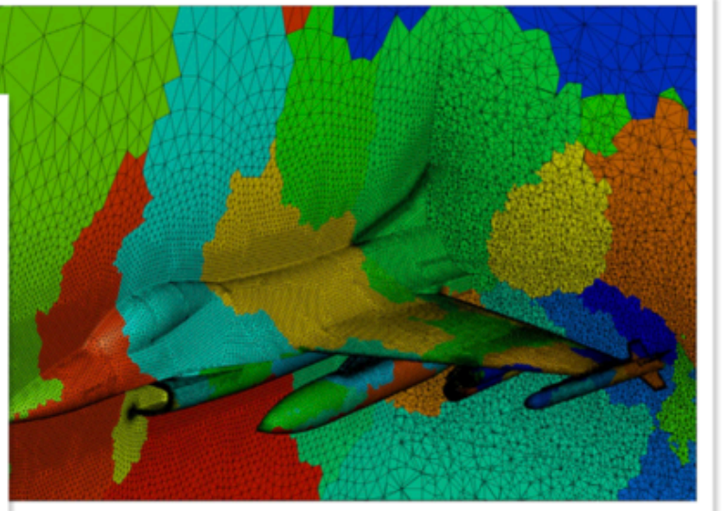

Lecture 7: The Conjugate Gradients algorithm applied to the Laplace system. Implementation walkthrough (cont'd)

Tuesday February 14th 2023

Logistics

- Programming Assignment #1 due ***Friday (in 3 days)!***
    - Deadline is at midnight (see late policy in the detailed homework description, on Canvas)
- Reminder: Instructor's office hours at at CS6387 (in-person) Wednesdays 1:15-2:15pm.

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Performance

- Typically requires at least as many iterations as the domain diameter
- Or fixed number with a good "preconditioner" (M)

Prerequisites

- Requires a symmetric system matrix
- Matrix needs to be positive definite

Benefits

- Low storage overhead
- Simple component kernels

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Vectors or arrays?

- Variables ***x***, ***f***, ***r***, ***p***, ***z*** are shown in this pseudocode as mathematical "vectors"
- However, their real representation is 3D arrays (grid-based arrays)
- e.g. u[..][..][..] is the actual implementation of ***x***!
- Instead of "converting" them to traditional vectors, we emulate operations in their native representation.

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Non-boldfaced symbols are numbers (scalars)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as
ComputeLaplacian(p, z)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

For vectors **x**, **y**, **z**, (and a number c)
SAXPY(**x**, **y**, **z**, c) is defined as:

```
for i=1 ... N
    z_i = c * x_i + y_i
```

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Pointwise Ops (PointwiseOps.h)

LaplaceSolver/LaplaceSolver_0_0


```
#pragma once

#include "Parameters.h"

// Copy array x into y
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM]);

// Scale array x by given number, add y, and write result into z
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale);
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0


```
#include "PointwiseOps.h"

void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        y[i][j][k] = x[i][j][k];
}
```

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const float scale)
{
    // Should we use OpenMP parallel for here?
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
}
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0


```
#include "PointwiseOps.h"

void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        y[i][j][k] = x[i][j][k];
}
```

For vectors **x**, **y**, **z**, (and a number c)
SAXPY(**x**, **y**, **z**, c) is defined as:

for i=1 ... N

$z_i = c * x_i + y_i$

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const float scale)
{
    // Should we use OpenMP parallel for here?
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
}
```

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Implemented as
Saxpy(p, x, x, alpha)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as
Saxpy(p, z, p, beta)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Recap

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as
Saxpy(z, r, r, -alpha)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as
Saxpy(z, r, r, -alpha)

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "PointwiseOps.h"

void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        y[i][j][k] = x[i][j][k];
}
```

OpenMP parallelization deferred until later (when we separate out special cases for any of x, y, z being aliases of each other)

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const float scale)
{
    // Should we use OpenMP parallel for here?
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
}
```

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Omitted (for us)

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- N

Subtracting a scalar (single number) from all entries of a vector

(useful for some simulations of fluids, not necessary for our examples here; feel free to omit all red-colored instructions)

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
$\leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
$\leftarrow \rho^{\text{new}}/\rho$
$\leftarrow \rho^{\text{new}}$
$\leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
or
edure

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

Without a "preconditioner" we assume that M=identity. These two commands become "vector copy" directives

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

For vectors **x**, **y**
Copy(**x**, **y**) is defined as:

```
for i=1 ... N
    yi = xi
```

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as ***Copy(r, p)***

Implemented as ***Copy(r, z)***

Pointwise Ops (PointwiseOps.h)

LaplaceSolver/LaplaceSolver_0_0

```
#pragma once

#include "Parameters.h"

// Copy array x into y
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM]);

// Scale array x by given number, add y, and write result into z
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale);
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "PointwiseOps.h"
```

```
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        y[i][j][k] = x[i][j][k];
}
```

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const float scale)
{
    // Should we use OpenMP parallel for here?
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
}
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "PointwiseOps.h"

void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        y[i][j][k] = x[i][j][k];
}
```

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const float scale)
{
    // Should we use OpenMP parallel for here?
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
}
```

For vectors **x**, **y**
Copy(**x**, **y**) is defined as:

for i=1 ... N
$y_i = x_i$

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

For vectors **x**, **y**
InnerProduct(**x**, **y**) is defined as:

```
result = 0
for I = 1 ... N
    result += yi * xi
```

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

re MGPCG($\mathbf{f}$, $\mathbf{x}$)
$\mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
$< \nu_{max}$) **then return**
$\mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
$= 0$ **to** k_{max} **do**
$\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
$\alpha \leftarrow \rho/\sigma$
$\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
if ($\nu < \nu_{max}$ **or** $k = k_{max}$) **then**
$\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
return
end if
$\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{new} \leftarrow \mathbf{z}^T\mathbf{r}$
$\beta \leftarrow \rho^{new}/\rho$
$\rho \leftarrow \rho^{new}$
$\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
end for
end procedure

Reduction Ops (Reductions.h)

LaplaceSolver/LaplaceSolver_0_0

```
#pragma once

#include "Parameters.h"

// Compute the maximum absolute value among the array elements
float Norm(const float (&x)[XDIM][YDIM][ZDIM]);

// Compute the "dot product" between the two arrays
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM]);
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}
```

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}
```

For vectors **x**, **y**
InnerProduct(**x**, **y**) is defined as:

```
result = 0
for I = 1 ... N
   result += yi * xi
```

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

For false sharing, imaging
float partial_result [N] = {0,0,0...0};

Add up all partial -> Final sum but suffers from false sharing

Avoid this by padding

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}
```

Observation #1: A special treatment of the reduction operation is needed to avoid false sharing (in-class discussion here …)

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}
```

Observation #2: Casting to double is needed to avoid loss of precision (more of a numerical analysis issue than a parallel programming one)

Floating point can wrap-around on overflow, this is why double precision is used.

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Implemented as
rho = InnerProduct(p, r)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

Implemented as
sigma = InnerProduct(p, z)

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

Implemented as
rho_new = InnerProduct(z, r)

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
μ, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\rho \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Kernels

- Multiply()
- Saxpy()
- Subtract()
- Copy()
- Inner_Product()
- Norm()

Reduction Ops (Reductions.h)

LaplaceSolver/LaplaceSolver_0_0

```
#pragma once

#include "Parameters.h"

// Compute the maximum absolute value among the array elements
float Norm(const float (&x)[XDIM][YDIM][ZDIM]);

// Compute the "dot product" between the two arrays
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM]);
```

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}

float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

Same considerations about reduction/sharing as in InnerProduct

Benchmark launcher (main.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "ConjugateGradients.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    float *fRaw = new float [XDIM*YDIM*ZDIM];
    float *pRaw = new float [XDIM*YDIM*ZDIM];
    float *rRaw = new float [XDIM*YDIM*ZDIM];
    float *zRaw = new float [XDIM*YDIM*ZDIM];
    array_t x = reinterpret_cast<array_t>(*xRaw);
    array_t f = reinterpret_cast<array_t>(*fRaw);
    array_t p = reinterpret_cast<array_t>(*pRaw);
    array_t r = reinterpret_cast<array_t>(*rRaw);
    array_t z = reinterpret_cast<array_t>(*zRaw);

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        timer.Stop("Initialization : ");
    }
    // Call Conjugate Gradients algorithm
    ConjugateGradients(x, f, p, r, z);
    return 0;
}
```

Benchmark launcher (main.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "ConjugateGradients.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    float *fRaw = new float [XDIM*YDIM*ZDIM];
    float *pRaw = new float [XDIM*YDIM*ZDIM];
    float *rRaw = new float [XDIM*YDIM*ZDIM];
    float *zRaw = new float [XDIM*YDIM*ZDIM];
    array_t x = reinterpret_cast<array_t>(*xRaw);
    array_t f = reinterpret_cast<array_t>(*fRaw);
    array_t p = reinterpret_cast<array_t>(*pRaw);
    array_t r = reinterpret_cast<array_t>(*rRaw);
    array_t z = reinterpret_cast<array_t>(*zRaw);

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        timer.Stop("Initialization : ");
    }
    // Call Conjugate Gradients algorithm
    ConjugateGradients(x, f, p, r, z);
    return 0;
}
```

*Allocate variables **x**, **f**, **r**, **p**, **z***
(in flattened representation)

Test case: Preconditioned Conjugate Gradients

$$\mathcal{L}\mathbf{x} = \mathbf{f}$$

Vectors or arrays?

- Variables ***x***, ***f***, ***r***, ***p***, ***z*** are shown in this pseudocode as mathematical "vectors"
- However, their real representation is 3D arrays (grid-based arrays)
- e.g. u[..][..][..] is the actual implementation of ***x***!
- Instead of "converting" them to traditional vectors, we emulate operations in their native representation.

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Non-boldfaced symbols are numbers (scalars)

Benchmark launcher (main.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "ConjugateGradients.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    float *fRaw = new float [XDIM*YDIM*ZDIM];
    float *pRaw = new float [XDIM*YDIM*ZDIM];
    float *rRaw = new float [XDIM*YDIM*ZDIM];
    float *zRaw = new float [XDIM*YDIM*ZDIM];
    array_t x = reinterpret_cast<array_t>(*xRaw);
    array_t f = reinterpret_cast<array_t>(*fRaw);
    array_t p = reinterpret_cast<array_t>(*pRaw);
    array_t r = reinterpret_cast<array_t>(*rRaw);
    array_t z = reinterpret_cast<array_t>(*zRaw);

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        timer.Stop("Initialization : ");
    }
    // Call Conjugate Gradients algorithm
    ConjugateGradients(x, f, p, r, z);
    return 0;
}
```

… and reshape them so they are usable as 3D arrays

Benchmark launcher (main.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "ConjugateGradients.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    float *fRaw = new float [XDIM*YDIM*ZDIM];
    float *pRaw = new float [XDIM*YDIM*ZDIM];
    float *rRaw = new float [XDIM*YDIM*ZDIM];
    float *zRaw = new float [XDIM*YDIM*ZDIM];
    array_t x = reinterpret_cast<array_t>(*xRaw);
    array_t f = reinterpret_cast<array_t>(*fRaw);
    array_t p = reinterpret_cast<array_t>(*pRaw);
    array_t r = reinterpret_cast<array_t>(*rRaw);
    array_t z = reinterpret_cast<array_t>(*zRaw);

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        timer.Stop("Initialization : ");
    }
    // Call Conjugate Gradients algorithm
    ConjugateGradients(x, f, p, r, z);
    return 0;
}
```

Initialize the values of the right-hand-side (as well as initial guess of the solution)

The temperatures on the periphery

Benchmark utilities (Utilities.h)

LaplaceSolver/LaplaceSolver_0_2

```
#pragma once
#include "Parameters.h"

#include <string>

void Clear(float (&x)[XDIM][YDIM][ZDIM]);
void InitializeProblem(float (&x)[XDIM][YDIM][ZDIM], float (&b)[XDIM][YDIM][ZDIM]);
void WriteAsImage(const std::string& filenamePrefix, const float (&x)[XDIM][YDIM][ZDIM],
                  const int count, const int axis, const int slice);
```

Benchmark utilities (Utilities.h)

LaplaceSolver/LaplaceSolver_0_2

```
#pragma once
#include "Parameters.h"

#include <string>

void Clear(float (&x)[XDIM][YDIM][ZDIM]);
void InitializeProblem(float (&x)[XDIM][YDIM][ZDIM], float (&b)[XDIM][YDIM][ZDIM]);
void WriteAsImage(const std::string& filenamePrefix, const float (&x)[XDIM][YDIM][ZDIM],
                  const int count, const int axis, const int slice);
```

Examine Clear() and Initialize()
(we'll look at WriteAsImage() later …)

Benchmark utilities (Utilities.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "Utilities.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <stdexcept>

void Clear(float (&x)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        x[i][j][k] = 0.;
}
```

Vector "b" set to zero
Initial Guess "x" zeroed out
(but incorporating boundary values)

```
void InitializeProblem(float (&x)[XDIM][YDIM][ZDIM], float (&b)[XDIM][YDIM][ZDIM]){

    // Start by zeroing out x and b
    Clear(x);
    Clear(b);

    // Make some of the boundary of values of x non-zero
    // (this operation is far too simple to be worth parallelizing)

    for(int i = XDIM/4; i < 3*(XDIM/4); i++)
    for(int j = XDIM/4; j < 3*(XDIM/4); j++)
        x[i][j][0] = 1.;
}
[...]
```

Benchmark utilities (Utilities.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
#include "Utilities.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <stdexcept>
```

```
void Clear(float (&x)[XDIM][YDIM][ZDIM])
{
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        x[i][j][k] = 0.;
}
```

Clear() zeroes out its argument
(not necessarily the "x" of Conjugate Gradients!)

```
void InitializeProblem(float (&x)[XDIM][YDIM][ZDIM], float (&b)[XDIM][YDIM][ZDIM]){

    // Start by zeroing out x and b
    Clear(x);
    Clear(b);

    // Make some of the boundary of values of x non-zero
    // (this operation is far too simple to be worth parallelizing)

    for(int i = XDIM/4; i < 3*(XDIM/4); i++)
    for(int j = XDIM/4; j < 3*(XDIM/4); j++)
        x[i][j][0] = 1.;
}
[...]
```

Benchmark utilities (Utilities.cpp)

LaplaceSolver/LaplaceSolver_0_2

```
[...]
void WriteAsImage(const std::string& filenamePrefix, const float (&x)[XDIM][YDIM][ZDIM],
                  const int count, const int axis, const int slice)
{
    std::ostringstream filename;
    filename << filenamePrefix << "." << std::setfill('0') << std::setw(4) << count << ".pgm";
    std::ofstream output(filename.str());
    output << "P2" << std::endl;
    switch(axis){
        case 0: output << YDIM << " " << ZDIM << std::endl; break;
        case 1: output << XDIM << " " << ZDIM << std::endl; break;
        case 2: output << XDIM << " " << YDIM << std::endl; break;
        default: throw std::logic_error("Invalid axis in WriteAsImage()");}
    output << "255" << std::endl;
    switch(axis){
    case 0:
        for (int j = 0; j < YDIM; j++){
            for (int k = 0; k < ZDIM; k++)
                output << (int)(x[slice][j][k]*255.0) << " ";
            output << std::endl;}
        break;
    case 1:
        for (int i = 0; i < XDIM; i++){
            for (int k = 0; k < ZDIM; k++)
                output << (int)(x[i][slice][k]*255.0) << " ";
            output << std::endl;}
        break;
    case 2:
        for (int i = 0; i < XDIM; i++){
            for (int j = 0; j < YDIM; j++)
                output << (int)(x[i][j][slice]*255.0) << " ";
            output << std::endl;}
        break;
    default: throw std::logic_error("Invalid axis in WriteAsImage()");}
output.close();}
```

Outputs a “slice” of the variable “x” as a PGM (grayscale) image

CG Routine (ConjugateGradients.h)

LaplaceSolver/LaplaceSolver_0_2

```
#pragma once

#include "Parameters.h"

void ConjugateGradients(
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations = true);
```

Solver Parameters (Parameters.h)

LaplaceSolver/LaplaceSolver_0_2

```
#pragma once

#define XDIM 256
#define YDIM 256
#define ZDIM 256

constexpr int kMax = 1000;
constexpr float nuMax = 1e-3;
```

Solver Parameters (Parameters.h)

LaplaceSolver/LaplaceSolver_0_2

```
#pragma once

#define XDIM 256
#define YDIM 256
#define ZDIM 256

constexpr int kMax = 1000;
constexpr float nuMax = 1e-3;
```

Termination criteria

Solver body (ConjugateGradients.cpp)

```
#include "Laplacian.h"
#include "Parameters.h"
#include "PointwiseOps.h"
#include "Reductions.h"
#include "Utilities.h"

#include <iostream>

void ConjugateGradients(
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    ComputeLaplacian(x, z);
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
```

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Solver body (ConjugateGradients.cpp)

```
#include "Laplacian.h"
#include "Parameters.h"
#include "PointwiseOps.h"
#include "Reductions.h"
#include "Utilities.h"

#include <iostream>

void ConjugateGradients(
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    ComputeLaplacian(x, z);
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
```

Solver body (ConjugateGradients.cpp)

```
// Beginning of loop from Line 5
for(int k=0;;k++)
{
    std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

    // Algorithm : Line 6
    ComputeLaplacian(p, z);
    float sigma=InnerProduct(p, z);

    // Algorithm : Line 7
    float alpha=rho/sigma;

    // Algorithm : Line 8
    Saxpy(z, r, r, -alpha);
    nu=Norm(r);

    // Algorithm : Lines 9-12
    if (nu < nuMax || k == kMax) {
        Saxpy(p, x, x, alpha);
        std::cout << "Conjugate Gradients terminated after " << k << " iterations; residual norm (nu)
= " << nu << std::endl;
        if (writeIterations) WriteAsImage("x", x, k, 0, 127);
        return;
    }

    // Algorithm : Line 13
    Copy(r, z);
    float rho_new = InnerProduct(z, r);

    // Algorithm : Line 14
    float beta = rho_new/rho;
```

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Solver body (ConjugateGradients.cpp)

```
// Beginning of loop from Line 5
for(int k=0;;k++)
{
    std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

    // Algorithm : Line 6
    ComputeLaplacian(p, z);
    float sigma=InnerProduct(p, z);

    // Algorithm : Line 7
    float alpha=rho/sigma;

    // Algorithm : Line 8
    Saxpy(z, r, r, -alpha);
    nu=Norm(r);

    // Algorithm : Lines 9-12
    if (nu < nuMax || k == kMax) {
        Saxpy(p, x, x, alpha);
        std::cout << "Conjugate Gradients terminated after " << k << " iterations; residual norm (nu)
= " << nu << std::endl;
        if (writeIterations) WriteAsImage("x", x, k, 0, 127);
        return;
    }

    // Algorithm : Line 13
    Copy(r, z);
    float rho_new = InnerProduct(z, r);

    // Algorithm : Line 14
    float beta = rho_new/rho;
```

Solver body (ConjugateGradients.cpp)

```
        Saxpy(z, r, r, -alpha);
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            Saxpy(p, x, x, alpha);
            std::cout << "Conjugate Gradients terminated after " << k << " iterations; residual norm (nu)
= " << nu << std::endl;
            if (writeIterations) WriteAsImage("x", x, k, 0, 127);
            return;
        }

        // Algorithm : Line 13
        Copy(r, z);
        float rho_new = InnerProduct(z, r);

        // Algorithm : Line 14
        float beta = rho_new/rho;

        // Algorithm : Line 15
        rho=rho_new;

        // Algorithm : Line 16
        Saxpy(p, x, x, alpha);
        Saxpy(p, r, p, beta);

        if (writeIterations) WriteAsImage("x", x, k, 0, 127);
    }
}
```

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**