EasyChair Preprint

№ 10298

A Fast Algorithm for the Inversion of the Biharmonic in Plate Dynamics Applications

Zehao Wang and Miller Puckette

May 30, 2023

A Fast Algorithm for the Inversion of the Biharmonic in Plate Dynamics Applications

Zehao Wang
Department of Music
University of California, San Diego
La Jolla, CA, USA
zehaowang@ucsd.edu

Miller Puckette
Department of Music
University of California, San Diego
La Jolla, CA, USA
msp@ucsd.edu

ABSTRACT

In this paper, we present a numerical method for solving the biharmonic equation using finite difference methods, which can be used for fast acoustic simulation with nonlinear plate dynamics. With the simply supported boundary condition, the linear system could be regarded as a composition of two Poisson's equations, and these Poisson's equations are solved by the Thomas algorithm for a series of tridiagonal systems after transpositions and linear transformations for vectors in the systems and all nonempty blocks of the Laplacian matrix. We also point out that the eigendecomposition used for these linear transformations has a closed-form formula, which is easy to be pre-computed and also space-saving. Furthermore, since this solver is computed block by block and does not need sparse matrix operations, this method is good for single instruction multiple data (SIMD) parallelization using advanced vector extensions (AVX) intrinsics on central processing units (CPUs), which makes it possible to execute at high speeds for real-time music applications. We also show that this solver for the simply supported boundary condition can also be easily adapted for other boundary conditions using Woodbury matrix identity with a little extra complexity. Numerical experiments show that the C++ implementation of this method is faster than decomposition-based solvers (like LU or Cholesky decomposition) of some well-known C++ libraries at the scale of applications in the field of musical acoustics.

1. INTRODUCTION

Physical modeling methods have a long-established history in simulating musical instruments. This involves representing a particular musical instrument using a system of differential equations, which can be solved using various numerical techniques such as finite-difference, finite-element, and finite-volume methods. The application of physical modeling extends to both musical acoustics, facilitating an examination of the intricate dynamics of musical instruments, and sound synthesis. Of particular importance are the strongly nonlinear effects that underlie the behavior of numerous musical instruments, which present significant challenges in terms of algorithmic design and computation cost.

The simulation of nonlinear plate dynamics problems, such as that of von Kármán [1, 2], typically requires the inversion of the biharmonic operator, which is a computational bottleneck, which guides us to find a fast solver for the linear system with the biharmonic operator.

In the realm of scientific computing and computational mathematics, a range of sparse matrix solvers have been developed to address numerical partial differential equations (PDEs) using methods like fast Fourier transform (FFT) [3], matrix decomposition, or iterative approaches [4]. However, the computational concerns that these methods address are generally related to scalability, which differs significantly from the needs of acoustic simulation, particularly in fast simulation scenarios. In general, algorithms for fast acoustic simulation like sound synthesis should be suitable for low-level SIMD parallelization on CPUs such as advanced vector extensions (AVX) intrinsics, since such optimization methods show great efficiency in the application of fast musical acoustic simulation scenarios using finite-difference schemes [5]. While methods that exploit the structure of sparse matrices, such as FFT-based or cyclic-reduction-based methods [3, 6], can be effective, they may not perform optimally for the scale of musical instrument simulation, since the scale of problems involved in this application is relatively small. For instance, some cyclic-reduction-based methods stop doing cyclic reductions when the matrix size is around 3×3 to 7×7 and directly solve it instead. Nonetheless, in many musical instrument simulation problems, the grid size required for acceptable sound quality ranges from 15×15 to 40×40, meaning that only a limited amount of time for cyclic reductions will be executed, and these operations may take extra time, which is a significant concern at this scale and cannot be optimized much by low-level SIMD parallelization.

In order to accelerate the performance of specific problems of modeling nonlinear plate dynamics, i.e., the fast inverse of the biharmonic operator, we propose a method adapted from [6] to solve the linear system that appeared in most of the schemes for this model with a high speed. After a series of transpositions and linear transformations derived from the closed-form eigendecomposition of a given matrix, the original system could be solved by applying

Thomas algorithm [7] which only requires linear time cost to diagonal blocks. Optimization techniques for C++ implementations like loop unrolling or SIMD parallelization using AVX intrinsics which are compatible with different platforms are also proposed to achieve high speed. Numerical results show that the C++ implementation of this solver could be optimized a lot by those techniques, and its performance is much better than several widely-used solvers, and the timing results show the possibility of fast simulation of plate models with nonlinear plate dynamics. A real-time algorithm for the solution of the von Kármán system for real-time synthesis of gong-like sounds is under development [8].

2. PRELIMINARIES

2.1 The von Kármán plate model

For modeling the nonlinear vibration of plates at moderate amplitudes, the following von-Kármán equation is commonly used:

$$\rho H u_{tt} = -D\Delta\Delta u + \mathcal{L}(\Phi, u), \tag{1a}$$

$$\Delta\Delta\Phi = -\frac{EH}{2}\mathcal{L}(u, u), \tag{1b}$$

where

$$\mathcal{L}(\alpha, \beta) = \alpha_{xx}\beta_{yy} + \alpha_{yy}\beta_{xx} - 2\alpha_{xy}\beta_{xy}, \tag{2}$$

$\Phi(x, y, t)$ is the airy stress function.

The followings are two sets of boundary conditions (clamped and simply supported, respectively) over the boundary ∂U of the domain U:

$$u = \frac{\partial}{\partial \mathbf{n}} u = 0, \quad \Phi = \frac{\partial}{\partial \mathbf{n}}\Phi = 0 \qquad \text{clamped}, \tag{3a}$$

$$u = \Delta_{\mathrm{n}} u = 0, \quad \Phi = \Delta_{\mathrm{n}}\Phi = 0 \qquad \text{simply supported}, \tag{3b}$$

where $\frac{\partial}{\partial \mathbf{n}}$ and Δ_{n} denote the first-order and second-order scalar derivative in the normal direction of the boundary ∂U.

This paper is not concerned with the numerical solution of the von Kármán equations, but rather with the problem of the inversion of the biharmonic operator that appears in (1b).

2.2 Grid functions and finite difference operators

Assume the domain of interest U is a rectangular domain with side lengths $L_x \times L_y$, and its discretization is $N_x \times N_y$ with grid spacing $h_x = h_y = h$, where $h_x = L_x/N_x$ and $h_y = L_y/N_y$.

For a given grid function $u_{l,m}$, define the following spatial difference operator to approximate the derivative operators:

$$\delta_{x\pm} \triangleq \pm\frac{1}{h}\left(e_{x\pm} - 1\right) \approx \frac{\partial}{\partial x}, \delta_{y\pm} \triangleq \pm\frac{1}{h}\left(e_{y\pm} - 1\right) \approx \frac{\partial}{\partial y},$$

where $e_{x\pm} u^n_{l,m} = u^n_{l\pm1,m}$ and $e_{y\pm} u^n_{l,m} = u^n_{l,m\pm1}$. Then we define centered second derivative approximations as follows,

$$\delta_{xx} = \delta_{x+}\delta_{x-} \approx \frac{\partial^2}{\partial x^2}, \quad \delta_{yy} = \delta_{y+}\delta_{y-} \approx \frac{\partial^2}{\partial y^2}$$

The Laplacian and biharmonic operators may then be approximated as

$$\delta_{\Delta\boxplus} = \delta_{xx} + \delta_{yy} \approx \Delta, \quad \delta_{\Delta\boxplus,\Delta\boxplus} \triangleq \delta_{\Delta\boxplus}\delta_{\Delta\boxplus} \approx \Delta\Delta.$$

With simply support boundary condition, we can also write them in matrix form as

$$\mathbf{D}_{\boldsymbol{\Delta}} = \mathbf{L}/h^2, \quad \mathbf{D}_{\boldsymbol{\Delta\Delta}} = \mathbf{D}_{\boldsymbol{\Delta}}\mathbf{D}_{\boldsymbol{\Delta}} = \mathbf{L}^2/h^4 = \mathbf{B}/h^4,$$

here

$$\mathbf{L} = \begin{bmatrix} \mathbf{A} & \mathbf{I} & & & 0 \\ \mathbf{I} & \mathbf{A} & \mathbf{I} & & \\ & \cdots & \cdots & & \\ & & \cdots & \cdots & \\ & & \mathbf{I} & \mathbf{A} & \mathbf{I} \\ 0 & & & \mathbf{I} & \mathbf{A} \end{bmatrix} \in \mathbb{R}^{NN \times NN}, \tag{4}$$

where $NN = (N_y - 1)(N_x - 1)$, L is the discrete Laplacian operator, $I \in \mathbb{R}^{(N_y-1)\times(N_y-1)}$ is the identity matrix, and

$$\mathbf{A} = \begin{bmatrix} -4 & 1 & & & 0 \\ 1 & -4 & 1 & & \\ & \cdots & \cdots & & \\ & & \cdots & \cdots & \\ & & 1 & -4 & 1 \\ 0 & & & 1 & -4 \end{bmatrix} \in \mathbb{R}^{(N_y-1)\times(N_y-1)}.$$

2.3 FDTD schemes

To numerically solve the system (1), a number of schemes could be used [2, 9–11]. Here we only focus on the major computational bottleneck of FDTD schemes which is to solve discrete Φ from (1b), which requires us to find the solution of a linear system. In general, the linear system should have the following formula,

$$\delta^4[\Phi] = d, \tag{5}$$

where δ^4 is the discrete counterpart of $\Delta\Delta$, and d is the discrete vector of the right-hand side of Eq. (1b) derived by a given FDTD scheme. With the simply supported boundary condition which is often used for sound synthesis [2, 10], the form of δ^4 we consider here is $\mathbf{D}_{\boldsymbol{\Delta\Delta}}$. Thus, the linear system to be solved is equivalent to

$$\mathbf{B}[\Phi] = h^4\mathbf{D}_{\boldsymbol{\Delta\Delta}}[\Phi] = h^4 d. \tag{6}$$

2.4 A decomposition of A

Notice that $\mathbf{A}$ has a decomposition $\mathbf{Q}^*\mathbf{V}\mathbf{Q}$, where $\mathbf{Q}$ is a unitary matrix [1] and $\mathbf{V}$ is a diagonal matrix. $\mathbf{Q}$ and $\mathbf{V}$ have the following closed-form formulas,

$$\mathbf{Q}_{kj} = \sqrt{\frac{2}{N_y}}\sin\left(\frac{kj\pi}{N_y}\right), \tag{7}$$

---

[1] $\mathbf{Q} = \mathbf{Q}^*$ and $\mathbf{Q}\mathbf{Q}^* = \mathbf{I}$, actually here we have $\mathbf{Q}* = \mathbf{Q}^T$.

$$\mathbf{V}_{kk} = 2\cos\left(\frac{k\pi}{N_y}\right) - 4, \tag{8}$$

where $1 \leq k, j \leq N_y - 1$. To prove this decomposition, we only need to prove the following lemma:

Lemma 2.1. *$\mathbf{A}$'s $(N_y - 1)$ distinct eigenvalues are $\mathbf{V}_{11}, \mathbf{V}_{22}, \ldots, \mathbf{V}_{(N_y-1)(N_y-1)}$, and $\mathbf{Q}_{k*}$ is the unit eigenvector of $\mathbf{A}$ with respect to $\mathbf{V}_{kk}$, $1 \leq k \leq N_y - 1$.*

The proof is shown in Section 7.

2.5 Thomas algorithm for tridiagonal systems

The Thomas algorithm [7] for tridiagonal systems (9) is shown in Algorithm 1.

$$\mathbf{M}x = \mathbf{M}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = y, \tag{9}$$

where

$$\mathbf{M} = \begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \cdots & \cdots & & \\ & & \cdots & \cdots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix}_{n\times n},$$

Algorithm 1 Thomas algorithm

Input: $y = [y_1, y_2, \ldots, y_n]^T, b = [b_1, b_2, \ldots, b_n]^T \in \mathbb{R}^n$,
$a = [a_2, a_3, \ldots, a_n]^T, c = [c_1, c_2, \ldots, c_{n-1}]^T \in \mathbb{R}^{n-1}$

Output: $x = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$

```
function ThomasAlgorithm($a, b, c, y$)
    Forward elimination:
    for $i = 2$ to $n$ do
        $w \leftarrow a_{i-1}/b_{i-1}$
        $b_i \leftarrow b_i - wc_{i-1}$
        $y_i \leftarrow y_i - wy_{i-1}$
    end for
    $x_n \leftarrow y_n/b_n$
    Backward substitution:
    for $i = n - 1$ to 1 by -1 do
        $x_i \leftarrow (y_i - c_i x_{i+1})/b_i$
    end for
    return $x$
end function
```

A simple sufficient condition to ensure the stability of Algorithm 1 is diagonally dominant (either by row or column) [4][2], which means $|b_i| \geq |a_i| + |c_i|$, $i = 1, 2, \ldots n$ for system (9)[3].

3. THE BIHARMONIC SOLVER

In short, the linear system (6) could be solved by the Thomas algorithm for a series of tridiagonal systems after transpositions and linear transformations for vectors in the systems and all non-empty blocks of the Laplacian matrix. In this section, we will develop details of the biharmonic solver.

Since the discrete biharmonic system with simply supported conditions can be regarded as a composition of two discrete Laplacian systems,

$$b = \mathbf{B}x = \mathbf{LL}x, \tag{10}$$

the system could be solved by applying the above Laplacian solver twice,

$$\begin{cases} \mathbf{L}v = b \\ \mathbf{L}x = v \end{cases}, \tag{11}$$

which means we first solve v from the first equation, and solve u from the second equation. Thus, we first introduce the solver for discrete Laplacian systems using this decomposition and the Thomas algorithm [7][4] from [6] first.

Let

$$\tilde{x} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1(N_x-1)} \\ x_{21} & x_{22} & \cdots & x_{2(N_x-1)} \\ \vdots & \vdots & \vdots & \vdots \\ x_{(N_y-1)1} & x_{(N_y-1)2} & \cdots & x_{(N_y-1)(N_x-1)} \end{bmatrix}$$

be unknown on the grids of the discrete rectangular plate area, and $x = [(\tilde{x}_{*1})^T, (\tilde{x}_{*2})^T, \ldots, (\tilde{x}_{*(N_x-1)})^T]^T$ is the flattened vector of $\tilde{x}$ by column. Thus, the discrete Laplacian system we need to solve is as follows,

$$\mathbf{L}x = b, \tag{12}$$

where $\tilde{b}$ is the known on the grids of the discrete rectangular plate area and b is the flattened vector of $\tilde{b}$ by column. Consider the block structure of L in (4), we have the following equivalent system

$$\begin{array}{ccccccc} & & \mathbf{A}\tilde{x}_{*1} & + & \tilde{x}_{*2} & = & \tilde{b}_{*1}, \\ \tilde{x}_{*(j-1)} & + & \mathbf{A}\tilde{x}_{*j} & + & \tilde{x}_{*(j+1)} & = & \tilde{b}_{*j}, \\ \tilde{x}_{*(N_x-2)} & + & \mathbf{A}\tilde{x}_{*(N_x-1)} & & & = & \tilde{b}_{*(N_x-1)}, \end{array} \tag{13}$$

for $j = 2, 3, \ldots, N_x - 2$.

Consider the transformation for $\bar{x}_{*j} = \mathbf{Q}^*\tilde{x}_{*j}$ and $\bar{b}_{*j} = \mathbf{Q}^*\tilde{b}_{*j}$[5], and multiply Q^* on both sides of each equation in (13), we have the following equivalent system

$$\begin{array}{ccccccc} & & \mathbf{V}\bar{x}_{*1} & + & \bar{x}_{*2} & = & \bar{b}_{*1}, \\ \bar{x}_{*(j-1)} & + & \mathbf{V}\bar{x}_{*j} & + & \bar{x}_{*(j+1)} & = & \bar{b}_{*j}, \\ \bar{x}_{*(N_x-2)} & + & \mathbf{V}\bar{x}_{*(N_x-1)} & & & = & \bar{b}_{*(N_x-1)}, \end{array} \tag{14}$$

for $j = 2, 3, \ldots, N_x - 2$.

---

[2] There are other sufficient conditions for the stability of such systems.

[3] Assume $a_1 = c_n = 0$.

[4] A direct method based on LU decomposition for solving tridiagonal systems with time complexity of $O(n)$, where $n \times n$ is the size of the matrices.

[5] where $\bar{x} = [\bar{x}_{*1}, \bar{x}_{*2}, \ldots, \bar{x}_{*(N_x-1)}]$ and $\bar{b} = [\bar{b}_{*1}, \bar{b}_{*2}, \ldots, \bar{b}_{*(N_x-1)}]$.

Consider each entry in each equation of (14), the system could be rewritten for $k = 1, 2, \dots, N_y - 1$,

$$\begin{array}{ccccccc} & & \mathbf{V}_{kk}\bar{x}_{k1} & + & \bar{x}_{k2} & = & \bar{b}_{k1}, \\ \bar{x}_{k(j-1)} & + & \mathbf{V}_{kk}\bar{x}_{kj} & + & \bar{x}_{k(j+1)} & = & \bar{b}_{kj}, \\ \bar{x}_{k(N_x-2)} & + & \mathbf{V}_{kk}\bar{x}_{k(N_x-1)} & & & = & \bar{b}_{k(N_x-1)}, \end{array} \tag{15}$$

for $j = 2, 3, \dots, N_x - 2$.

Now denote

$$\mathbf{\Gamma}_k = \begin{bmatrix} \mathbf{V}_{kk} & 1 & & & & \\ 1 & \mathbf{V}_{kk} & 1 & & & \\ & \cdots & \cdots & & & \\ & & \cdots & \cdots & & \\ & & 1 & \mathbf{V}_{kk} & 1 & \\ & & & 1 & \mathbf{V}_{kk} & \end{bmatrix}_{(N_x-1)\times(N_x-1)}, \tag{16}$$

for $k = 1, 2, \dots, N_y - 1$, $\hat{x} = \bar{x}^T$, and $\hat{b} = \bar{b}^T$, where $\hat{x}_{*k} = [\bar{x}_{k1}, \bar{x}_{k2}, \dots, \bar{x}_{k(N_x-1)}]^T$ and $\hat{b}_{*k} = [\bar{b}_{k1}, \bar{b}_{k2}, \dots, \bar{b}_{k(N_x-1)}]^T$. Then we have the following system,

$$\mathbf{\Gamma}_k \hat{x}_{*k} = \hat{b}_{*k}, \quad k = 1, 2, \dots, N_y - 1, \tag{17}$$

which is equivalent to (13), (14), and (15). Consider the tridiagonal systems in (17), we can easily show that

$$|\mathbf{V}_{kk}| = |2\cos(\frac{k\pi}{N_y}) - 4| \geq 4 - 2|\cos(\frac{k\pi}{N_y})| \geq |1|,$$

for $k = 1, 2, \dots N_y - 1$, which means all Γ_ks are diagonally dominant so that Algorithm 1 is stable for systems in (17). Since Γ_ks have simple structures, we can simplify Algorithm 1 to Algorithm 2.

Algorithm 2 Thomas algorithm simplified for $\mathbf{\Gamma}_k$s

Input: $\lambda(= V_{kk})$, $y = [y_1, y_2, \dots, y_n]^T \in \mathbb{R}^n$
Output: $x = [x_1, x_2, \dots, x_n]^T \in \mathbb{R}^n$

```
function SimplifiedThomasAlgorithm($\lambda, y$)
    Initialize an empty vector $q = [q_1, q_2, \dots, q_n]^T \in \mathbb{R}^n$
    $q_1 \leftarrow \lambda$
    Forward elimination:
    for $i = 2$ to $n$ do
        $w \leftarrow 1/q_{i-1}$
        $q_i \leftarrow \lambda - w$
        $y_i \leftarrow y_i - w y_{i-1}$
    end for
    $x_n \leftarrow y_n/q_n$
    Backward substitution:
    for $i = n - 1$ to 1 by -1 do
        $x_i \leftarrow (y_i - x_{i+1})/q_i$
    end for
    return x
end function
```

Notice that after we solve the above system, we can easily transform $\bar{x}$ (which equals to $\hat{x}^T$) back into $\tilde{x}$ by $\tilde{x}_{*k} = \mathbf{Q}\bar{x}_{*k}$.

As we can see above, the original discrete Laplacian system (13) could be transformed into a series of tridiagonal systems (17), and those tridiagonal systems could be solved by the Thomas algorithm (shown in Algorithm 1 and 2).

As we mentioned before, to solve the biharmonic system (10), we only need to solve two Laplacian systems (11). However, notice that $\mathbf{QQ}^* = \mathbf{I}$, there are two linear transformations at the beginning ($\mathbf{Q}^*$) and the end of the algorithm ($\mathbf{Q}$), and the solution of the first Laplacian system should be the right-handed side of the second Laplacian system, which means we can simply discard the last transformation in the first solver and the beginning transformation of the second solver without changing the result for the whole biharmonic system. A brief diagram is shown in Fig. 1, and the pseudo-code for this solver is given in Algorithm 3.

3.1 Other boundary conditions

For other boundary conditions, like the clamped boundary condition or the free boundary condition, the matrix δ^4 in Eq. (5) could be written as $B + UV$, where $\mathbf{B} = \mathbf{L}^2 \in \mathbb{R}^{NN\times NN}$ is the discrete biharmonic operator we derived before, $\mathbf{U} \in \mathbb{R}^{NN\times m}$ and $V \in \mathbb{R}^{m\times NN}$. Actually, for most boundary conditions, $\mathbf{V} = \mathbf{U}^T$ and $\mathbf{U}$ is a rank-m sparse matrix with cNN nonzero entries, where $c = 1$ for the clamped boundary condition.

Therefore, one can use the following formula adapted from Woodbury matrix identity [12],

$$(\mathbf{B} + \mathbf{UV})^{-1} = \mathbf{B}^{-1} - \mathbf{B}^{-1}\mathbf{U}\left(\mathbf{I} + \mathbf{VB}^{-1}\mathbf{U}\right)^{-1}\mathbf{VB}^{-1}, \tag{18}$$

which means we need to precompute and store several matrices like $\mathbf{B}^{-1}\mathbf{U}$ using the biharmonic solver we described before, and do some extra sparse matrix-vector multiplication (like vectors multiplied by $\mathbf{U}$) and dense matrix-vector multiplication (like vectors multiplied by $\mathbf{B}^{-1}\mathbf{U}$) at each time step.

4. IMPLEMENTATION

4.1 Implementation and optimization of the biharmonic solver

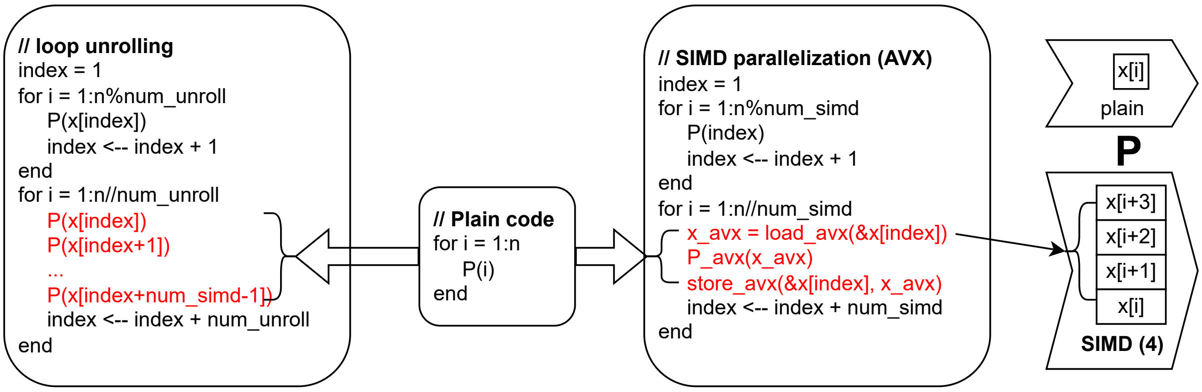
Consider the linear transformation stage and the Thomas algorithm stage, each column of $\tilde{x}$ and $\hat{x}$ is individual, so the optimization technique is to unroll or parallelize using AVX intrinsics every for-loop in Algorithm 3, and use AVX's fused operations like fused multiply-add (fmadd) instead of two separate operations if supported. However, the optimization using AVX is a little complicated since all matrices that will be parallelized need to be stored by some specific orders by column or row [6]. A brief demonstration of these optimization techniques is shown in Fig. 2. Although the transpose operations introduce extra complexity, numerical results from the next section show that the overall performance of the AVX version is better than the plain version and loop-unrolling version, and the loop-unrolling version which can be used if AVX is not compatible with the hardware is a little slower than the AVX version but faster than the plain version.

---

[6] For example, if the index m that will be parallelized indicates m-th column, the array should be flattened by row, and vice versa.

$$\begin{gathered}
\boxed{\mathbf{L}v=b} \Rightarrow b \xrightarrow{\text{flatten}} \tilde{b} \Rightarrow \hat{b}_{j*} = (\bar{b}_{*j})^T = (\mathbf{Q}^*\tilde{b}_{*j})^T \Rightarrow \hat{v}_{*k} = \mathbf{\Gamma}_k^{-1}\hat{b}_{*k} \\
\cancel{\Rightarrow \tilde{v}_{*j} = \mathbf{Q}\bar{v}_{*j} = \mathbf{Q}(\hat{v}_{j*})^T \Rightarrow \tilde{v} \xrightarrow{\text{reshape}} v \Rightarrow \boxed{\mathbf{L}x=v} \Rightarrow v \xrightarrow{\text{flatten}} \tilde{v} \Rightarrow \hat{v}_{j*} = (\bar{v}_{*j})^T = (\mathbf{Q}^*\tilde{v}_{*j})^T} \\
\Rightarrow \hat{x}_{*k} = \mathbf{\Gamma}_k^{-1}\hat{v}_{*k} \Rightarrow \tilde{x}_{*j} = \mathbf{Q}\bar{x}_{*j} = \mathbf{Q}(\hat{x}_{j*})^T \Rightarrow \tilde{x} \xrightarrow{\text{reshape}} x.
\end{gathered}$$

Figure 1. A diagram of the biharmonic solver. The strikethrough across the second line means that these operations can be discarded because they cancel each other out.



```
// loop unrolling
index = 1
for i = 1:n%num_unroll
    P(x[index])
    index <-- index + 1
end
for i = 1:n//num_unroll
    P(x[index])
    P(x[index+1])
    ...
    P(x[index+num_simd-1])
    index <-- index + num_unroll
end
```

```
// Plain code
for i = 1:n
    P(i)
end
```

```
// SIMD parallelization (AVX)
index = 1
for i = 1:n%num_simd
    P(index)
    index <-- index + 1
end
for i = 1:n//num_simd
    x_avx = load_avx(&x[index])
    P_avx(x_avx)
    store_avx(&x[index], x_avx)
    index <-- index + num_simd
end
```

Figure 2. Demonstration for loop unrolling and SIMD parallelization. P means the operation for each time step (including all parameters and coefficients from some data), and x is the array-type data used for and updated by the operation. For the operation using SIMD parallelization, several entries of the data need to be loaded to some consecutive memory addresses, then the single instruction will be applied to these loaded entries simultaneously, and finally, store the result back to the data. Here for AVX2, the number of these double-precision entries for each iteration (num_simd) is 4.

abbr. of platform	machine	operating system	supported instruction sets
MBA	MacBook Air 2020 with 1.1 GHz 4-core Intel i5	MacOS 12	AVX, AVX2
MBP	MacBook Pro 2021 with 10-core M1 Max	MacOS 12	N/A
PC_Linux	AMD Ryzen 7 5800X 8-core 4.7 GHz	Ubuntu 22.04 LTS	AVX, AVX2
PC_Win	AMD Ryzen 7 5800X 8-core 4.7 GHz	Windows 11	AVX, AVX2

Table 1. Systems and hardware for numerical experiments

Algorithm 3 A fast biharmonic solver (simply supported boundary condition)

Input: $\mathbf{Q} \in \mathbb{R}^{(N_y-1)(N_y-1)}$, $\mathbf{V}_{kk}$ $(k = 1, 2, \ldots, N_y - 1)$, $\tilde{b} \in \mathbb{R}^{(N_y-1)(N_x-1)}$
Output: $\tilde{x} \in \mathbb{R}^{(N_y-1)(N_x-1)}$

```
function BiharmonicSolverSS($\mathbf{Q}, \mathbf{V}_{kk}, \tilde{b}$)
    Initialize an empty matrix $\tilde{v} \in \mathbb{R}^{(N_y-1)(N_x-1)}$
    Solve $\mathbf{L}v = b$ and get $\hat{v}$:
    for $j = 1$ to $N_x - 1$ do
        $\hat{b}_{j*} \leftarrow (\mathbf{Q}\tilde{b}_{*j})^T$    ▷ i.e., $(\bar{b}_{*j})^T$, and here we use $\mathbf{Q}$ instead of $\mathbf{Q}^*$ since $\mathbf{Q} = \mathbf{Q}^*$
    end for
    for $k = 1$ to $N_y - 1$ do
        $\hat{v}_{*k} \leftarrow$ SimplifiedThomasAlgorithm($\mathbf{V}_{kk}, \hat{b}_{*k}$)    ▷ Solve $\mathbf{\Gamma}_k \hat{v}_{*k} = \hat{b}_{*k}$
    end for
    Solve $\mathbf{L}x = v$ and get $\tilde{x}$:
    for $k = 1$ to $N_y - 1$ do
        $\hat{x}_{*k} \leftarrow$ SimplifiedThomasAlgorithm($\mathbf{V}_{kk}, \hat{v}_{*k}$)    ▷ Solve $\mathbf{\Gamma}_k \hat{x}_{*k} = \hat{v}_{*k}$
    end for
    for $j = 1$ to $N_x - 1$ do
        $\tilde{x}_{*j} \leftarrow \mathbf{Q}(\hat{x}_{j*})^T$    ▷ i.e., $\mathbf{Q}\bar{x}_{*j}$
    end for
    return $\tilde{x}$
end function
```

In C++ implementation of the biharmonic solver, all matrices and vectors are using double-precision array data type, and matrices are stored by flattening them by column.

4.2 Alternative solvers

In the field of computational mathematics and scientific computing, people usually use either sparse matrix decomposition or iterative methods to solve linear systems [4], and the latter always deal with large-scale linear systems for storage or memory concerns and their time costs are generally higher than decomposition methods. The direct FFT-based solver is also another method to solve Poisson's equation with the Laplacian operator [3]. Therefore, considering the scale of the problem that is focused on in this paper, I'll only run the numerical experiments on the simple FFT-based solver and several decomposition-based solvers for comparison. Here, we use LU- and Cholesky-decomposition-based solvers for sparse matrices from Matlab and Eigen [13], a well-known high-level C++ library for linear algebra, for these numerical experiments.

5. NUMERICAL RESULTS

In this paper, four platforms including three machines and three operating systems listed in Table 1 are used for numerical experiments. The version of Matlab we used is R2022a, and plain code means no optimization techniques are used. All numerical results are cumulative time costs in seconds for 44100 iterations. All code should be compiled with at least -O3/-Ofast and -mavx2 [7] -march=native flags.

5.1 Comparisons between the biharmonic solver and alternative solvers

Here we compare the performance between the biharmonic solver and alternative solvers on PC_Linux platform. The results is shown in Table 2, where Matlab_B means directly solving the system $Bx = b$ using Matlab's default solver $B\backslash b$, Matlab_L^2 means solving two Laplacians using Matlab's default solver $L\backslash L\backslash b$, and Matlab_*_X means solving the linear system regarding B or two Ls by Matlab's default solver using decomposition X (chosen from LU or Cholesky). Eigen's abbreviations are similar to the above Matlab's. All C++ implementations using Eigen are complied with -O3 and -mavx2 -march=native flags since Eigen3.3 we used here supports both -O3 and AVX2 optimization. And we use four sizes of matrices, $(N_x - 1) \times (N_y - 1) = 14 \times 14, 16 \times 20, 23 \times 17, 25 \times 25$ for comparisons.

	14×14	16×20	23×17	25×25
plain code	0.916	2.051	2.202	4.686
loop unrolling	0.887	1.933	1.987	4.592
AVX2	0.551	1.007	1.246	2.385
plain code (-O3)	0.087	0.207	0.181	0.310
loop unrolling (-O3)	0.075	0.220	0.135	0.244
AVX2 (-O3)	**0.049**	**0.111**	**0.101**	**0.236**
Eigen_FFT	0.842	1.049	1.498	2.071
Eigen_B_LU	0.319	0.631	0.812	1.554
Eigen_B_Chol	0.227	0.416	0.527	0.930
Eigen_L^2_LU	0.357	0.854	0.943	1.997
Eigen_L^2_Chol	0.278	0.639	0.727	1.269
Matlab_FFT	1.030	1.960	2.062	2.630
Matlab_B	0.596	0.993	2.113	6.560
Matlab_B_LU	0.378	0.497	0.635	1.024
Matlab_B_Chol	0.363	0.403	0.523	0.919
Matlab_L^2	0.635	1.249	2.395	6.834
Matlab_L^2_LU	0.576	0.657	0.854	1.264
Matlab_L^2_Chol	0.548	0.603	0.732	1.043

Table 2. Numerical results of comparisons between the biharmonic solver and alternative solvers on PC_Linux. The first three rows are compiled without -O3 flag. Bold results are the best results for each column. Unit: second.

5.2 Comparisons of different optimization techniques for the biharmonic solver

Here we compare the performance of different optimization techniques for the implementation of the biharmonic solver on those four platforms. For the sake of brevity, we only show the results of three different optimization techniques compiled with -O3 flag with a grid size of $(N_x - 1) \times (N_y - 1) = 23 \times 17$. The results are shown in Table. 3.

---

[7] For AVX2. Choose -mavx for AVX and -mavx512f for AVX512.

	MBA	MBP	PC_Linux	PC_Win
plain code (-O3)	0.826	0.367	0.181	0.202
loop unrolling (-O3)	0.595	**0.242**	0.135	0.147
AVX2 (-O3)	**0.340**	N/A	**0.101**	**0.118**

Table 3. Numerical results of the biharmonic solver on different platforms. $(N_x - 1) \times (N_y - 1) = 23 \times 17$. Bold results are the best results for each column. Unit: second.

6. CONCLUSIONS

In this paper, we describe an algorithm for solving the discrete biharmonic system for modeling nonlinear plate dynamics based on a series of linear transformations and Thomas algorithm for tridiagonal systems. This method is good for optimization techniques on CPUs like loop unrolling or low-level SIMD parallelization using AVX intrinsics. At the scale of fast musical instrument simulation, the numerical results show that the C++ implementation of this method has better performance than other generally used methods for solving such linear systems like FFT-based and decomposition-based solvers, and indicate fast musical instrument simulation with nonlinear plate dynamics, which is actually used for real-time gong-like musical instruments synthesis based on the von-Kármán plate equation [8].

Acknowledgments

The authors would like to express their gratitude to Prof. Stefan Bilbao for his valuable contributions to this paper including the closed-form formulas presented in Eq. (7) and (8) and his constructive suggestions for modifications.

7. APPENDIX: PROOF OF LEMMA 2.1

Proof. First, it's obvious that $E = \{\mathbf{V}_{kk} \mid i = 1, 2, ..., N_y - 1\}$ has $N_y - 1$ distinct elements.

Then we only need to show the following equations,

$$\mathbf{A}q_k = \left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right) q_k, \quad 1 \le i \le N_y - 1,$$

where $q_{kj} = \sin\left(\frac{kj\pi}{N_y}\right), 1 \le k, j \le N_y - 1$.

For $2 \le j \le N_y - 2$, we have

$$\begin{aligned} & \left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right) q_{kj} \\ = \;& \sin\left(\frac{kj\pi}{N_y}\right)\left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right) \\ = \;& 2\sin\left(\frac{kj\pi}{N_y}\right)\cos\left(\frac{k\pi}{N_y}\right) - 4\sin\left(\frac{kj\pi}{N_y}\right) \\ = \;& \sin\left(\frac{(kj+k)\pi}{N_y}\right) + \sin\left(\frac{(kj-k)\pi}{N_y}\right) - 4\sin\left(\frac{kj\pi}{N_y}\right) \\ = \;& \sin\left(\frac{(k(j-1)\pi}{N_y}\right) - 4\sin\left(\frac{kj\pi}{N_y}\right) + \sin\left(\frac{k(j+1)\pi}{N_y}\right) \\ = \;& \mathbf{A}_{j(j-1)}\sin\left(\frac{(k(j-1)\pi}{N_y}\right) + \mathbf{A}_{jj}\sin\left(\frac{kj\pi}{N_y}\right) \\ & + \mathbf{A}_{jj}\sin\left(\frac{k(j+1)\pi}{N_y}\right) \\ = \;& A_{j*}q_k. \end{aligned}$$

For $j = 1$ or $N_y - 1$, notice that $\sin\left(\frac{k(1-1)\pi}{N_y}\right) = 0$ and $\sin\left(\frac{k(N_y-1+1)\pi}{N_y}\right) = 0$, which means the equation

$$\left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right) q_{kj} = \mathbf{A}_{j*}q_k$$

still holds for $j = 1$ and $N_y - 1$. Therefore, we have

$$\mathbf{A}q_k = \left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right) q_k,$$

which means E is the set of all $\mathbf{A}$'s eigenvalues, and q_k is the eigenvector w.r.t. $\left(2\cos\left(\frac{k\pi}{N_y}\right) - 4\right)$.

Notice that

$$\begin{aligned} ||q_k||_2^2 &= \sum_{j=1}^{N_y-1} \sin\left(\frac{kj\pi}{N_y}\right)^2 \\ &= \sum_{j=1}^{N_y-1} \frac{1 - \cos\left(\frac{2kj\pi}{N_y}\right)}{2} \\ &= \frac{N_y - 1}{2} - \sum_{j=1}^{N_y-1} \frac{\cos\left(\frac{2kj\pi}{N_y}\right)}{2} \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\sum_{j=1}^{N_y-1} \mathbf{Re}\left(\exp\left(\frac{2kj\pi i}{N_y}\right)\right) \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\mathbf{Re}\left(\sum_{j=1}^{N_y-1} \exp\left(\frac{2kj\pi i}{N_y}\right)\right) \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\mathbf{Re}\left(\frac{\exp\left(\frac{2k\pi i}{N_y}\right) - \exp\left(\frac{2kN_y\pi i}{N_y}\right)}{1 - \exp\left(\frac{2k\pi i}{N_y}\right)}\right) \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\mathbf{Re}\left(\frac{\exp\left(\frac{2k\pi i}{N_y}\right) - \exp\left(2k\pi i\right)}{1 - \exp\left(\frac{2k\pi i}{N_y}\right)}\right) \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\mathbf{Re}\left(\frac{\exp\left(\frac{2k\pi i}{N_y}\right) - 1}{1 - \exp\left(\frac{2k\pi i}{N_y}\right)}\right) \\ &= \frac{N_y - 1}{2} - \frac{1}{2}\mathbf{Re}\left(-1\right) \\ &= \frac{N_y}{2}, \end{aligned}$$

for every $1 \le k \le N_y - 1$, where $i = \sqrt{-1}$. Thus, we have $||q_k||_2 = \frac{N_y}{2}$, and

$$\left[\frac{q_1}{||q_1||_2}, \frac{q_2}{||q_2||_2}, \cdots, \frac{q_{N_y-1}}{||q_{N_y-1}||_2}\right]^T = \mathbf{Q},$$

which leads to the following decomposition

$$\mathbf{QVQ}^* = \mathbf{A}.$$

□

8. REFERENCES

[1] T. von Kármán, “Festigkeitsprobleme im maschinenbau,” *Encyklopädie der Mathematischen Wissenschaften*, vol. 4, no. 4, pp. 311–385, 1910.

[2] S. Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. Chichester, UK: John Wiley and Sons, 2009.

[3] C. Temperton, “Direct methods for the solution of the discrete poisson equation: some comparisons,” *Journal of Computational Physics*, vol. 31, no. 1, pp. 1–20, 1979.

[4] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2013.

[5] Z. Wang, M. Puckette, T. Erbe, and S. Bilbao, “Real-time implementation of the kirchhoff plate equation using finite-difference time-domain methods on cpu,” *2023 Sound and Music Computing Conference (SMC)*, 2023.

[6] B. L. Buzbee, G. H. Golub, and C. W. N. R. work(s):, “On Direct Methods for Solving Poisson’s Equations,” *SIAM Journal on Numerical Analysis*, vol. 7, no. 4, pp. 627–656, 1970. [Online]. Available: http://www.jstor.org/stable/2949380

[7] L. Thomas, “Elliptic problems in linear differential equations over a network: Watson scientific computing laboratory,” *Columbia Univ., NY*, 1949.

[8] S. Bilbao, C. Webb, Z. Wang, and M. Ducceschi, “Real-time gong synthesis,” in *Proceedings of the 26th International Conference on Digital Audio Effects*, 2023.

[9] S. Bilbao, “A family of conservative finite difference schemes for the dynamical von karman plate equations,” *Numerical Methods for Partial Differential Equations: An International Journal*, vol. 24, no. 1, pp. 193–216, 2008.

[10] S. Bilbao and M. Ducceschi, “Fast explicit algorithms for Hamiltonian numerical integration,” in *Proc. Eur. Nonlinear Dynamics Conf.*, Lyon, France, July 2022.

[11] S. Bilbao, M. Ducceschi, and F. Zama, “Explicit exactly energy-conserving methods for Hamiltonian systems,” *J. Comp. Phys.*, vol. 427, p. 111697, 2023.

[12] W. W. Hager, “Updating the inverse of a matrix,” *SIAM review*, vol. 31, no. 2, pp. 221–239, 1989.

[13] G. Guennebaud, B. Jacob *et al.*, “Eigen v3,” http://eigen.tuxfamily.org, 2010.