Using Sparse Matrix for the Contact Calculation

Zhan Wang

13 July 2014

Contents

1 The Formulation of G Matrix

We have the EoM:

$$0 = \ddot{g} = W^{\mathsf{T}}\dot{u} \tag{1}$$
$$M\dot{u} = h + W\lambda \ . \tag{2}$$

From 2, the acceleration $\dot{u}$ can be expressed as

$$\dot{u} = M^{-1}(h + W\lambda). \tag{3}$$

Inserting 3 into 1, we can get

$$\begin{aligned} 0 = \ddot{g} &= W^{\mathsf{T}}\dot{u} \\ &= W^{\mathsf{T}}(M^{-1}h + M^{-1}W\lambda) \\ &= W^{\mathsf{T}}M^{-1}h + (W^{\mathsf{T}}M^{-1}W)\lambda \\ &= W^{\mathsf{T}}M^{-1}h + G\lambda \end{aligned} \tag{4}$$

where G is

$$G = W^{\mathsf{T}}M^{-1}W \ . \tag{5}$$

As the mass matrix M is a real-valued symmetric positive-definite matrix, it can be decomposed by the Cholesky decomposition:

$$M = LL^{\mathsf{T}}, \tag{6}$$

where L is a lower triangular matrix with real and positive diagonal entries.

Inserting 6 in to 5, we can get

$$G = W^{\mathsf{T}}M^{-1}W = W^{\mathsf{T}}\left(LL^{\mathsf{T}}\right)^{-1}W \tag{7}$$
$$= W^{\mathsf{T}}\left(L^{\mathsf{T}}\right)^{-1}L^{-1}W \tag{8}$$
$$= W^{\mathsf{T}}\left(L^{-1}\right)^{\mathsf{T}}L^{-1}W \tag{9}$$
$$= \left[W^{\mathsf{T}}\left(L^{-1}\right)^{\mathsf{T}}\right]\left(L^{-1}W\right) \tag{10}$$
$$= \left(L^{-1}W\right)^{\mathsf{T}}\left(L^{-1}W\right) \tag{11}$$
$$= y^{\mathsf{T}}y \tag{12}$$

where

$$y = L^{-1}W \ . \tag{13}$$

Therefore, the calculation of matrix G can be performed in two steps:

- solving the triangular system $Ly = W$ represented by 13
- multiplying the solved matrix y with its transpose to get G

2 Implementation of Solving the Triangular System $Ly = W$

2.1 Should we choose iterative method or a direct method for getting L?

- LU decomposition has a complexity of $O(\frac{2}{3}n^3)$, if the matrix is positive definite, Cholesky decomposition is roughly twice as efficient as the LU decomposition for solving systems of linear equations, and more stable. And backward or forward substitution has a complexity $O(n^2)$.
- For iterative methods, the number of scalar multiplications is $0(n^2)$ at each iteration.
- If the total number of iterations required for convergence is much less than n, then iterative methods are more efficient than direct methods.
- Iterative method has a less computational cost and also demands less memory. So iterative methods are well suited for solving large sparse linear system.

As in our case, the dimension of the linear system is not so large, the L is already computed in the previous step.

2.2 Choosing the library for the forward substitution of $Ly = W$

There are a lot of open sourced libraries containing the routine of forward substitution for sparse matrices. Some of them focus on the optimization of memory access, e.g.,
Sparse Basic Linear Algebra Subprograms (BLAS) Library. Others focus on the optimization of algorithm of solving process, e.g. csparse.

CSPARSE is a C library which implements a number of direct methods for sparse linear systems, by Timothy Davis. For forward subsection, csparse applies graph theory to find out the nonzero pattern of the solution y first. And then performs the substitution only for the nonzero entries of y. Because its fully utilization of the sparsity of the linear system to avoid unnecessary calculation and memory access, this algorithm can reduce complexity of the forward substitution from $O(n^2)$ to $O(n)$.

In the book *Timothy Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006*, the author discusses this algorithm in details .

2.3 Creating the Compressed Column format sparse matrices

As the algorithms in csparse are based on the *compressed-column* format, we need to transform the matrix L and W from $fmatvec : Mat$ into compressed-column format. It is performed into two steps:

- construct a triplet format matrix C by cs_spalloc (m, n, nz, 1, 1), where m, n are the declared row and column size, nz is the number of nonzero entries, and the last input value 1 indicates the format to be triplet. (In csparse, cs_spalloc is used for the memory allocation

for both triplet and compressed column format. When the last value equals to zero, it performs allocation for compressed column format.)

- read every the blocked potential submatrix, and push the nonzero entries into a triplet format matrix using function cs_entry(C, row, column, entry). The order of pushing nonzero entries can be arbitrary.
- call cs_triplet (C) to transform the triplet format into compressed format.

In this way, nz can be arbitrary given, because cs_entry() will dynamic double the memory size by realloc() when the allocated memory is not enough. It will consider whether there is still enough space after the original memory position. If it is, it will extend the memory area of the original pointer. Otherwise it will look for a new memory position, declare new allocated memory size, and then copy the original data to this new position.

In order to avoid this unnecessary coping in every timestep, it is better to estimate a appropriate value for nz. In our case, $nz = 7 * n$ is recommended.

2.4 Solving the System using csparse

After we get the compressed L and W matrices, we can call the cs_splsolve() for every column of W, to get the y. where yele + ylda * k is the pointer to the kth column of y.ele, and xi stores

Listing 1: solving the kth column of W

```
for (int k = 0; k < W[j].cols(); k++) {
  cs_splsolve (cs_L_LLM, cs_Wj, k, xi, yele + ylda * k , 0);
```

nonzero pattern of the solution (it is not needed in our case).

3 Implementation of the matrix multiplication $G = y^\mathsf{T} y$

Similar to solving the triangular system, the implementation the matrix multiplication needs to consider both how to access the memory efficiently and how to do the operation in order to avoid the multiplication for zeros. It turns out that most of the modern processors have the ability to simplify the multiplication for zeros automatically. That is, the multiplication between zero and a nonzero is much more faster than the normal multiplication between two nonzero values.

So I focus on the optimization of how to access the memory more efficiently.

3.1 Do we need sparse matrix multiplication for $G = y^\mathsf{T} y$

The Matlab test shows that with full matrix format, the multiplication is about ten times faster than a direct multiplication between two compressed column format sparse matrices. That is

because when y is stored in compressed-column format, it is time consuming to access the every column of y^{T}.

Then I considered whether it is possible to use the multiplication routine provided by csparse. The multiplication routine from csparse is designed for the multiplication of two arbitrary sparse matrices. It spends some time to check the sparsity pattern before doing the real calculation.

In our case, which turns out to be a very special multiplication, because the the resulting matrix G will be a symmetric matrix. Taking this into consideration, the computational cost can be reduced in half immediately. And also, the two multiplied matrices are transposed matrices to each other. So the entry of $G(i, j)$ can be calculated by

$$G(i, j) = y^{\mathsf{T}}(i, :)y(:, j) = y(:, i)^{\mathsf{T}}y(:, j) \tag{14}$$

Both $y(:, i)$ and $y(:, j)$ are column vectors of matrix y, which can be accessed very efficiently.

The algorithm of calculating G is then described by algorithm 2.

Listing 2: Algorithm of calculating G

```
for (int i = 0; i < y.cols(); i++) {
  Vec temp = y.col(i); // temp can be kept in the fast memory (cache)
  for (int j = i; j < y.cols(); j++) {
    double val = scalarProduct(temp, y.col(j));
    G(i, j) = val;
    G(j, i) = val;
  }
}
```

In the inner iteration, the ith column will be accessed repeatedly. Vec temp = y.col(i); is written explicitly so that it may be easier for the complier to detect this feature and to keep the ith column in the fast memory (cache).

Currently G is declared as a general matrix type. If G can be declared to be fmatvec:Symmetric, then only the elements in the lower triangular will be stored (e.g. in column wise). In that case, G(j, i) = val will not be needed in algorithm 2. And the order of calculating entries of G, described in algorithm 2, is suitable for the column wise stored G matrix, because it access the entries of G in the order how it is stored in memory. In our case, I test both row wise and column wise calculating order, the performance does not show a big difference.