

# Numeric simulation using finite elements

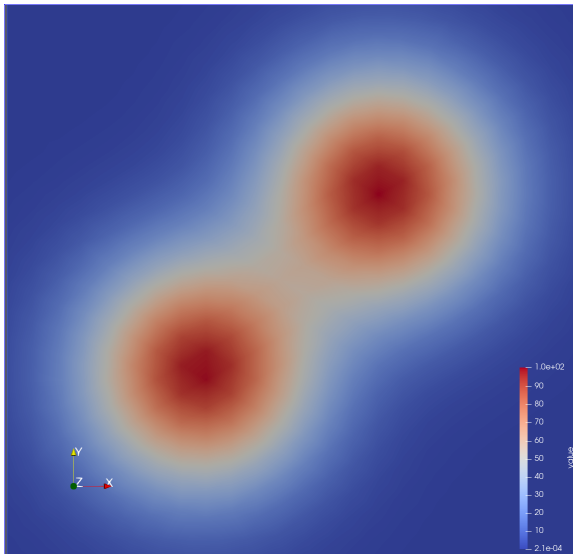
Sparse but not scarce

Charlotte Vavourakis, Martin Fasser

University of Innsbruck

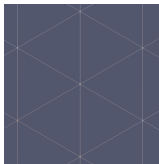
June 23, 2023

# Original implementation: dense matrix

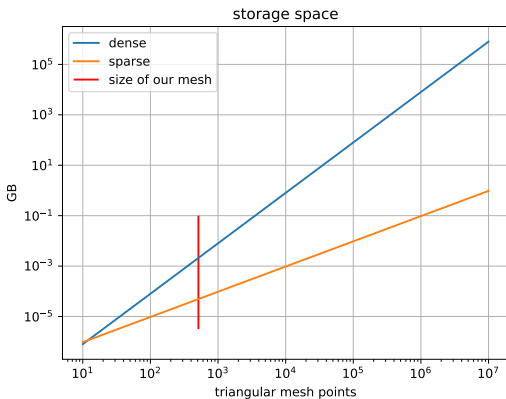


# Why Sparse?

## Storage space



1.  $N$  = number of mesh points
2.  $\approx 6 \cdot N$  -nonzero elements in matrix
3. storage for dense matrix ( $\propto N^2$ ):  
2.11 MB
4. storage for sparse matrix ( $\propto N$ ):  
0.05 MB



# Assignment: Sparse but not scarce

- Convert PDE solver matrix to sparse format and compare the performance with your dense matrix implementation.
- Use library implementation and compare it with a self-implemented CSR format.

# Different Sparse Formats

- sparse - manual
  - values = [5.3, 1.5, 4.2, 3.1, 2, 2.2, 1.9]
  - pos = [1, 4, 5, 7, 9, 18, 20]
- sparse - coo (coordinate list)
  - values = [5.3, 1.5, 4.2, 3.1, 2, 2.2, 1.9]
  - col\_ind = [1, 4, 0, 2, 4, 3, 0]
  - row\_ind = [0, 0, 1, 1, 1, 3, 4]
- sparse - csr (compressed row storage)
  - values = [5.3, 1.5, 4.2, 3.1, 2, 2.2, 1.9]
  - col\_ind = [1, 4, 0, 2, 4, 3, 0]
  - row\_ptr = [0, 2, 5, 5, 6, 7]

$$\begin{pmatrix} 0 & 5.3 & 0 & 0 & 1.5 \\ 4.2 & 0 & 3.1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.2 & 0 \\ 1.9 & 0 & 0 & 0 & 0 \end{pmatrix}$$

# Matrix assembly

# Conversion coo to csr

coo format:

row\_ind = [0, 0, 1, 1, 1, 3, 4]

```
for (int i = 0; i < row_ind.size(); i++) {  
    row_ptr[row_ind[i] + 1]++;  
}
```

how many elements in each row:

row\_step = [0, 2, 3, 0, 1, 1]

```
for (int i = 0; i < dim; i++) {  
    row_ptr[i + 1] += row_ptr[i];  
}
```

csr format:

row\_ptr = [0, 2, 5, 5, 6, 7]

$$\begin{pmatrix} 0 & 5.3 & 0 & 0 & 1.5 \\ 4.2 & 0 & 3.1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.2 & 0 \\ 1.9 & 0 & 0 & 0 & 0 \end{pmatrix}$$

# Sparse - manual

- sparse - manual
  - values = [5.3, 1.5, 4.2, 3.1, 2, 2.2, 1.9]
  - pos = [1, 4, 5, 7, 9, 18, 20]
- matrix-vector multiplication

```
for (int k = 0; k < _count; k++) {  
    const int i = _pos[k] / _dim;  
    const int j = _pos[k] % _dim;  
    res[i] += _val[k] * u[j];  
}
```

$$\begin{pmatrix} 0 & 5.3 & 0 & 0 & 1.5 \\ 4.2 & 0 & 3.1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.2 & 0 \\ 1.9 & 0 & 0 & 0 & 0 \end{pmatrix}$$



# Sparse - CSR

- sparse - csr (compressed row storage)
  - values = [5.3, 1.5, 4.2, 3.1, 2, 2.2, 1.9]
  - col\_ind = [1, 4, 0, 2, 4, 3, 0]
  - row\_ptr = [0, 2, 5, 5, 6, 7]
- matrix-vector multiplication

```
for (int i=0; i<u.size(); i++) {  
    res[i]=0;  
    for (int j= _row_ptr[i]; j<_row_ptr[i+1]; j++) {  
        res[i]+= _val[j]*u[_col_ind[j]];  
    }  
}
```

$$\begin{pmatrix} 0 & 5.3 & 0 & 0 & 1.5 \\ 4.2 & 0 & 3.1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.2 & 0 \\ 1.9 & 0 & 0 & 0 & 0 \end{pmatrix}$$

# Solution: Sparse but not scarce

- two self-implemented classes for sparse matrices: non-CSR and CSR format.
- Eigen library
  - added to repository as a git submodule
  - SparseMatrix class
  - SparseMatrix::makeCompressed() conversion to CSR format
  - forward Euler simplified using matrix multiplication notation:

`Eigen::SparseMatrix B * Eigen::VectorXd u`

# Google Benchmark

- [github.com/google/benchmark](https://github.com/google/benchmark)
- a library to benchmark code snippets (functions)
- added to repository as a git submodule

```
#include <benchmark/benchmark.h>

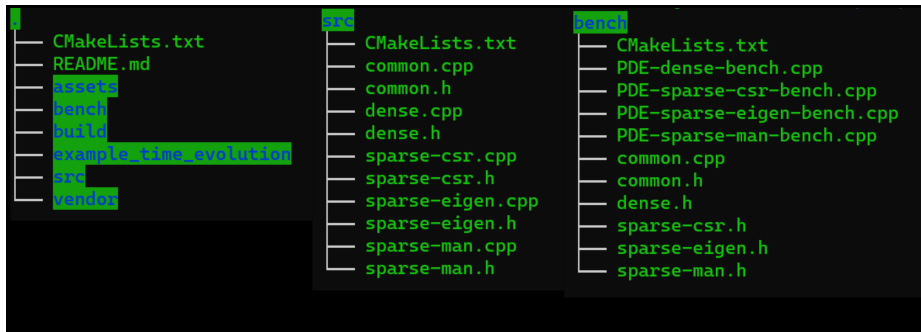
static void BM_SomeFunction(benchmark::State& state) {
    // Perform setup here
    for (auto _ : state) {
        // This code gets timed
        SomeFunction();
    }
}

// Register the function as a benchmark
BENCHMARK(BM_SomeFunction);

// Run the benchmark
BENCHMARK_MAIN();
```

# Code organization

- Cmake build 8 targets, common + unique code
- 4 versions of PDE solver
- corresponding benchmarks, testing functions unique to each implementation



# Benchmark example output

Run on (4 X 2995 MHz CPU s)

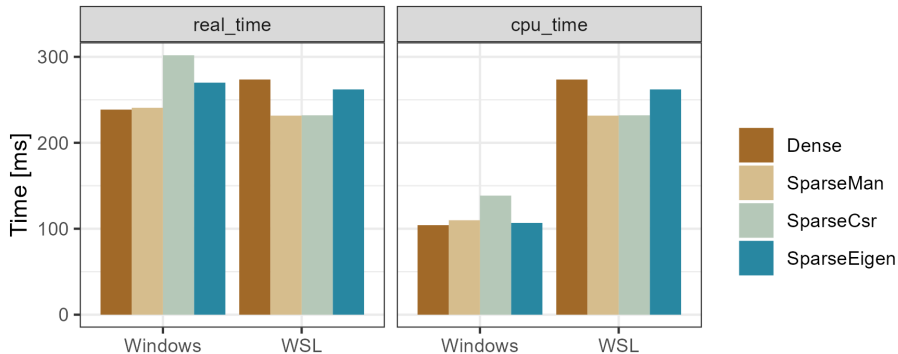
CPU Caches:

L1 Data 48 KiB (x4)  
L1 Instruction 32 KiB (x4)  
L2 Unified 1280 KiB (x4)  
L3 Unified 12288 KiB (x1)

Benchmark	Time	CPU	Iterations
assembleMatrixSparseEigenBench1/iterations:30	308 ms	149 ms	30
timeEvolutionSparseEigenBench2/iterations:30	36.7 ms	17.2 ms	30
odeSparseEigenBench3/0/iterations:2	0.360 s	0.180 s	2
odeSparseEigenBench3/1/iterations:2	0.373 s	0.211 s	2
odeSparseEigenBench3/2/iterations:2	0.372 s	0.195 s	2
odeSparseEigenBench3/3/iterations:2	0.363 s	0.211 s	2
odeSparseEigenBench3/4/iterations:2	0.347 s	0.234 s	2
odeSparseEigenBench3/5/iterations:2	0.335 s	0.195 s	2
odeSparseEigenBench3/6/iterations:2	0.382 s	0.141 s	2
odeSparseEigenBench3/7/iterations:2	0.331 s	0.164 s	2
odeSparseEigenBench3/8/iterations:2	0.341 s	0.172 s	2
odeSparseEigenBench3/9/iterations:2	0.353 s	0.133 s	2

# Benchmark assembly matrix B

- 30 iterations



# Benchmark time evolution

Graph or table comparing 4 implementations

# Benchmark assembly B + time evolution for different dt

Graph or table comparing 4 implementations



# Conclusions

- Trade off own dense vs sparse format:
  - With the dense format and increasing mesh size, will run out of memory far sooner
  - Time needed for matrix assembly unpredictable (dependence on format and machine used)
  - Significant 100-fold time-speed up using sparse format for time evolution
- Trade off own sparse format vs CSR-format:
  - Extra time needed for converting COO to CSR
  - Speed up at time evolution
- Trade-off library vs own implementation:
  - Time spent on learning the library (once) vs on correctly implementing the sparse format
  - With Eigen even more speed up and code simplification using matrix product expressions

Questions?