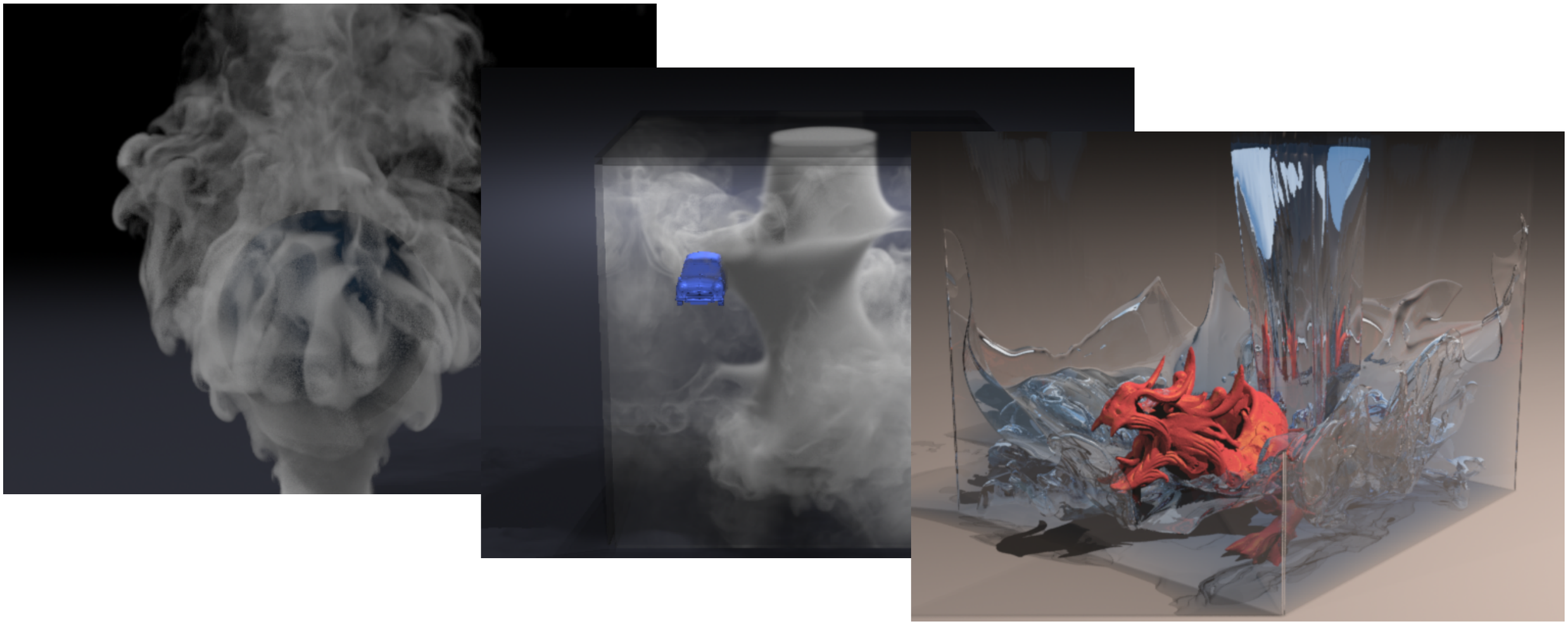


Sparse Matrix Computations

Simple Preconditioner Application

New main routine (main.cpp)

LaplaceSolver_1_3

```
[.. .]
int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];
    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [.. .]
    array_t z = reinterpret_cast<array_t>(*zRaw);

 CSRMatrix matrix;
    CSRMatrix L;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildLaplacianMatrix(); // This takes a while ...
        L = BuildPreconditionerMatrix(); // This takes a while ...

        // Make sure that the preconditioner is a valid lower-triangular CSR matrix ...
        L.CheckLowerTriangular();

        timer.Stop("Initialization : ");
    }

    // Call Conjugate Gradients algorithm
    ConjugateGradients(matrix, L, x, f, p, r, z, true);
[.. .]
}
```

New main routine (main.cpp)

LaplaceSolver_1_3

```
[.. .]
int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];
    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [.. .]
    array_t z = reinterpret_cast<array_t>(*zRaw);

 CSRMatrix matrix;
    CSRMatrix L;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildLaplacianMatrix(); // This takes a while ...
        L = BuildPreconditionerMatrix(); // This takes a while ...

        // Make sure that the preconditioner is a valid lower-triangular CSR matrix ...
        L.CheckLowerTriangular();

        timer.Stop("Initialization : ");
    }

    // Call Conjugate Gradients algorithm
    ConjugateGradients(matrix, L, x, f, p, r, z, true);
[.. .]
}
```

New Laplacian (Laplacian.cpp)

LaplaceSolver_1_3

```
[...]

CSRMatrix BuildPreconditionerMatrix()
{
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    // A cheap but effective approximation of an Incomplete Cholesky preconditioner
    // NOTE : You don't need to explain *why* this ends up being an effective preconditioner
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 3.05; // <-- Don’t worry!
        if (i > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i-1, j, k) ) = -1.;
        if (j > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j-1, k) ) = -1.;
        if (k > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k-1) ) = -1.;}

    // Need to put part of the "identity" matrix on nodes of the margin
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        if ( i == 0 || i == XDIM-1 || j == 0 || j == YDIM-1 || k == 0 || k == ZDIM-1 )
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 1.;
    return matrixHelper.ConvertToCSRMatrix();
}[.. .]
```

New Laplacian (Laplacian.cpp)

LaplaceSolver_1_3

```
[...]

CSRMatrix BuildPreconditionerMatrix()
{
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    // A cheap but effective approximation of an Incomplete Cholesky preconditioner
    // NOTE : You don't need to explain *why* this ends up being an effective preconditioner
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 3.05; // <-- Don’t worry!
        if (i > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i-1, j, k) ) = -1.;
        if (j > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j-1, k) ) = -1.;
        if (k > 1)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k-1) ) = -1.;}

    // Need to put part of the "identity" matrix on nodes of the margin
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        if ( i == 0 || i == XDIM-1 || j == 0 || j == YDIM-1 || k == 0 || k == ZDIM-1 )
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 1.;
    return matrixHelper.ConvertToCSRMatrix();
}[.. .]
```

Forward/BackSub (Substitutions.cpp)

LaplaceSolver_1_3

```
[...]

void ForwardSubstitution(CSRMatrix& L, float *x) { // In-place

    const auto N = L.mSize;
    const auto rowOffsets = L.GetRowOffsets();
    const auto columnIndices = L.GetColumnIndices();
    const auto values = L.GetValues();

    for (int i = 0; i < N; i++) {
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]-1; k++)
            x[i] -= values[k] * x[columnIndices[k]]; // Move all terms of the i-th equation
                                                     // involving x[j]'s (with j<i) to the right
                                                     // hand side (which is stored in x[i]
        x[i] /= values[rowOffsets[i+1]-1]; // Divide by the diagonal matrix entry
    }
}

[.. .]
```

Forward/BackSub (Substitutions.cpp)

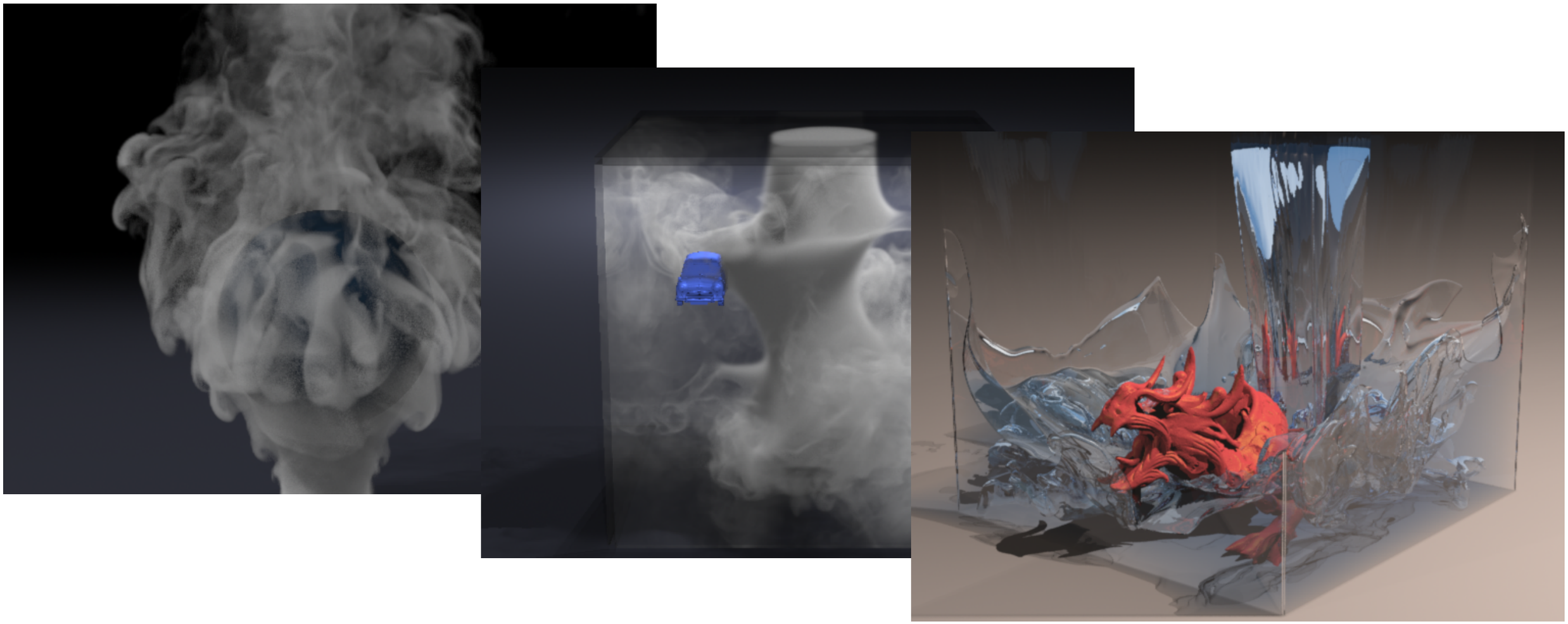
LaplaceSolver_1_3

```
[...]

void BackwardSubstitution(CSRMatrix& Ut, float *x) { // In-place

    const auto N = Ut.mSize;
    const auto columnOffsets = Ut.GetRowOffsets();
    const auto rowIndices = Ut.GetColumnIndices();
    const auto values = Ut.GetValues();

    for (int j = N-1; j >= 0; j--) {
        x[j] /= values[columnOffsets[j+1]-1]; // Divide by the diagonal matrix entry
        for (int k = columnOffsets[j+1]-2; k >= columnOffsets[j]; k--)
            x[rowIndices[k]] -= values[k] * x[j];
// Annihilate the value of x[j] from all equations
// (indexed lower than the j-th equation) where such
// a term exists. The right hand side of the i-th
// equation is stored in x[i]
    }

[.. .]
```

New CG routine (ConjugateGradients.cpp)

LaplaceSolver_1_3

```
void ConjugateGradients(
    CSRMatrix& matrix, CSRMatrix& L,
    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 13
        Copy(r, z);
        ForwardSubstitution(L, &z[0][0][0]);
        BackwardSubstitution(L, &z[0][0][0]);
        float rho_new = InnerProduct(z, r);
[...]
}
```



Sparse Matrix Computations

Introduction to Intel MKL (BLAS/Sparse BLAS L1/2)

The Intel Math Kernel Library (MKL)

Getting started :

- Download from : https://software.intel.com/en-us/mkl
- Documentation : https://software.intel.com/en-us/mkl/documentation/view-all
- Compilation options https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/
- Easiest with Intel Compiler, but supported on all platforms!

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.h)

LaplaceSolver_1_4

```
#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);

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

MKL-based vector ops (PointwiseOps.h)

LaplaceSolver_1_4

```
#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);

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

Special case, when the output vector (z) is the same as the one we add to (y)

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#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++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}
```

Special case, when the output vector (z) is the same as the one we add to (y)

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#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++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#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++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#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++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

Original version of Saxpy (output not the same as an argument)

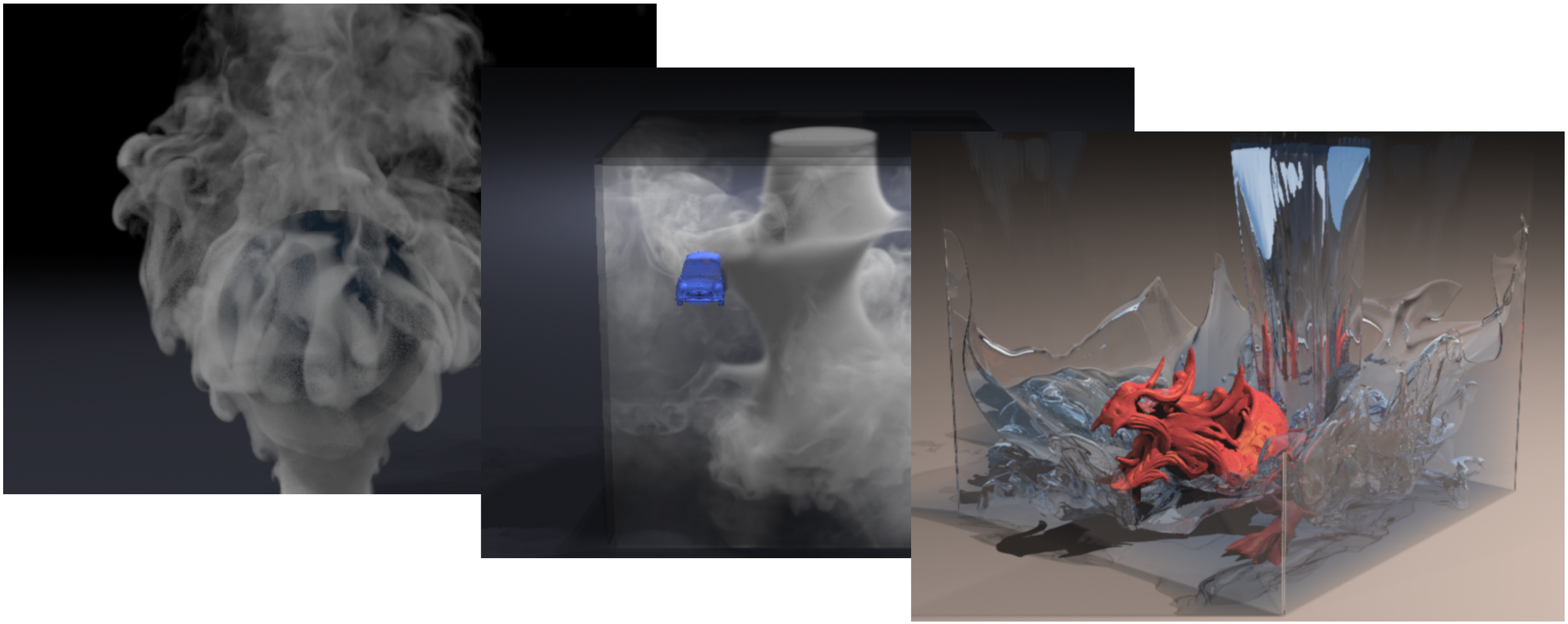
```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

Special case implementation applies here!

```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```



Sparse Matrix Computations

Introduction to Intel MKL (BLAS/Sparse BLAS L1/2)