Numeric simulation using finite elements

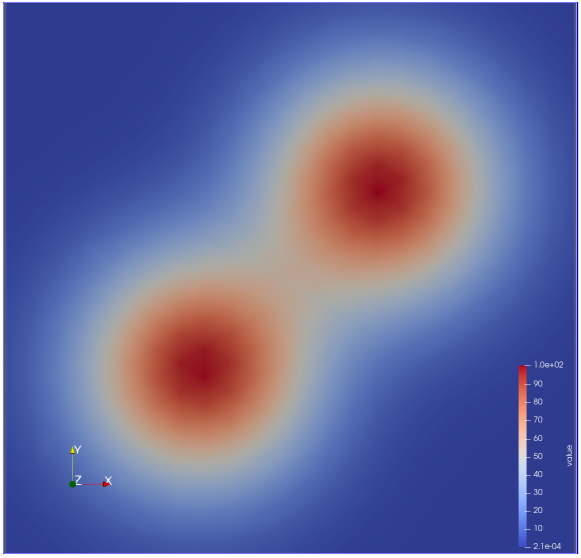
Sparse but not scarce

Martin Fasser, Charlotte Vavourakis

University of Innsbruck

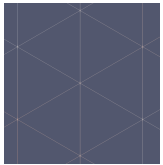
June 26, 2023

Original implementation: dense matrix

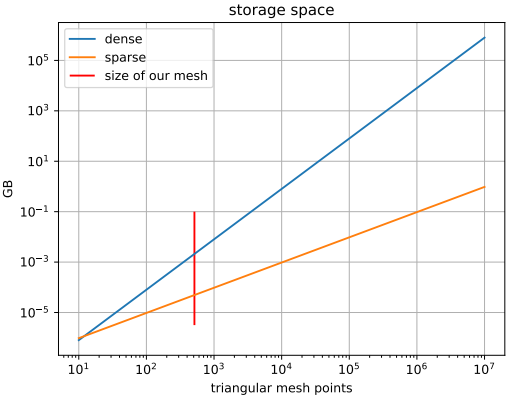


Why Sparse?

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
  - $\text{values} = \begin{bmatrix} 5.3, & 1.5, & 4.2, & 3.1, & 2, & 2.2, & 1.9 \end{bmatrix}$
  - $\text{pos} = \begin{bmatrix} 1, & 4, & 5, & 7, & 9, & 18, & 20 \end{bmatrix}$
- sparse - coo (coordinate list)
  - $\text{values} = \begin{bmatrix} 5.3, & 1.5, & 4.2, & 3.1, & 2, & 2.2, & 1.9 \end{bmatrix}$
  - $\text{col\_ind} = \begin{bmatrix} 1, & 4, & 0, & 2, & 4, & 3, & 0 \end{bmatrix}$
  - $\text{row\_ind} = \begin{bmatrix} 0, & 0, & 1, & 1, & 1, & 3, & 4 \end{bmatrix}$
- sparse - csr (compressed row storage)
  - $\text{values} = \begin{bmatrix} 5.3, & 1.5, & 4.2, & 3.1, & 2, & 2.2, & 1.9 \end{bmatrix}$
  - $\text{col\_ind} = \begin{bmatrix} 1, & 4, & 0, & 2, & 4, & 3, & 0 \end{bmatrix}$
  - $\text{row\_ptr} = \begin{bmatrix} 0, & 2, & 5, & 5, & 6, & 7 \end{bmatrix}$

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

Matrix assembly

class $\mathrm{sparse\_mat}$:

```
sparse_mat::sparse_mat() {}
sparse_mat::sparse_mat(int dim, vector<int> col_ind,
        vector<int> row_ind, vector<double> val) {
        _dim=dim;
        _col_ind=col_ind;
        _row_ind=row_ind;
        _val=val;
        _row_ptr=convert_coo_to_csr(dim, row_ind);
    }
```

preliminaries:

```
vector<int> col_ind;
vector<int> row_ind;
vector<double> values;
```

filling:

```
if (Bij != 0) {
    Bij /= H;
    row_ind.push_back(i);
    col_ind.push_back(j);
    values.push_back(Bij);
```

constructing the $\mathrm{sparse\_mat}$ object:

```
// construct B
sparse_mat B;
B = sparse_mat(points.size(), col_ind, row_ind, values);
return B;
```

Matrix assembly: Conversion coo to csr

coo format:

$\text{row\_ind} = \begin{bmatrix} 0, & 0, & 1, & 1, & 1, & 3, & 4 \end{bmatrix}$

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

how many elements in each row:

$\text{row\_step} = \begin{bmatrix} 0, & 2, & 3, & 0, & 1, & 1 \end{bmatrix}$

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

csr format:

$\text{row\_ptr} = \begin{bmatrix} 0, & 2, & 5, & 5, & 6, & 7 \end{bmatrix}$

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

Matrix-Vector multiplication: Sparse - manual

- sparse - manual
  - $\text{values} = \begin{bmatrix} 5.3, & 1.5, & 4.2, & 3.1, & 2, & 2.2, & 1.9 \end{bmatrix}$
  - $\text{pos} = \begin{bmatrix} 1, & 4, & 5, & 7, & 9, & 18, & 20 \end{bmatrix}$
- 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];
}
```

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

Matrix-Vector multiplication: Sparse - CSR

- sparse - csr (compressed row storage)
  - $\text{values} = \begin{bmatrix} 5.3 & 1.5 & 4.2 & 3.1 & 2 & 2.2 & 1.9 \end{bmatrix}$
  - $\text{col\_ind} = \begin{bmatrix} 1 & 4 & 0 & 2 & 4 & 3 & 0 \end{bmatrix}$
  - $\text{row\_ptr} = \begin{bmatrix} 0 & 2 & 5 & 5 & 6 & 7 \end{bmatrix}$
- 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

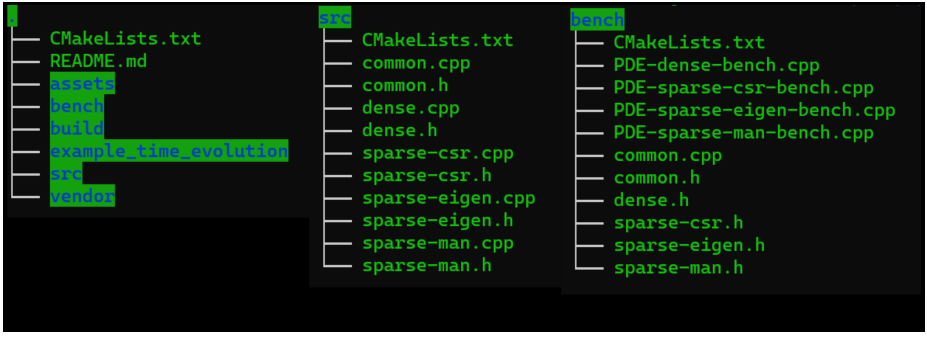
- github.com/google/benchmark
- a library to benchmark code snippets (functions)
- added to repository as a git submodule

```
#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



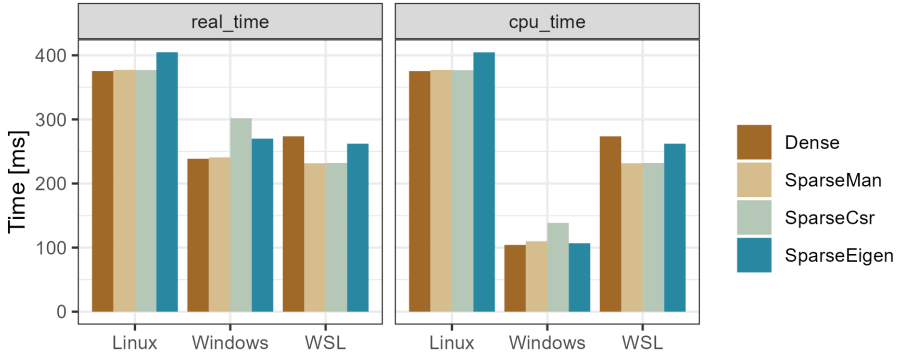
Benchmark example output

```
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

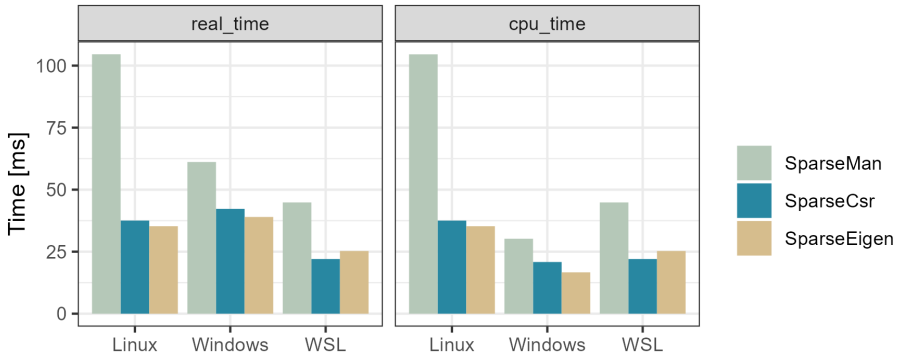
Benchmark assembly matrix B

- 30 iterations
- Linux (Martin's machine), Windows and WSL (Charlotte's machine)



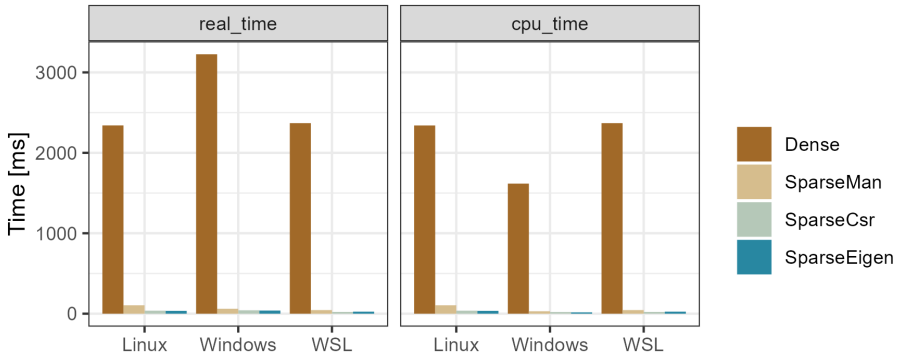
Benchmark time evolution

- 30 iterations



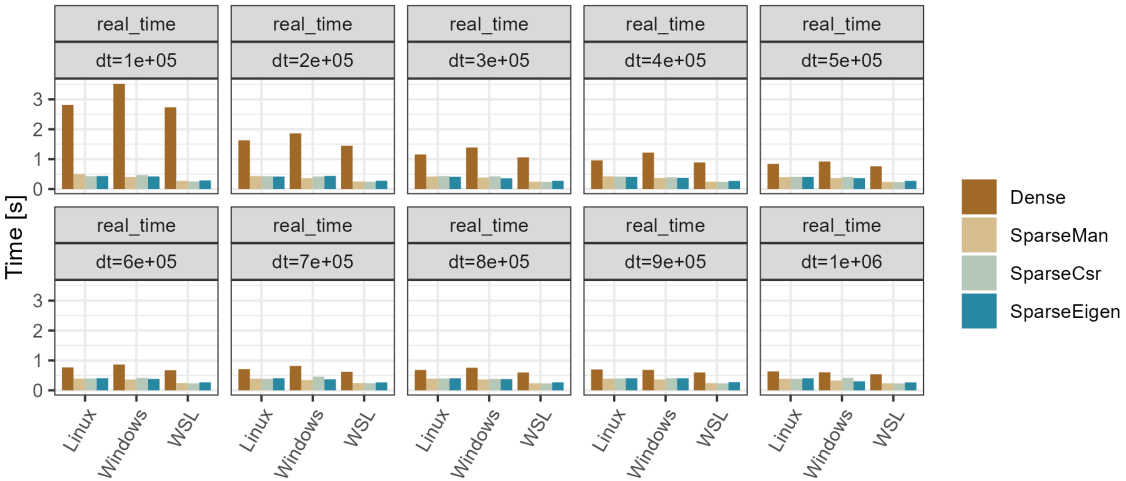
Benchmark time evolution

- 30 iterations



Benchmark time evolution

- 2 iterations, for different timesteps (dt)



Conclusions

- Sparse formats preferred over dense format:
  - Memory issues for large mesh sizes
  - Time needed for matrix assembly similar (dependence on machine and OS used?)
  - Significant 100-fold time-speed up using sparse format for time evolution
- Tradeoff own sparse format vs CSR-format:
  - Extra time needed for converting COO to CSR
  - Speed up at time evolution
- Tradeoff library vs own implementation:
  - Time spent on learning the library (once) vs on correctly implementing the sparse format
  - With Eigen code simplification using matrix product expressions

Questions?