Interactive FEM CPU deformable simulation:

What you should (and shouldn't) use with Eigen

Pol Martín and Toni Susín
ViRVIG, Universitat Politècnica de Catalunya
UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB

Problem statement

One of the most challenging branches of computer animation is the physics simulation of the behavior of objects in a scene. In particular, deformable objects are complex and challenging to simulate since their behavior is defined by continuum mechanics. The Finite Elements Method is just one way of tackling such a problem, following Baraff & Witkin's (1998) formulation.

$$\left[\mathbf{M} - \Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right]\Delta\vec{v} = \Delta t\,\vec{f}^{(t)} + \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\vec{v}^{(t)}$$

Suppose we also add the requirement of interaction in the simulation. In that case, we need fast linear algebra implementations that entirely use the underlying hardware to reach the desired frame rate. Usually, such solvers are implemented using C++ together with some linear algebra library such as **Eigen**. Of course, these libraries are already heavily optimized, with very performant solvers; however, by analyzing the actual problem to solve we can do better than raw **Eigen**. What is worth implementing from scratch, and what should you reuse from **Eigen**?

Sparse Matrix random indexing

Our system's matrix:

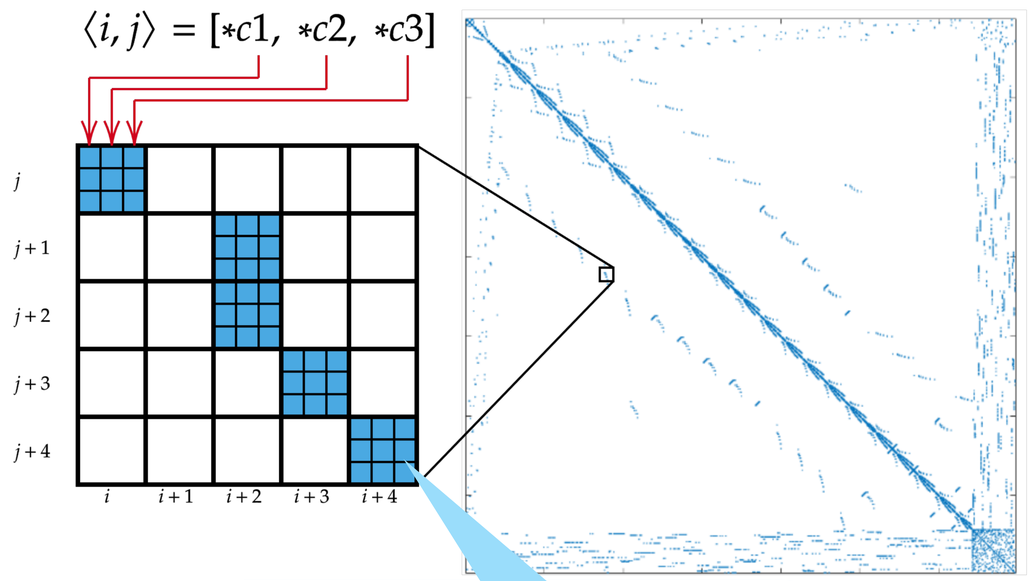
$$\left[\mathbf{M} - \Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right]$$

It is a symmetric sparse matrix, whose entries are always determined by the connectivity of the simulated mesh, if the elements are constant. Thus, this matrix is allocated only once with entries for each connected pair of nodes $< i, j >$.
The issue here is filling and cleaning the sparse matrix.

Random indexing into **Eigen**'s sparse matrices require a binary search on all column entries.

- `Scalar SparseMatrix::coeff(Index row, Index col) const`
- `Scalar& SparseMatrix::coeffRef(Index row, Index col)`

Instead, we build a table or hash table with entries for each pair of nodes connected $< i, j >$ with pointers to the first element of the 3 columns corresponding to the pair of nodes.
Eigen's Sparse Matrix compressed format guarantees consecutive elements in a column will be consecutive in memory.



We could do even better if we manage to use Sparse block matrices of 3×3 blocks. However, `Eigen::BlockSparseMatrix` is currently unsupported.

Preconditioned Conjugate Gradient

Eigen's Preconditioned Conjugate Gradient is, algorithmically, as good as any CG solver can be. However, it allocates multiple vectors on each `solve()` call, and an extra `memcpy` on `solveWithGuess()`.
A Jacobi diagonal preconditioner can also be applied faster by using our previous static indexed matrix.
Implementing a copy of **Eigen**'s CG solver, reusing memory, guarantees a noticeable speedup.

References

Baraff, D., & Witkin, A. (1998, July). Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (pp. 43-54).
Tamstorf, R., Jones, T., & McCormick, S. F. (2015). Smoothed aggregation multigrid for cloth simulation. *ACM Transactions on Graphics (TOG)*, 34(6), 1-13.

Avoiding hidden allocations in Eigen's assignment

All the heavy work in **Eigen** is done by the assign operator, which tries to optimize the assigned expression as much as possible by using something akin to lazy evaluation at compilation.
However, these expressions can have penalizations due to their implicit aliasing suppositions. Certain statements involving non-component-wise operations will dynamically allocate memory to hold a temporary result before assigning it.

These hidden allocations can be easily detected by placing breakpoints in **Eigen**'s `malloc` calls at `Core/util/Memory.h`.

To solve most of the aliasing suppositions, one must use `MatrixBase<>::noalias()` on the assigned variable when it does not appear on the right-hand-side.
Complex operations, or statements using sparse vectors, sometimes produce unnecessary copies and allocations, even without aliasing.

This example tries to contribute $\Delta t^2\mathbf{A}\vec{df} + \Delta t\mathbf{A}\vec{f}_{\text{sparse}}$ to a dense vector. It produces multiple allocations and copies.

```
rhs += dt * (A * ((dt * vec) + sparse_vec));
```

Use `MatrixBase<>::noalias()` to prevent allocations with the matrix dense vector product. However, we cannot use this with the sparse matrix-sparse vector product.

```
rhs.noalias() += dt * dt * (A * vec);
rhs += dt * (A * sparse_vec);
```

Use a temporary vector, reused across iterations, to simplify the expression. Apply only one matrix product with a dense vector.

```
tmp.noalias() = dt * vec;
tmp += sparse_vec;
rhs.noalias() += dt * (A * tmp);
```

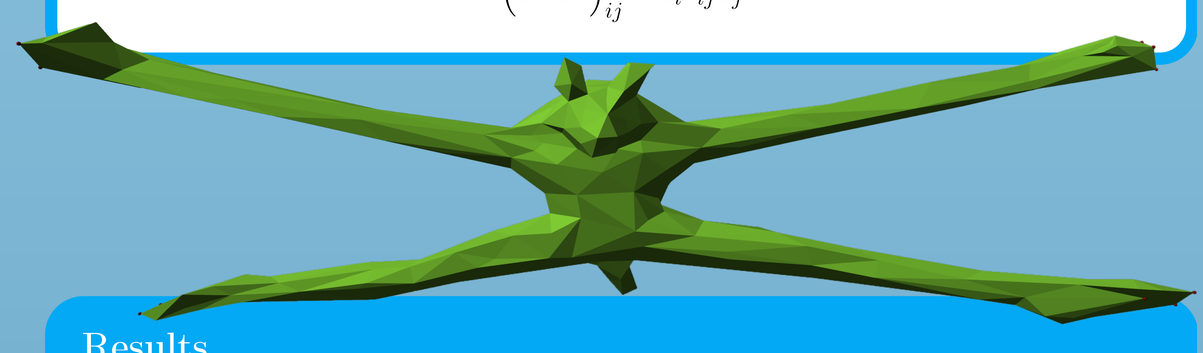
Velocity Constraints Projection

To enforce velocity constraints on a system $\mathbf{A}$, its solution is projected onto a frame that satisfies some constraints encoded in the matrix $\mathbf{S}$. The projected system is $\mathbf{SAS}^T$ (Baraff & Witkin, 1998). The sparse matrix $\mathbf{S}$ has only 3×3 block entries in its diagonal with each node's filter $\mathbf{S}_i$.

$$\mathbf{S}_i = \begin{cases} \mathbf{I} & \text{if no DoF is constrained} \\ \mathbf{I} - \hat{p}_i\hat{p}_i^T & \text{if 1 DoF } \hat{p}_i \text{ is constrained} \\ \mathbf{I} - \hat{p}_i\hat{p}_i^T - \hat{q}_i\hat{q}_i^T & \text{if 2 DoFs, } \hat{p}_i \text{ and } \hat{q}_i \text{, are constrained} \\ \mathbf{0} & \text{if all 3 DoFs are constrained} \end{cases}$$

Building matrix $\mathbf{S}$ in each step is absurdly expensive, as it requires building and compressing a sparse matrix.
We can avoid computing $\mathbf{S}$ altogether by noticing that the resulting structures of $\mathbf{SAS}^T$ will be equivalent to $\mathbf{A}$. And by just iterating the constrained nodes we can update our system sparse matrix as follows:

$$\left(\mathbf{SAS}^T\right)_{ij} = \mathbf{S}_i\mathbf{a}_{ij}\mathbf{S}_j$$

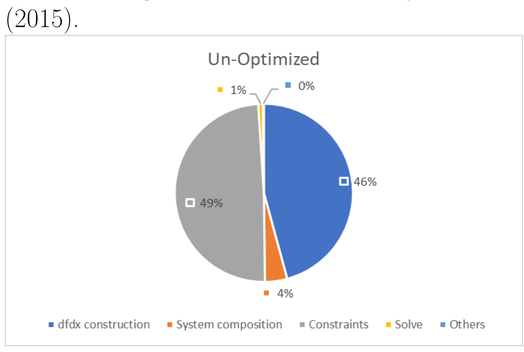


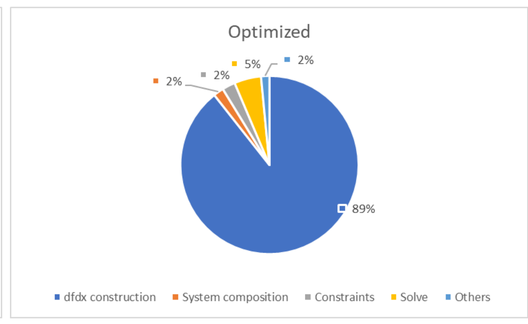
Results

In a simulation with almost 6000 elements and 2100 nodes, we get the following speedups:

$\partial\vec{f}/\partial\vec{x}$ construction	System composition	Constraints	Overall Step
x5.487	x21.358	x218.79	x9.849

The $\partial\vec{f}/\partial\vec{x}$ construction heavily uses the random indexing, the system composition deals with the statement organization to build the system, and Constraints applies the PPCG method of Tamstorf (2015).





Solve	Eigen	Eigen with guess	Ours
Time	0.0071 s	0.00439 s	0.00168 s
Speedup	x1	x1.614	x4.226