# Linear Solver Optimization in A Nonlinear Finite Element Application

Tongge Wu

*Department of Mechanical Engineering, University of California, Berkeley*
May 11th, 2020

## 1 Introduction

Finite Element Method (FEM), as a numerical method to solve partial differential equation (PDE), is widely used in numerical analysis and engineering analysis (e.g. mechanical, civil, and material engineering field). Starting with Galerkin method, FE method discretizes the problem domain into *elements*, each element has multiple *nodes* (e.g. triangle or quadrilateral element). Based on the properties of the PDE, FE method then constructs local matrix (e.g. mass matrix, stiffness matrix) for each element. After this, all local matrices are *assembled* into a global matrix using the connectivity between element (i.e. the way elements share nodes). The key step (math core) of FE method boiled into solving a linear system or a group of linear systems. Moreover, if the property of PDE involves nonlinearity, then the math core of FE method becomes solving a nonlinear system or a group of nonlinear system.

Within the scope of this project, we aimed to investigate the performance of various linear solvers (direct and iterative) applied in a mechanical FE simulation, where a nonlinear material (rubber) is under large-deformation (i.e. nonlinear), and the mechanical simulation was transient (i.e. time-dependent).

Following this introduction, in Section 2 we will elaborate the numerical framework and general code scheme of the FE simulation. Section 3 lists various choices of linear solvers and corresponding accelerating techniques, along with the improvements/degeneration in performance. Section 4 shows our conclusion.

## 2 Math Framework and Code Scheme

In this project, we used FE method to simulate the deformation of a two-dimensional solid rubber-like rectangular block ($X \times Y = 2m \times 10m$). The bottom surface of the block was fixed on the flat ground, and the top surface of the block was pulled into a prescribed displacement ($u_Y = 1m$). The block was discretized into a regular mesh consisting of $EX \times EY$ elements. Each linear quadrilateral element has 4 nodes, resulted in $NX \times NY = (EX+1) \times (EY+1)$ nodes totally. Since the bottom nodes were boundary conditions (fixed in both directions), and the top nodes were boundary condition in the pulling stage, the total degree-of-freedom (DOF) was roughly $NX \times NY \times 2$.

Briefly, the FE method resulted in a group of nonlinear ODEs for this particular simulation,

$$[\boldsymbol{M}][\hat{\boldsymbol{a}}_{n+1}] + [\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})] = \boldsymbol{0}. \tag{1}$$

where the $\boldsymbol{M}$ was the mass matrix, it was sparse, symmetric and positive definite. $\hat{\boldsymbol{a}}_{n+1}$ was the acceleration of each DOF. $\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})$ was the stress divergence of each DOF, which was a nonlinear function of displacement ($\hat{\boldsymbol{u}}_{n+1}$) of each DOF. Eq.1 was essentially Newton's second law applied at a continuum level. Newmark integrator was then used to convert the group of nonlinear ODEs (Eq.1) into a group of nonlinear difference equations, plus some update operations (not shown in the text),

$$\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}) = \frac{1}{\beta \Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1} + \boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1}) - \boldsymbol{M}[(\hat{\boldsymbol{u}}_n + \hat{\boldsymbol{v}}_n \Delta t)\frac{1}{\beta \Delta t^2} + \frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n] = \boldsymbol{0}. \tag{2}$$

where $\beta$ was the Newmark parameter. $\Delta t$ was the time step. $\hat{\boldsymbol{v}}_n$ was the velocity of each DOF. Subscripts represented solution at different time steps - old ($n$) versus new ($n+1$). On a deeper level, we used Newton-Raphson nonlinear solver to solve the group of nonlinear algebraic equation (Eq. 2), which required us to form a tangent matrix (gradient) in each Newton-Raphson iteration,

$$\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)} = D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)}) \backslash (-\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})). \tag{3}$$

where $\Delta\hat{\boldsymbol{u}}_{n+1}^{(k)}$ was the update part of the solution. $D\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1}^{(k)})$ was the tangent matrix, which was sparse, symmetric and positive definite. Backslash represented any linear solver.

All intermediate math and other FE method details were in section 5 (Appendix), including the discretization, updating operations of Newmark integrator, and forming the tangent matrix, etc.

**Algorithm 1:** Code Scheme

```
Construct mesh - node, element, boundary condition, material constant;
Construct Mass matrix;
% Pull Stage;
while time < $T$ do
    Prescribe boundary condition;
    %call Newmark-integrator function;
    Save old $u, v, a$;
    Form residual based on initial guess $u$;
    %call Newton-Rasphson nonlinear solver;
    while norm(residual) > tol AND iter < maxiter do
        Form tangent matrix (S.P.D. SpM);
        A few SpMSpM, SpMV operations;
        $\delta u$ = linsolve(tangent matrix, residual);
        $u = u + \delta u$;
        Form residual based on $u$;
    end
    %Newton-Rasphson solver converges;
    $v, a$ update;
end
```

To code such numerical framework, following scheme was developed (see Algo.1). The initialization part included setting the mesh, node coordinates, the connectivity array and the boundary condition. This part was designed to be generic such that it could also accept other FE input files. Since the mass matrix was a constant matrix, it was created only once in the beginning of the program.

Then, we called the Newmark integrator for time stepping. Before the while loop of Newton-Rasphson iteration, we guessed a solution and formed a initial residual (RHS of Eq.2). Within the while loop of Newton-Rasphson solver, the solver would formed the tangent matrix and **linsolve** for a solution update (RHS and LHS of Eq.3).

In the design space of the linear solver (showed in Algo.1 in blue), since the tangent matrix was a sparse, symmetric, and positive definite matrix, we could use direct solver (such as Cholesky decomposition), iterative solver (such as Conjugate Gradient, CG), and iterative solver with various choices of preconditioner. To ensure the consistency in measurement of the performance of different solvers, we used both direct and iterative solvers provided in the Intel Math Kernel Library (MKL). In terms of the scale of the problem, the dimension of $Ax = b$ linear system (number of rows of $A$) was $NX \times NY \times 2 - NX \times 3$, where $NX \times 3$ DOFs were taken as boundary condition.

If the Newton-Raphson solver converged, hence the inner while loop broke, the Newmark integrator would update related terms and step forward.

## 3 Linear Solver Optimizations

### 3.1 Direct Sparse Solver

To solve the linear system in our FE simulation, the most convenient way was to call the Direct Sparse Solver (DSS) Interface in the MKL. Using DSS interface involved calling following functions in order:

- `dss_create` (initializes basic data structures necessary for the solver)
- `dss_define_structure` (informs the solver the structure of the matrix)
- `dss_reorder` (permutation of the matrix to reduce fill-in in the factoring stage)
- `dss_factor` (computes Cholesky decomposition $A = LL^\top$)
- `dss_solve` (backward and forward substitution)
- `dss_delete` (free memory)

To measure the performance of our implementations, we used a test FE simulation case, where we have 16,000 DOFs, and the Newmark integrator iterated for 30 time steps. With a naive DSS implementation, the FE simulation finished in 50.81 seconds, as shown in Table 1.

| DOF | time steps | solver | modification | Wall-time (s) |
|---|---|---|---|---|
| 16,000 | 30 | DSS | - | 50.81 |
| 16,000 | 30 | DSS | Parallel Nested Dissection Reordering | 35.97 |
| 16,000 | 30 | DSS | Reuse Permutation | 35.70 |
| 16,000 | 30 | PARDISO | - | 48.60 |
| 16,000 | 30 | PARDISO | Parallel Reordering and Substitution | 37.66 |
| 16,000 | 30 | PARDISO | Reuse Permutation | 26.24 |

Table 1: Direct solver performance on one Cori KNL node.

The first optimization on DSS was about the reordering stage. In the reordering stage, if we used parallel nested dissections algorithm (`MKL_DSS_METIS_OPENMP_ORDER` option), we had a 29.21% less computing time compared to the naive DSS implementation (Table 1). The second optimization was more relate to our Newton-Raphson solver design. According to Algo. 1, in the Newton-Raphson solver, the linear solver (DSS here) was called multiple times for the same sparse pattern tangent matrix. Since the nonzero entries in the tangent matrix depended on the connectivity array of the mesh, which was not changed in the whole period of program running. If the ordering information could be kept from the first DSS call, all later DSS calls could use the permutation information directly. Luckily, `dss_reorder` had the features to keep the permeation in an assigned array (`MKL_DSS_GET_ORDER`) and to accept user input permutation (`MKL_DSS_MY_ORDER`). Such optimization rendered another 1% less computing time compare to the parallel reordering DSS case (Table 1).

The reasons why reusing the permutation did not "significantly" improve the performance was following. If we used the parallel nested dissections algorithm for all tangent matrices, each `dss_reorder` call cost about 0.06 seconds. When the permutation array was kept in an external array, the parallel nested dissections algorithm was not allowed. Then, the regular `dss_reorder` call cost about 0.2 seconds. Even later `dss_reorder` call with user input permutation cost only 0.02 seconds, we could not get much benefits if the number of Newton-Raphson iteration was not large (about 6-7 in the test FE simulation). However, if the Newton-Raphson iteration number was large, the reusing permutation should result in an much better improvement.

## 3.2 Parallel Direct Sparse Solver

While DSS was an interface provided in MKL, all DSS calls internally used Parallel Direct Sparse Solver (PARDISO) with difference phases [2, 4, 3]. PARDISO exposed more options and parameters to user if called directly, thus could have better performance if user had more information about the matrix. However, the price of flexibility was inconvenience. To use PARDISO as our linear solver, we would call following phases in order:

- user configuration (initializes options and parameters)
- `phase 11` (reordering and symbolic factorization)
- `phase 22` (numerical factorization)
- `phase 33` (backward and forward substitution)
- `phase -1` (free memory)

Using the same test case as in section 3.1, we saw a naive PARDISO implementation was already better than the naive DSS implementation by 4.35%, as the data shown in Table 1.

The first optimization in PARDISO was tuning `iparm` array that controlled all aspects of behaviors of PARDISO. Similar to the DSS reordering stage, we could use parallel version of the nested dissection algorithm in the PARDISO reordering stage (`iparm[1]`). More than this, we could also use parallel version of the backward and forward substitution in PARDISO (`iparm[24]`). Meanwhile, we turned off all statistic information generation in PARDISO to save computing time (`iparm[17,18,19]`). Those tuning improved the performance of PARDISO by 22.51% compared to the naive PARDISO implementation (Table 1). Inspired by the reusing permutation in the DSS interface, we also explored the reusing permutation in PARDISO, though it was implicit. PARDISO saved the permutation somewhere hidden to the user. However, if we did not call the free memory phase (`phase -1`) after the first PARDISO call, this internal information was kept in PARDISO handle and could used for the numerical factorization phase (`phase 22`) in next Newton-Raphson iteration. Unlike the DSS interface, reusing permutation in PARDISO "significantly" reduced our computation time by 30.32% compared to the parallel reordering and solving PARDISO (Table 1).

## 3.3 Iterative Sparse Solver

Contrast to the direct linear solver like DSS ans PARDISO, iterative solver like conjugate gradient (CG) could be used when the FE simulation had a relatively large scale. For example, when we had 6,400,000 DOFs in our FE simulation, using direct method would require more than 96GB memory (DRAM) to save the internal numerical factorization and other variables, which was larger than the physical memory on one Cori KNL node (out-of-memory error). Even the PARDISO provided some out-of-core execution (swap memory with disk), from lecture we learned that disk latency was much larger than the memory latency, which was already much larger than the cache latency [5]. Hence, using direct linear solver would not be optimal in this situation. By contrast, Iterative Sparse Solver (ISS) interface from MKL was a better choice for the FE simulation.

**Algorithm 2:** CG from MKL Scheme

```
User input data and options;
dcg_init initialize the solver handle;
dcg_check check the correctness of input data and options;
dcg compute the some vectors, assign a request;
if request = 0 then
    go to success
end
if request = 1 then
    do SpMV;
    go to dcg;
end
if request = 2 then
    do norm of residual;
    go to dcg;
end
if request = 3 then
    do preconditioner inversion with some vectors;
    go to dcg;
end
if request = other then
    go to failure;
end
success: dcg_get assign solution and statistic during CG, return 0;
failure: report detailed error, return 0;
```

In general, ISS from MKL provided three choices: CG for single RHS, CG for multiple RHSs, General Minimal Residual (GMRES) for single RHS. Considering the tangent matrix was symmetric and positive definite, we used the CG for single RHS from MKL, which required a code scheme as Algo.2. In the Algo. 2, MKL functions provided only the framework of the actual CG method. The sparse matrix-vector multiplication (SpMV), vetor norm, linear solver associated with the preconditioner was fully provide by the user, and those calls depended on the request returned by the `dcg` function. Thus, such CG method was called reverse communication interface conjugate gradient (RCI-CG) in ISS from MKL.

To measure and compare the performance of iterative solvers, we constructed a larger test FE simulation than the one in section 3.1. We had a 100,000 DOFs FE simulation where the Newmark integrator iterated for 20 time steps. With a naive CG implementation, we set the convergence criterion to be absolute residual less than 1E-6, or relative residual (relative to the initial residual) less than 1E-11. And the maximum number of iteration was 5,000. With such configuration, the FE simulation finished in 644.95 seconds, as shown in Table 2. Meanwhile, a DSS solver (with parallel reordering and reusing permutation array) applied on the same test case cost only 290.59 seconds.

Aimed to improve the performance of the CG method, we investigated two types of preconditioner: Jacobi and Incomplete Cholesky (IC). Jacobi preconditioner was simply the diagonal of tangent matrix. Since it was a diagonal matrix $D$, the "precondition inversion" step cost was cheap, and hopefully $D^{-1}A$ had a lower condition number. Using the Jacobi preconditioner, we observed an increased number of iteration of CG in the first a few Newton-Raphson iteration (1-5), and decreased number of iteration of CG in later Newton-Raphson iteration (6-9), as shown in Fig.1 (a). However, the total computation cost was 33.43% more than the naive CG implementation (Table 2). IC preconditioner was the Cholesky decomposition of tangent matrix, but no new

| DOF | time steps | solver | modification | Wall-time (s) |
|---|---|---|---|---|
| 100,000 | 20 | CG | - | 644.95 |
| 100,000 | 20 | CG | Jacobi Preconditioner | 860.47 |
| 100,000 | 20 | CG | Incomplete Cholesky Preconditioner | 1293.64 |
| 100,000 | 20 | Homemade CG | - | 957.79 |
| 100,000 | 20 | Homemade CG | Incomplete Cholesky Preconditioner | 2077.12 |

Table 2: Iterative solver performance on one Cori KNL node.

fill-in was allowed in the decomposition process, ideally we would have

$$A \approx L_1 L_1^\top = M$$

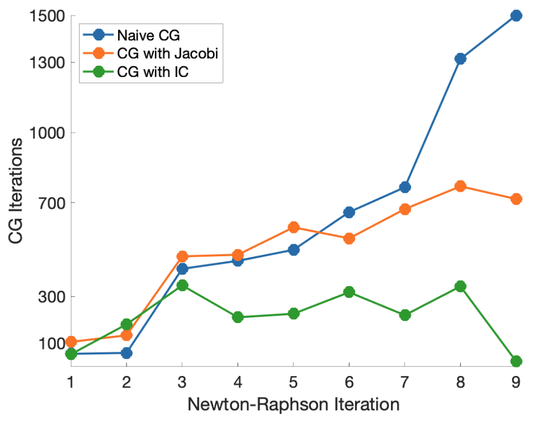
Since there was no IC function provided in MKL, we had to use incomplete sparse LU (ILU) decomposition (`dsilu0` function), where we would have

$$A \approx L_2 U_2$$

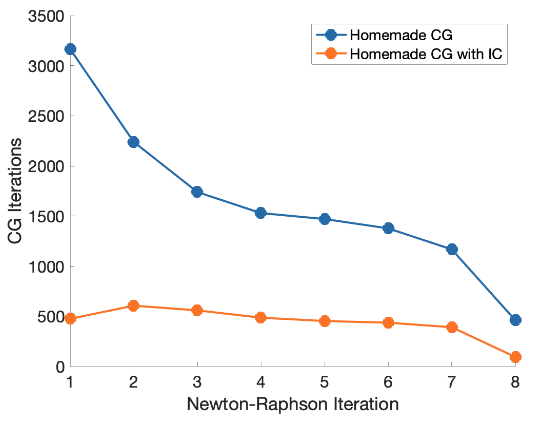
here, $L_2$ was an unit-diagonal lower triangular matrix, $U_2$ was an non-unit-diagonal upper triangular matrix. To get the $L_1$ matrix from $L_2$, we used

$$L_2 U_2 = L_2 D_2 U_2^* = (L_2 D_2^{1/2})(D_2^{1/2} U_2^*) = L_1 L_1^\top$$

where $D_2$ was the diagonal of $U_2$. Since tangent matrix was symmetric, we claimed the $L_2 D^{1/2} = L_1$ and $D^{1/2} U^* = L_1^\top$. Then, in the preconditioner inversion step, we used a triangular solver (`mkl_sparse_d_trsv`) twice to apply the preconditioning. Using the IC preconditioner, we also observed decreased number of iteration of CG during most Newton-Raphson iterations, as shown in Fig.1 (a). However, the total computation cost was even 50.43% more than the CG with Jacobi preconditioner. All actual wall-time data was in Table 2.



(a) MKL CG and MKL CG with preconditioner



(b) Homemade CG and Homemade CG with preconditioner

Figure 1: Number of CG iteration requires at each Newton-Raphson iteration

## 3.4 Homemade Conjugate Gradient

Using RCI-CG from MKL with Jacobi preconditioner or IC preconditioner resulted in a lower number of iteration in the CG method, but a longer computation time. To explain such discrepancy, we used a homemade version conjugate gradient (CG) adopted from textbook Algorithm 6.12. Since the RCI-CG from MKL involved multiple convergence criterion and was not user-friendly in statistic reporting, our homemade CG was used to study the computation time at each CG stages.

With the very same test FE simulation in section 3.3, we used zero vector as our initial guess in CG method, the convergence criterion was $\|r_k\|_2$ less than 1E-6. Using the naive CG implementation, the FE simulation finished in 957.79 seconds, while using the CG with IC preconditioner, the FE simulation finished in 2077.12 seconds, 116% more than the naive CG implementation. All data was shown in Table 2. We still observed that CG with IC preconditioner required much less number of iteration (avg: 438) than the naive CG implementation (avg: 1644), as shown in Fig.1. This was excepted since the condition number of $M^{-1}A$ was less than $A$ itself, especially when $M = LL^{-1} \approx A$.

Using hpctoolkit installed on Cori, we could investigate time decomposition of each implementation [1]. If we used the computation time of norm of residual as the baseline - 1 unit, SpMV in naive CG cost 11.11 units, while SpMV in ICCG cost 33.13 units. This was surprising since the number of norm calling and the number of SpMV calling were the same in either implementation, and those two operations were not affected by using preconditioner or not. Thus, we should have the same SpMV cost in both implementations. The triangular solving in naive CG (identical matrix as preconditioner) cost 3.46 units, while the triangular solving in ICCG cost 16.45 units. This partially explained why the CG with IC preconditioner resulted in a longer computation time in the FE simulation. Further investigation should be done for detailed understanding.

## 4 Conclusion

This project focused on a nonlinear FE simulation, and explored various sparse linear solvers used in the Newton-Raphson solver with in the FE simulation. In terms of direct solver, we found that using PARDISO directly was more efficient than using DSS interface from MKL. Parallel reordering and parallel solving technique, as well as reusing permutation technique, could significantly improved the performance of PARDISO. When physical memory was sufficient, using directly sparse solver was superior to iterative sparse solver in our FE simulation. In terms of iterative solver, using incomplete Cholesky preconditioner could significantly decrease the number of iteration in conjugate gradient method, however, such preconditioning may not achieve a better computation time cost in our FE simulation. In conclusion, with a small scale nonlinear FE simulation using Newton-Raphson solver, PARDISO with optimizations was recommended; with a large scale case, simple conjugate gradient method in ISS interface from MKL was recommended.

## References

[1] L. Adhianto et al. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701. DOI: `10.1002/cpe.1553`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553`.

[2] Arne De Coninck et al. "Needles: Toward Large-Scale Genomic Prediction with Marker-by-Environment Interaction". In: 203.1 (2016), pp. 543–555. ISSN: 0016-6731. DOI: `10.1534/genetics.115.179887`. eprint: `http://www.genetics.org/content/203/1/543.full.pdf`. URL: `http://dx.doi.org/10.1534/genetics.115.179887`.

[3] D. Kourounis, A. Fuchs, and O. Schenk. "Towards the Next Generation of Multiperiod Optimal Power Flow Solvers". In: *IEEE Transactions on Power Systems* PP.99 (2018), pp. 1–10. ISSN: 0885-8950. DOI: `10.1109/TPWRS.2017.2789187`. URL: `https://doi.org/10.1109/TPWRS.2017.2789187`.

[4] Fabio Verbosio et al. "Enhancing the scalability of selected inversion factorization algorithms in genomic prediction". In: *Journal of Computational Science* 22.Supplement C (2017), pp. 99–108. ISSN: 1877-7503. URL: `https://doi.org/10.1016/j.jocs.2017.08.013`.

[5] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. `http://www.tacc.utexas.edu/~eijkhout/istc/istc.html`. lulu.com, 2011. ISBN: 978-1-257-99254-6.

## 5 Appendix

### 5.1 FE Formulation

Used the total Lagrangian finite element method, we need to solve the linear momentum balance equation at time step $t_{n+1}$

$$\int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{a}_{n+1} dV + \int_{R_0} \frac{\partial \boldsymbol{\xi}_{n+1}}{\partial \boldsymbol{X}} \cdot \boldsymbol{P}_{n+1} dV = \int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{b}_{n+1} dV + \int_{\Gamma_{q0}} \boldsymbol{\xi}_{n+1} \cdot \bar{\boldsymbol{p}}_{n+1} dA. \tag{4}$$

When there was no body force and prescribed traction, Eq.4 could be simplified

$$\int_{R_0} \boldsymbol{\xi}_{n+1} \cdot \rho_0 \boldsymbol{a}_{n+1} dV + \int_{R_0} \frac{\partial \boldsymbol{\xi}_{n+1}}{\partial \boldsymbol{X}} \cdot \boldsymbol{P}_{n+1} dV = 0. \tag{5}$$

The first term was the acceleration, and the second term was stress divergence.

Using discretization, within each 4-node quadrilateral element, we could have

$$\begin{aligned}
\boldsymbol{\xi}^e_{n+1} &= [\boldsymbol{N}^e][\hat{\boldsymbol{\xi}}^e_{n+1}] \\
\boldsymbol{a}^e_{n+1} &= [\boldsymbol{N}^e][\hat{\boldsymbol{a}}^e_{n+1}] \\
\langle \frac{\partial \boldsymbol{\xi}^e_{n+1}}{\partial \boldsymbol{X}} \rangle &= [\boldsymbol{B}^e][\hat{\boldsymbol{\xi}}^e_{n+1}] \\
\langle \boldsymbol{P}_{n+1} \rangle &= \begin{bmatrix} P_{11} \\ P_{21} \\ P_{12} \\ P_{22} \end{bmatrix}_{n+1}.
\end{aligned}$$

Where $[\boldsymbol{N}^e]$ was a 2 by 8 array of interpolation function (shape function), $[\hat{\boldsymbol{\xi}}^e_{n+1}]$ was a 8 by 1 array, standed for the tangent vector, $[\boldsymbol{N}^e][\hat{\boldsymbol{\xi}}^e_{n+1}]$ together would yield a 2 by 1 tangent vector. $[\hat{\boldsymbol{a}}^e_{n+1}]$ was a 8 by 1 array, standed for the acceleration vector, $[\boldsymbol{N}^e][\hat{\boldsymbol{a}}^e_{n+1}]$ together would yield a 2 by 1 acceleration vector. $\langle \frac{\partial \boldsymbol{\xi}^e_{n+1}}{\partial \boldsymbol{X}} \rangle$ was a 4 by 1 array to represent tangent vector gradient, $[\boldsymbol{B}^e]$ was a 4 by 8 array of interpolation function gradient. $\langle \boldsymbol{P}_{n+1} \rangle$ was a 4 by 1 array of reshaped first Piola-Kirchhoff stress.

Plug the discretization into the Eq.5, we found a matrix form

$$[\boldsymbol{\xi}^e_{n+1}]^T \left( \int_{\Omega_0} [\boldsymbol{N}^e]^T \rho_0 [\boldsymbol{N}^e] dV \right) [\hat{\boldsymbol{a}}^e_{n+1}] + [\boldsymbol{\xi}^e_{n+1}]^T \int_{\Omega_0} [\boldsymbol{B}^e]^T \langle \boldsymbol{P}_{n+1} \rangle dV = 0.$$

Since the tangent vector was arbitrary, assembling process led to

$$[\boldsymbol{M}][\hat{\boldsymbol{a}}_{n+1}] + [\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})] = \boldsymbol{0}. \tag{6}$$

Where $[\boldsymbol{M}]$ was the global mass matrix, $[\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})]$ was the global stress divergence term as a function of displacement. This was a **group of nonlinear ordinary differential equations** to be solved.

### 5.2 Time Stepping

Eq.6 was a group of nonlinear ordinary differential equations w.r.t. time. To solve the group of ODEs, we used Newmark method with Newmark parameters $\beta = 1/2, \gamma = 1/4$. Given the parameters, and the known solution at time step $t_n$, solution at $t_{n+1}$ could be calculated by

$$\frac{1}{\beta\Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1}+\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})=\boldsymbol{M}[(\hat{\boldsymbol{u}}_n+\hat{\boldsymbol{v}}_n\Delta t)\frac{1}{\beta\Delta t^2}+\frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n] \tag{7}$$

$$\hat{\boldsymbol{a}}_{n+1}=\frac{1}{\beta\Delta t^2}(\hat{\boldsymbol{u}}_{n+1}-\hat{\boldsymbol{u}}_n-\hat{\boldsymbol{v}}_n\Delta t)-\frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n \tag{8}$$

$$\hat{\boldsymbol{v}}_{n+1}=\hat{\boldsymbol{v}}_n+[(1-\gamma)\hat{\boldsymbol{a}}_n+\gamma\hat{\boldsymbol{a}}_{n+1}]\Delta t. \tag{9}$$

Where $\Delta t$ was the time step, and was assumed to be constant. Hence, it was not difficult to see that Eq.7 was the key equation to be solved to progress from $t_n$ to $t_{n+1}$.

## 5.3 Nonlinear solver

Eq.7 derived from last section was a **group of nonlinear algebraic equations**, one could used Newton-Raphson method to solve it. To render more formal Newton-Raphson method, Eq.7 could be written as

$$\boldsymbol{f}(\hat{\boldsymbol{u}}_{n+1})=\frac{1}{\beta\Delta t^2}\boldsymbol{M}\hat{\boldsymbol{u}}_{n+1}+\boldsymbol{R}(\hat{\boldsymbol{u}}_{n+1})-\boldsymbol{M}[(\hat{\boldsymbol{u}}_n+\hat{\boldsymbol{v}}_n\Delta t)\frac{1}{\beta\Delta t^2}+\frac{1-2\beta}{2\beta}\hat{\boldsymbol{a}}_n]=\boldsymbol{0}. \tag{10}$$

Then, in each iteration, the Newton-Raphson solver yield

$$\Delta\hat{\boldsymbol{u}}^{(k)}_{n+1}=D\boldsymbol{f}(\hat{\boldsymbol{u}}^{(k)}_{n+1})\backslash(-\boldsymbol{f}(\hat{\boldsymbol{u}}^{(k)}_{n+1})). \tag{11}$$

Where $\Delta\hat{\boldsymbol{u}}^{(k)}_{n+1}$ was the increment of displacement vector from $k$ iteration to $k+1$ iteration, $D\boldsymbol{f}(\hat{\boldsymbol{u}}^{(k)}_{n+1})$ was the "tangent" matrix, and $\boldsymbol{f}(\hat{\boldsymbol{u}}^{(k)}_{n+1})$ was the residual at iteration $k$. One of the most computationally heavy task was to form the **tangent matrix**.

## 5.4 How to get the Tangent matrix

In the Eq.11, we formed the tangent matrix analytically. The first term in Eq.10 generated part of the tangent matrix, element-wisely

$$\boldsymbol{K}^{e(k)}_{{n+1}_{\text{inertial}}}=\frac{1}{\beta\Delta t^2}\boldsymbol{M}^e=\frac{1}{\beta\Delta t^2}(\int_{\Omega_0}[\boldsymbol{N}^e]^T\rho_0[\boldsymbol{N}^e]dV). \tag{12}$$

This was the inertial stiffness matrix.

The second term in Eq.10 was from, element-wisely

$$\boldsymbol{R}^e(\hat{\boldsymbol{u}}_{n+1})=\int_{\Omega_0}[\boldsymbol{B}^e]^T\langle\boldsymbol{P}_{n+1}\rangle dV.$$

Further back, this term was from

$$[\boldsymbol{\xi}^e_{n+1}]^T\int_{\Omega_0}[\boldsymbol{B}^e]^T\langle\boldsymbol{P}_{n+1}\rangle dV=\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e_{n+1}}{\partial\boldsymbol{X}}\cdot\boldsymbol{P}_{n+1}dV. \tag{13}$$

To get another part of tangent matrix, we had to differentiate RHS of Eq.13

$$\begin{aligned}D[\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{P}dV](\boldsymbol{u},\Delta\boldsymbol{u})&=D[\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}\boldsymbol{S}dV](\boldsymbol{u},\Delta\boldsymbol{u})\\&=\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot D\boldsymbol{F}(\boldsymbol{u},\Delta\boldsymbol{u})\boldsymbol{S}dV+\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}D\boldsymbol{S}(\boldsymbol{u},\Delta\boldsymbol{u})dV.\end{aligned} \tag{14}$$

The first term in RHS of Eq.14 was

$$\begin{aligned}\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot D\boldsymbol{F}(\boldsymbol{u},\Delta\boldsymbol{u})\boldsymbol{S}dV &= \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{S}dV\\ &= \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\boldsymbol{S}^T\cdot\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}dV\\ &= [\boldsymbol{\xi}^e]^T\int_{\Omega_0}[\boldsymbol{B}^e]^T[\boldsymbol{S}_{mod}][\boldsymbol{B}^e][\Delta\boldsymbol{u}^e]dV.\end{aligned}$$

Where

$$[\boldsymbol{S}_{mod}] = \begin{bmatrix}S_{11}\boldsymbol{I}_2 & S_{21}\boldsymbol{I}_2\\ S_{12}\boldsymbol{I}_2 & S_{22}\boldsymbol{I}_2\end{bmatrix}.$$

Where $\boldsymbol{I}_2$ was 2 by 2 identity matrix. Thus

$$\boldsymbol{K}^{e(k)}_{n+1_{\text{geo}}} = \int_{\Omega_0}[\boldsymbol{B}^e]^T[\boldsymbol{S}_{mod}][\boldsymbol{B}^e]dV \tag{15}$$

was the geometric stiffness matrix.

The second term in RHS of Eq.14 could be expanded as

$$\begin{aligned}\int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}D\boldsymbol{S}(\boldsymbol{u},\Delta\boldsymbol{u})dV &= \int_{\Omega_0}\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}\cdot\boldsymbol{F}\frac{\partial\boldsymbol{S}}{\partial\boldsymbol{E}}\Delta\boldsymbol{E}dV\\ &= \int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot\frac{\partial\boldsymbol{S}}{\partial\boldsymbol{E}}\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T)dV.\end{aligned} \tag{16}$$

Incorporating the constitutive law

$$\boldsymbol{S} = 2\mu\boldsymbol{E}+\lambda tr(\boldsymbol{E})\boldsymbol{I}_2.$$

Where $\boldsymbol{I}_2$ was a second-order identity tensor. And

$$\frac{\partial\boldsymbol{S}}{\partial\boldsymbol{E}} = 2\mu\boldsymbol{I}_4+\lambda tr()\boldsymbol{I}_2. \tag{17}$$

Where $\boldsymbol{I}_4$ was a fourth-order identity tensor, $tr()$ was a trace operator on any second-order tensor.

Plug Eq.17 into Eq.16, we could have

$$\begin{aligned}&\int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot\frac{\partial\boldsymbol{S}}{\partial\boldsymbol{E}}\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T)dV\\ &=\int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot 2\mu\boldsymbol{I}_4\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T)dV\\ &+\int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot\lambda tr(\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T))\boldsymbol{I}_2dV\\ &=2\mu\int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T)dV+\lambda\int_{\Omega_0}tr(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}})tr(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F})\boldsymbol{I}_2dV.\end{aligned} \tag{18}$$

Both terms in RHS of Eq.18 could be written in matrix forms as

$$2\mu\int_{\Omega_0}\frac{1}{2}(\boldsymbol{F}^T\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}+\frac{\partial\boldsymbol{\xi}^e}{\partial\boldsymbol{X}}^T\boldsymbol{F})\cdot\frac{1}{2}(\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}^T\boldsymbol{F}+\frac{\partial\Delta\boldsymbol{u}}{\partial\boldsymbol{X}}\boldsymbol{F}^T)dV = [\boldsymbol{\xi}^e]^T\int_{\Omega_0}2\mu[\boldsymbol{B}^e]^T[\boldsymbol{F}_{mod1}]^T[\boldsymbol{F}_{mod1}][\boldsymbol{B}^e][\Delta\boldsymbol{u}^e]dV.$$

Where

$$[\boldsymbol{F}_{mod1}] = \begin{bmatrix} F_{11} & F_{21} & 0 & 0 \\ \frac{1}{2}F_{12} & \frac{1}{2}F_{22} & \frac{1}{2}F_{11} & \frac{1}{2}F_{21} \\ \frac{1}{2}F_{12} & \frac{1}{2}F_{22} & \frac{1}{2}F_{11} & \frac{1}{2}F_{21} \\ 0 & 0 & F_{12} & F_{22} \end{bmatrix}.$$

And

$$\lambda \int_{\Omega_0} tr(\boldsymbol{F}^T \frac{\partial \boldsymbol{\xi}^e}{\partial \boldsymbol{X}}) tr(\frac{\partial \Delta \boldsymbol{u}}{\partial \boldsymbol{X}}^T \boldsymbol{F}) \boldsymbol{I}_2 dV = [\boldsymbol{\xi}^e]^T \int_{\Omega_0} \lambda [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod2}]^T [\boldsymbol{F}_{mod2}][\boldsymbol{B}^e][\Delta \boldsymbol{u}^e] dV.$$

Where

$$[\boldsymbol{F}_{mod2}] = \begin{bmatrix} F_{11} & F_{21} & F_{12} & F_{22} \end{bmatrix}.$$

Thus,

$$\boldsymbol{K}^{e(k)}_{n+1_{\text{mat}}} = \int_{\Omega_0} 2\mu [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod1}]^T [\boldsymbol{F}_{mod1}][\boldsymbol{B}^e] dV + \int_{\Omega_0} \lambda [\boldsymbol{B}^e]^T [\boldsymbol{F}_{mod2}]^T [\boldsymbol{F}_{mod2}][\boldsymbol{B}^e] dV \qquad (19)$$

was the material stiffness matrix.

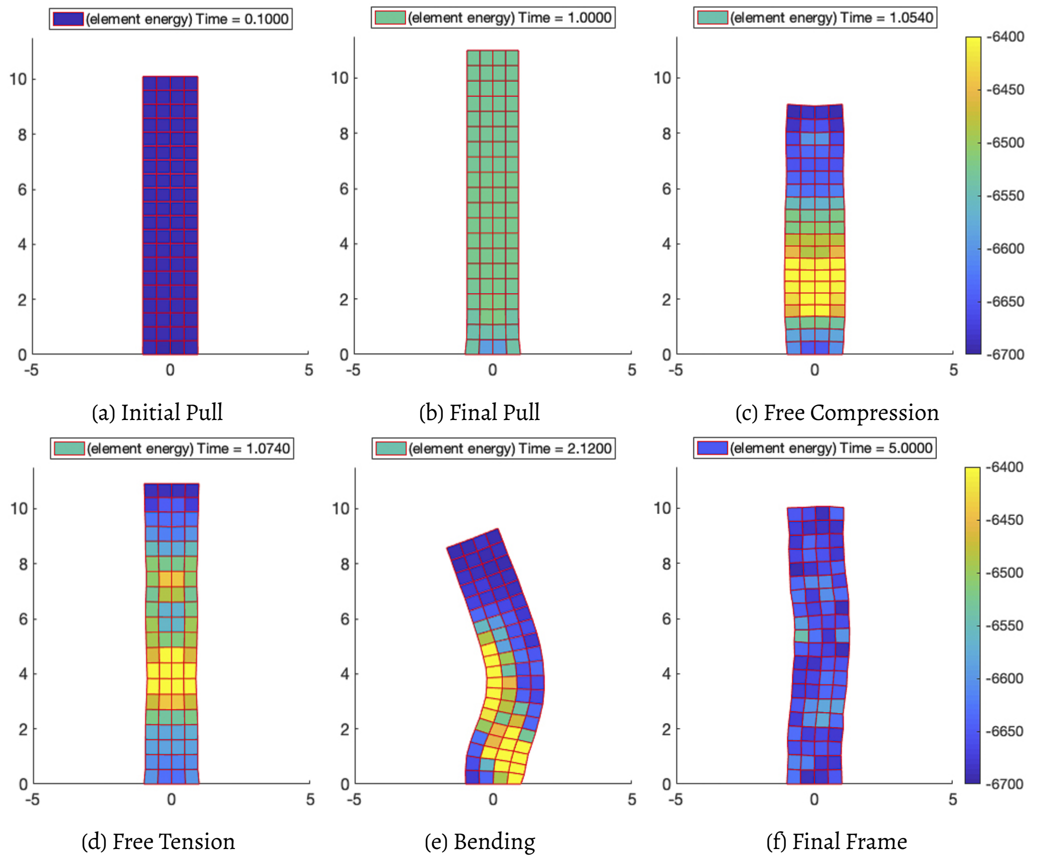
Finally, assembling Eq.12, Eq.15 and Eq.19, the global tangent matrix in Eq.11 could be formed. When the tangent matrix was known, iterative linear system solver could be used to solve Eq.11. Each solution of Eq.11 was the incremental displacement vector in the Newton-Raphson iteration, the solution should added to the current displacement vector. When the residual of current displacement vector was less than some tolerance, Eq.10, hence Eq.7, was successfully solved. To this end, we progress the displacement vector from $t_n$ to $t_n + 1$. Following the displacement, velocity and acceleration vector could be updated using Eq.8 and Eq.9.

## 5.5 General FE Implementation

Proceeding derivations showed all mathematical and computational details. To solve the real finite element problem, firstly we need a mesh. Mesh was made of node list and element list. Secondly we prepared the output files. Thirdly, the prescribed boundary conditions should be initialized. Then, the program called the Newmark Integrator to start the simulation and incrementally push the time forward. Meanwhile, all displacement, velocity and acceleration vectors, as well as the residual of Newton-Raphson iteration, at each time step was written in the output files. The difference between PULLUP stage and VIBRATION stage was the boundary condition. Finally, when the main program stopped, the solving process was done.

## 5.6 Correct Results from Matlab

The top nodes of the 2D rubber bar was pulled up to $u = 1m$ at $t = 1.000s$, then released to introduce free vibration. Some representative frame was list in Fig.2. According to the strain contouring in Fig.2, as the top nodes moving up, the strain energy in the bar was increased. When the prescribed boundary condition was released, the bar started vibrating and cyclically generated compression and tension strain within the structure. Analytically, the bar should vibrating symmetrically forever. However, due to accumulation of numerical error, the vibration of bar became unbalanced and unpredictable after around $t = 2s$. Which could be clearly showed in the last two subfigure in the Fig.2.



(a) Initial Pull (b) Final Pull (c) Free Compression

(d) Free Tension (e) Bending (f) Final Frame

Figure 2: Deformed mesh colored by strain energy per element