Implicit-shifted Symmetric QR Singular Value Decomposition of 3×3 Matrices

Theodore Gast[*1], Chuyuan Fu[†1], Chenfanfu Jiang[‡1,2], and Joseph Teran[§1]

[1]Department of Mathematics, University of California, Los Angeles
[2]Department of Computer Science, University of California, Los Angeles

Abstract

Computing the Singular Value Decomposition (SVD) of 3×3 matrices is commonplace in 3D computational mechanics and computer graphics applications. We present a C++ implementation of implicit symmetric QR SVD with Wilkinson shift. The method is fast and robust in both float and double precisions. We also perform a benchmark test to study the performance compared to other popular algorithms.

Keywords: SVD, implicit symmetric QR, Wilkinson shift, Jacobi rotation, eigenvalue, Givens rotation

1 Problem Description

Our goal is finding the SVD of a real 3×3 matrix $\mathbf{A}$ so that

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T,$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, $\boldsymbol{\Sigma}$ is a diagonal matrix consisting of the singular values of $\mathbf{A}$. In computational mechanics, $\mathbf{U}$ and $\mathbf{V}$ are often enforced to be rotation matrices which better represent geometric transformations. Furthermore, many authors use the conventions as in [Irving et al. 2004], e.g., [Sin et al. 2011; Stomakhin et al. 2012; Hegemann et al. 2013; Stomakhin et al. 2013; Bouaziz et al. 2014; Stomakhin et al. 2014; Saito et al. 2015; Gast et al. 2015; Xu et al. 2015; Klar et al. 2016]. The conventions are

- $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$;
- $det(\mathbf{U}) = 1$, $det(\mathbf{V}) = 1$;
- $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$.

Note that $\sigma_3 < 0$ if $det(\mathbf{A}) < 0$.

2 Givens Rotation

The QR algorithm largely depends on Givens rotations. Once any c and s with $c^2 + s^2 = 1$ are computed from inputs x and y, a 2D Givens rotation is defined as

$$\mathbf{G}_2(1, 2, c(x, y), s(x, y)) = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}.$$

[*]tfg@math.ucla.edu
[†]fuchuyuan@math.ucla.edu
[‡]cffjiang@math.ucla.edu
[§]jteran@math.ucla.edu

Algorithm 1 Constructing a 3D Givens rotation

```
1:  procedure GivensConventional($x, y$)
2:      $d \leftarrow a^2 + b^2$
3:      $c \leftarrow 1$
4:      $s \leftarrow 0$
5:      if $d \neq 0$ then                    ▷ no tolerance needed
6:          $t \leftarrow rsqrt(d)$            ▷ fast inverse square root
7:          $c \leftarrow at$
8:          $s \leftarrow -bt$
9:      return $(c, s)$
10: procedure GivensUnconventional($x, y$)
11:     $d \leftarrow a^2 + b^2$
12:     $c \leftarrow 0$
13:     $s \leftarrow 1$
14:     if $d \neq 0$ then                    ▷ no tolerance needed
15:         $t \leftarrow rsqrt(d)$            ▷ fast inverse square root
16:         $s \leftarrow at$
17:         $c \leftarrow bt$
18:     return $(c, s)$
```

Algorithm 2 Fast inverse square root for float

```
1: procedure ApproximateRsqrt($a$)            ▷ inaccurate
2:     return $SIMD\_RSQRT(a)$
3: procedure Rsqrt($a$)                       ▷ much more accurate
4:     $b \leftarrow ApproximateRsqrt(a)$
5:     $b \leftarrow \frac{1}{2}(3 - ab^2)b$     ▷ Newton step with $f(x) = a - 1/x^2$
6:     return $b$
```

In 3D, there are 3 cases:

$$\mathbf{G}_3(1, 2, c(x, y), s(x, y)) = \begin{pmatrix} c & s & \\ -s & c & \\ & & 1 \end{pmatrix},$$

$$\mathbf{G}_3(2, 3, c(x, y), s(x, y)) = \begin{pmatrix} 1 & & \\ & c & s \\ & -s & c \end{pmatrix},$$

$$\mathbf{G}_3(1, 3, c(x, y), s(x, y)) = \begin{pmatrix} c & & s \\ & 1 & \\ -s & & c \end{pmatrix}.$$

s and c are usually constructed so that

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} * \\ 0 \end{pmatrix}.$$

Sometimes we also need to construct them so that

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ * \end{pmatrix},$$

we denote this *unconventional* Givens rotation with $\hat{\mathbf{G}}$ instead of $\mathbf{G}$. Algorithm 1 shows the pseudocode for constructing $\mathbf{G}$

Algorithm 3 Polar Decomposition of 2×2 matrices

```
1: procedure PolarDecomposition2D($\mathbf{A}$)
2:     $x \leftarrow A_{11} + A_{22}$
3:     $y \leftarrow A_{21} - A_{12}$
4:     $d \leftarrow \sqrt{x^2 + y^2}$
5:     $\mathbf{R} \leftarrow \mathbf{G}_2(1, 2, c = 1, s = 0)$          ▷ $\mathbf{R}$ is a Givens rotation
6:     if $d \neq 0$ then                                  ▷ no tolerance needed
7:         $\mathbf{R} \leftarrow \mathbf{G}_2(1, 2, c = x/d, s = -y/d)$
8:     $\mathbf{S} \leftarrow RowRotation(\mathbf{R}, \mathbf{A})$                  ▷ $\mathbf{S} = \mathbf{R}^T\mathbf{A}$
9:     return $(\mathbf{R}, \mathbf{S})$                ▷ $\mathbf{R}$ is a rotation, $\mathbf{S}$ is symmetric
```

and $\hat{\mathbf{G}}$. Note that we use a fast inverse square root function (Algorithm 2) from Streaming SIMD Extensions (SSE) intrinsics to accelerate the float case (we use the c++ function $\_mm\_cvtss\_f32(\_mm\_rsqrt\_ss(\_mm\_set\_ss(a)))$). Similarly to [McAdams et al. 2011], accuracy is improved by performing an additional Newton step. In the case of double precision, we simply use the standard C++ square root function to maintain accuracy.

We further use the definition that

- $\mathbf{B} = RowRotation(\mathbf{G}, \mathbf{A})$ means $\mathbf{B} = \mathbf{G}^T\mathbf{A}$,
- $\mathbf{B} = ColumnRotation(\mathbf{G}, \mathbf{A})$ means $\mathbf{B} = \mathbf{A}\mathbf{G}$.

In practice these operations are implemented more efficiently by updating four entries of $\mathbf{A}$ in place instead of performing matrix products.

3 SVD of 2×2 Matrices

As the to-be-presented algorithm proceeds, the problem will eventually degrade into computing the SVD of a 2×2 matrix. Here we briefly describe how to do so while obeying a similar sign convention ($\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_1 \geq |\sigma_2|$).

Assuming $\mathbf{A}$ is 2×2, the first step is computing its Polar Decomposition $\mathbf{A} = \mathbf{RS}$, where $\mathbf{R}$ is a rotation and $\mathbf{S}$ is symmetric. Assuming

$$\mathbf{R} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix},$$

requiring $\mathbf{R}^T\mathbf{A}$ being symmetric leads to $xs = yc$ where $x = A_{11} + A_{22}, y = A_{21} - A_{12}$. The two solutions are therefore

$$c = \frac{x}{\sqrt{x^2 + y^2}}, \quad s = \frac{y}{\sqrt{x^2 + y^2}}$$

or

$$c = \frac{-x}{\sqrt{x^2 + y^2}}, \quad s = \frac{-y}{\sqrt{x^2 + y^2}}.$$

By taking the difference of $\|\mathbf{S} - \mathbf{I}\|_F^2$ from two solutions, it can be shown that choosing the first one always minimizes it, therefore guarantees the chosen $\mathbf{R}$ is the closest rotation to $\mathbf{A}$ (or $\mathbf{S}$ is the closest symmetric matrix to $\mathbf{I}$).

Once we have the symmetric matrix $\mathbf{S}$, diagonalizing it with a Jacobi rotation can be done similarly by solving c and s from

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} S_{11} & S_{12} \\ S_{12} & S_{22} \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} * & \\ & * \end{pmatrix}.$$

Attentions need to be paid to prevent potential division by zero [Golub and Van Loan 2012]. Finally,

$$\mathbf{V} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix},$$
$$\mathbf{U} = \mathbf{RV},$$
$$\mathbf{\Sigma} = \mathbf{V}^T\mathbf{SV}.$$

After sorting σ_1, σ_2 to obey our sign convention and permuting columns of $\mathbf{U}$ and $\mathbf{V}$ accordingly, we have $\mathbf{A} = \mathbf{U\Sigma V}^T$ where

- $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$;
- $det(\mathbf{U}) = 1$, $det(\mathbf{V}) = 1$;
- $\sigma_1 \geq |\sigma_2|$.

2×2 Polar Decomposition and SVD are shown in Algorithm 3 and 4.

Algorithm 4 Singular Value Decomposition of 2×2 matrices

```
 1: procedure Svd2D($\mathbf{A}$)
 2:     $(\mathbf{R}, \mathbf{S}) \leftarrow PolarDecomposition2D(\mathbf{A})$
 3:     if $S_{12} = 0$ then                         ▷ $\mathbf{S}$ is already diagonal
 4:         $(\hat{c}, \hat{s}) \leftarrow (1, 0)$
 5:         $(\sigma_1, \sigma_2) \leftarrow (S_{11}, S_{22})$
 6:     else
 7:         $\tau \leftarrow \frac{1}{2}(S_{11} - S_{22})$
 8:         $w \leftarrow \sqrt{\tau^2 + S_{12}^2}$
 9:         $t = (\tau > 0)? \frac{S_{12}}{\tau + w} : \frac{S_{12}}{\tau - w}$              ▷ division is safe
10:         $\hat{c} \leftarrow 1/\sqrt{t^2 + 1}$
11:         $\hat{s} \leftarrow -t\hat{c}$
12:         $\sigma_1 \leftarrow \hat{c}^2 S_{11} - 2\hat{c}\hat{s}S_{12} + \hat{s}^2 S_{22}$
13:         $\sigma_2 \leftarrow \hat{s}^2 S_{11} + 2\hat{c}\hat{s}S_{12} + \hat{c}^2 S_{22}$
14:     if $\sigma_1 < \sigma_2$ then                                    ▷ sorting
15:         $swap(\sigma_1, \sigma_2)$
16:         $\mathbf{V} \leftarrow \mathbf{G}_2(1, 2, -\hat{s}, \hat{c})$
17:     else
18:         $\mathbf{V} \leftarrow \mathbf{G}_2(1, 2, \hat{c}, \hat{s})$
19:     $\mathbf{U} \leftarrow \mathbf{RV}$
20:     return $(\mathbf{U}, \sigma_1, \sigma_2, \mathbf{V})$        ▷ $\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_1 \geq |\sigma_2|$
```

4 Implicit Symmetric QR SVD

The QR algorithm iteratively applies Givens rotations to a tridiagonal symmetric matrix (which in the SVD case corresponds to $\mathbf{T} = \mathbf{A}^T\mathbf{A}$) to solve the symmetric eigenproblem. Instead of constructing $\mathbf{T}$, implicit symmetric QR SVD works on an upper bidiagonal $\mathbf{A}$ and implicitly does the same thing. This results in a much higher accuracy and improves efficiency [Golub and Van Loan 2012].

4.1 Bidiagonalization and Zerochasing

The implicit symmetric QR algorithm starts with making $\mathbf{A}$ upper bidiagonal. For 3×3 matrices, this can be done with 4 Givens

Algorithm 5 Zerochasing: Assuming input $A_{31} = 0$, $\mathbf{U}, \mathbf{V}$ are rotations, this function makes $\mathbf{A}$ upper bidiagonal while maintaining the product $\mathbf{U}\mathbf{A}\mathbf{V}^T$ unchanged

```
1: procedure ZEROCHASING($\mathbf{U}, \mathbf{A}, \mathbf{V}$)   ▷ update them in place
2:     $\mathbf{G} \leftarrow \mathbf{G}_3(2, 3, x = A_{12}, y = A_{13})$
3:     $\mathbf{A} \leftarrow \mathbf{A}\mathbf{G}, \quad \mathbf{U} \leftarrow \mathbf{G}^T\mathbf{U}$
4:     $\mathbf{G} \leftarrow \mathbf{G}_3(2, 3, x = A_{12}, y = A_{13})$
5:     $\mathbf{A} \leftarrow \mathbf{G}^T\mathbf{A}, \quad \mathbf{V} \leftarrow \mathbf{G}^T\mathbf{V}$
6:     $\mathbf{G} \leftarrow \mathbf{G}_3(2, 3, x = A_{22}, y = A_{32})$
7:     $\mathbf{A} \leftarrow \mathbf{G}^T\mathbf{A}, \quad \mathbf{U} \leftarrow \mathbf{U}\mathbf{G}$
8:     return $(\mathbf{U}, \mathbf{A}, \mathbf{V})$
```

Algorithm 6 Upper Bidiagonalizing: Assuming input $\mathbf{U}, \mathbf{V}$ are rotations, this function makes $\mathbf{A}$ upper bidiagonal while maintaining the product $\mathbf{U}\mathbf{A}\mathbf{V}^T$ unchanged

```
1: procedure BIDIAGONALIZE($\mathbf{U}, \mathbf{A}, \mathbf{V}$) ▷ update them in place
2:     $\mathbf{G} \leftarrow \mathbf{G}_3(2, 3, x = A_{21}, y = A_{31})$
3:     $\mathbf{A} \leftarrow \mathbf{G}^T\mathbf{A}, \quad \mathbf{U} \leftarrow \mathbf{U}\mathbf{G}$
4:     $(\mathbf{U}, \mathbf{A}, \mathbf{V}) \leftarrow Zerochasing(\mathbf{U}, \mathbf{A}, \mathbf{V})$
5:     return $(\mathbf{U}, \mathbf{A}, \mathbf{V})$
```

rotation. Starting with $\mathbf{A}^{(0)} = \mathbf{A}$,

$$\mathbf{A}^{(1)} = \mathbf{G}_3^{(1)}(2, 3, x = A_{21}^{(0)}, y = A_{31}^{(0)})^T \mathbf{A}^{(0)} = \begin{pmatrix} * & * & * \\ * & * & * \\ - & * & * \end{pmatrix},$$

$$\mathbf{A}^{(2)} = \mathbf{G}_3^{(2)}(1, 2, x = A_{11}^{(1)}, y = A_{21}^{(1)})^T \mathbf{A}^{(1)} = \begin{pmatrix} * & * & * \\ - & * & * \\ & * & * \end{pmatrix},$$

$$\mathbf{A}^{(3)} = \mathbf{A}^{(2)}\mathbf{G}_3^{(3)}(2, 3, x = A_{12}^{(2)}, y = A_{13}^{(1)}) = \begin{pmatrix} * & * & - \\ & * & * \\ & * & * \end{pmatrix},$$

$$\mathbf{A}^{(4)} = \mathbf{G}_3^{(4)}(2, 3, x = A_{22}^{(3)}, y = A_{32}^{(3)})^T \mathbf{A}^{(3)} = \begin{pmatrix} * & * & \\ & * & * \\ & - & * \end{pmatrix}.$$

In summary, $\mathbf{A}^{(4)} = \mathbf{G}_3^{(4)^T}\mathbf{G}_3^{(2)^T}\mathbf{G}_3^{(1)^T}\mathbf{A}^{(0)}\mathbf{G}_3^{(3)}$. The Givens rotations need to be absorbed by $\mathbf{U}$ and $\mathbf{V}$ accordingly during the process. The later three steps of this process is further called *Zerochasing*, which takes a matrix of form

$$\begin{pmatrix} * & * & * \\ * & * & * \\ & * & * \end{pmatrix}$$

and make it

$$\begin{pmatrix} * & * & \\ & * & * \\ & & * \end{pmatrix}.$$

We will be using it again in every implicit symmetric QR iteration (see Section 4.2). We summarize the algorithms for Zerochasing and upper bidiagonalization in Algorithm 5 and 6.

4.2 Implicit Symmetric QR SVD with Wilkinson Shift

Our algorithm follows [Golub and Van Loan 2012]. Starting from $\mathbf{U} = \mathbf{I}$ and $\mathbf{V} = \mathbf{I}$, we first perform the upper bidiagonalization described in Section 4.1 to matrix $\mathbf{A}$ with $\mathbf{U}$ and $\mathbf{V}$ also updated. We use $\mathbf{B}$ to denote the bidiagonal matrix, where we have $\mathbf{U}\mathbf{B}\mathbf{V}^T = \mathbf{A}$. The implicit QR iteration operates on $\mathbf{B}$ iteratively and update $\mathbf{U}$ and $\mathbf{V}$ on the fly. Denoting $\mathbf{B}$ with

$$\mathbf{B} = \begin{pmatrix} \alpha_1 & \beta_1 & \\ & \alpha_2 & \beta_2 \\ & & \alpha_3 \end{pmatrix},$$

the corresponding symmetric eigenproblem is on the matrix

$$\mathbf{T} = \mathbf{B}^T\mathbf{B} = \mathbf{B} = \begin{pmatrix} \alpha_1^2 & \alpha_1\beta_1 & \\ \alpha_1\beta_1 & \alpha_2^2 + \beta_1^2 & \alpha_2\beta_2 \\ & \alpha_2\beta_2 & \alpha_3^2 + \beta_2^2 \end{pmatrix}.$$

QR iteration seeks to eliminate the off-diagonal entries of $\mathbf{T}$. Equivalently, one or more values of $(\alpha_1, \beta_1, \alpha_2, \beta_2)$ will converge to something close to zero. We will show in Section 4.3 that once any of $(\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3)$ becomes smaller than a tolerance τ, we can terminate the QR iterations and degrade the problem to a 2×2 SVD. The termination tolerance τ is computed as a relative tolerance via

$$\tau = \max(\frac{1}{2}\|\mathbf{B}\|_F, 1)\eta$$

where we choose $\eta = 128\epsilon$ and ϵ is the floating point machine epsilon.

QR Factorization The QR Factorization of a symmetric tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ can be easily done using $n - 1$ Givens rotations with $\mathbf{Q}$ being a rotation matrix and $\mathbf{R}$ being upper triangular.

QR Iteration If $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric, $\mathbf{R}_0$ is orthogonal and $\mathbf{T}_0 = \mathbf{R}_0^T\mathbf{A}\mathbf{R}_0$, then the iteration

$$\mathbf{T}_{k-1} = \mathbf{Q}_k\mathbf{R}_k,$$
$$\mathbf{T}_k = \mathbf{R}_k\mathbf{Q}_k$$

implies $\mathbf{T}_k = (\mathbf{R}_0\mathbf{R}_1 \dots \mathbf{R}_k)^T\mathbf{A}(\mathbf{R}_0\mathbf{R}_1 \dots \mathbf{R}_k)$ is symmetric tridiagonal, and converges to a diagonal form [Trefethen and Bau III 1997; Golub and Van Loan 2012].

Implicit Q Theorem Given $\mathbf{A} \in \mathbb{R}^{n\times n}$ symmetric, $\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \mathbf{T}$, $\mathbf{V}^T\mathbf{A}\mathbf{V} = \mathbf{S}$, $\mathbf{Q}$ and $\mathbf{V}$ are orthogonal, $\mathbf{T}$ and $\mathbf{S}$ are symmetric tridiagonal. If $\mathbf{A}$ is unreduced (meaning it has non-zero sub-diagonal entries) and the first column of $\mathbf{Q}$ and $\mathbf{V}$ are equal ($\mathbf{q}_1 = \mathbf{v}_1$), then $\mathbf{q}_i = \pm\mathbf{v}_i$ and $|T_{ij}| = |S_{ij}|$ [Golub and Van Loan 2012].

Explicit Shifted QR Iteration If μ is a good approximate eigenvalue of $\mathbf{T}$, then $\mathbf{T}_{n,n-1}$ tends to become smaller after a shifted QR step:

$$\mathbf{T} - \mu\mathbf{I} = \mathbf{Q}\mathbf{R},$$
$$\mathbf{T}^{new} = \mathbf{R}\mathbf{Q} + \mu\mathbf{I} = \mathbf{Q}^T\mathbf{T}\mathbf{Q}$$

and $\mathbf{T}$ maintains a symmetric tridiagonal form [Golub and Van Loan 2012].

Wilkinson Shift A good choice of the shift μ is the eigenvalue of $\mathbf{T}$'s bottom right 2×2 block that is closer to T_{nn} [Golub and Van Loan 2012]. This shift gives average cubic convergence rate for reducing $T_{n,n-1}$ to zero. In the 3×3 case where

$$\mathbf{T} = \begin{pmatrix} a_1 & b_1 & \\ b_1 & a_2 & b_2 \\ & b_2 & a_3 \end{pmatrix},$$

the shift is given by $\mu = a_3 + d - sign(d)\sqrt{d^2 + b_2^2}$ where $d = (a_2 - a_3)/2$ and $sign(d) = \pm 1$ (choose 1 when $d = 0$).

Implicit Shifted QR Iteration The shifted QR iteration can be done without constructing $\mathbf{T} - \mu\mathbf{I}$ explicitly. Let's focus on the 3×3 case where we have

$$\mathbf{T} - \mu\mathbf{I} = \begin{pmatrix} a_1 - \mu & b_1 & \\ b_1 & a_2 - \mu & b_2 \\ & b_2 & a_3 - \mu \end{pmatrix}.$$

The QR decomposition $\mathbf{T} - \mu\mathbf{I} = \mathbf{QR}$ looks like

$$\begin{pmatrix} a_1 - \mu & b_1 & \\ b_1 & a_2 - \mu & b_2 \\ & b_2 & a_3 - \mu \end{pmatrix} = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \end{pmatrix} \begin{pmatrix} * & * & * \\ & * & * \\ & & * \end{pmatrix},$$

this implies $\mathbf{q}_1 = \gamma(a_1 - \mu, b_1, 0)^T$ for some normalization scale γ. If we construct a Givens rotation $\mathbf{G}^1 = \mathbf{G}_3(1, 2, x = a_1 - \mu, y = b1)$, then it follows $\mathbf{g}_1^1 = \omega(a_1 - \mu, b_1, 0)^T$ for some normalization scale ω. Therefore we know $\mathbf{g}_1^1 = \mathbf{q}_1$, i.e., $\mathbf{G}^1$ and $\mathbf{Q}$ has the same first column. If we further find $\mathbf{G}^2$ such that $\mathbf{Z} = \mathbf{G}^1\mathbf{G}^2$ has the same first column with $\mathbf{G}^1$ and $\mathbf{S} = \mathbf{Z}^T\mathbf{TZ}$ is symmetric tridiagonal, then by implicit Q Theorem, since $\mathbf{T}^{new} = \mathbf{Q}^T\mathbf{TQ}$ and $\mathbf{S} = \mathbf{Z}^T\mathbf{TZ}$, it follows $\mathbf{q}_i = \pm\mathbf{z}_i$ and $|T_{ij}^{new}| = |S_{ij}|$. Therefore, utilizing $\mathbf{G}^1$ and $\mathbf{G}^2$ accomplishes the same effect as an explicit shifted QR iteration step for updating $\mathbf{T}$.

Implicit Shifted QR in the SVD Case For SVD, we prefer operating on $\mathbf{B}$ directly to constructing $\mathbf{T}$. Applying $\mathbf{G}^1$ directly to $\mathbf{B}$ followed by Zerochasing $\mathbf{B}$ back to upper bidiagonal is equivalent to doing implicit QR on $\mathbf{T}$ [Golub and Van Loan 2012]. More specifically in our 3×3 case, after applying $\mathbf{G}_1$ as a column rotation to $\mathbf{B}$, the column rotation in the Zerochasing (i.e., $\mathbf{G}_3^2$ in Section 4.1) essentially is the $\mathbf{G}^2$ we want to find in the implicit QR for $\mathbf{T}$ with the property that $\mathbf{G}^1\mathbf{G}^2$ has the same first column with $\mathbf{Q}$. Therefore by operating on $\mathbf{B}$ directly, the implicit symmetric QR algorithm is correctly applied.

We summarize the implicit shifted QR SVD in Algorithm 7. The steps after exiting the loop is described in Section 4.3.

4.3 Postprocess and Sorting

If any α or β from Algorithm 7 becomes small, implicit QR iteration is terminated. Here we show how each case is degraded to a 2×2 easy problem.

4.3.1 Deflation Cases

Case 1: $|\beta_2| \leq \tau$. In this case

$$\mathbf{B} = \begin{pmatrix} * & * & \\ & * & \\ & & * \end{pmatrix},$$

we just need to compute the 2×2 SVD of the top left sub-matrix and assemble back to 3D with

$$\mathbf{B} = \begin{pmatrix} * & * & \\ * & * & \\ & & 1 \end{pmatrix} \begin{pmatrix} * & & \\ & * & \\ & & * \end{pmatrix} \begin{pmatrix} * & * & \\ * & * & \\ & & 1 \end{pmatrix}^T$$

Algorithm 7 Implicit Shifted QR SVD of 3×3 matrices

```
1:  procedure SVD3D($\mathbf{A}$)                         ▷ return $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$
2:      $\mathbf{B} \leftarrow \mathbf{A}$
3:      $\mathbf{U}, \mathbf{V} \leftarrow \mathbf{I}$
4:      $Bidiagonalize(\mathbf{U}, \mathbf{B}, \mathbf{V})$    ▷ $\mathbf{B}$ is now upper bidiagonal
5:      $(\alpha_1, \alpha_2, \alpha_3) \leftarrow (B_{11}, B_{22}, B_{33})$
6:      $(\beta_1, \beta_2) \leftarrow (B_{12}, B_{23})$
7:      $(\gamma_1, \gamma_2) \leftarrow (\alpha_1\beta_1, \alpha_2\beta_2)$
8:      $\eta \leftarrow 128\epsilon$
9:      $\tau \leftarrow \eta \max(0.5 \|\mathbf{B}\|_F, 1)$
10:     while $|\beta_2|, |\beta_1|, |\alpha_1|, |\alpha_2|, |\alpha_3| > \tau$ do
11:         $a_1 \leftarrow \alpha_2^2 + \beta_1^2$
12:         $a_2 \leftarrow \alpha_3^2 + \beta_2^2$
13:         $b_1 \leftarrow \gamma_2$
14:         $d \leftarrow (a_1 - a_2)/2$
15:         $\mu \leftarrow copysign(b_1^2/(|d| + \sqrt{d^2 + b_1^2}), d)$
16:         $\mathbf{G} \leftarrow \mathbf{G}_3(1, 2, \alpha_1^2 - \mu, \gamma_1)$
17:         $\mathbf{B} \leftarrow \mathbf{BG}$
18:         $\mathbf{V} \leftarrow \mathbf{VG}$
19:         $Zerochasing(\mathbf{U}, \mathbf{B}, \mathbf{V})$
20:         $(\alpha_1, \alpha_2, \alpha_3) \leftarrow (B_{11}, B_{22}, B_{33})$
21:         $(\beta_1, \beta_2) \leftarrow (B_{12}, B_{23})$
22:         $(\gamma_1, \gamma_2) \leftarrow (\alpha_1\beta_1, \alpha_2\beta_2)$
23:     $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}) \leftarrow Postprocess(\mathbf{B}, \mathbf{U}, \mathbf{V}, \alpha, \beta)$ ▷ section 4.3
24:     return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$   ▷ $\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
```

Case 2: $|\beta_1| \leq \tau$. In this case

$$\mathbf{B} = \begin{pmatrix} * & & \\ & * & * \\ & & * \end{pmatrix},$$

we just need to compute the 2×2 SVD of the bottom right sub-matrix and assemble back to 3D.

Case 3: $|\alpha_2| \leq \tau$. In this case

$$\mathbf{B} = \begin{pmatrix} * & * & \\ & & * \\ & & * \end{pmatrix},$$

performing an *unconventional* Givens rotation $\hat{\mathbf{G}} = \hat{\mathbf{G}}_3(2, 3, x = B_{23}, y = B_{33})$ with $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T\mathbf{B}$ reduces $\mathbf{B}$ to the form

$$\mathbf{B} = \begin{pmatrix} * & * & \\ & & \\ & & * \end{pmatrix},$$

where we just need to compute the 2×2 SVD of the top left sub-matrix and assemble back to 3D.

Case 4: $|\alpha_3| \leq \tau$. In this case

$$\mathbf{B} = \begin{pmatrix} * & * & \\ & * & * \\ & & \end{pmatrix}.$$

We can use $\mathbf{G} = \mathbf{G}_3(2, 3, x = B_{22}, y = B_{23})$ with $\mathbf{B} \leftarrow \mathbf{BG}$ to reduce $\mathbf{B}$ to the form

$$\mathbf{B} = \begin{pmatrix} * & * & + \\ & * & - \\ & & \end{pmatrix},$$

Algorithm 8 Postprocessing after QR Iterations

```
1: procedure POSTPROCESS($(\mathbf{B}, \mathbf{U}, \mathbf{V}, \alpha, \beta)$) ▷ return $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$
2:     if $|\beta_2| \leq \tau$ then
3:         $SolveReducedTopLeft(\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
4:         $SortWithTopLeftSub(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
5:     else if $|\beta_1| \leq \tau$ then
6:         $SolveReducedBotRight(\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
7:         $SortWithBotRightSub(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
8:     else if $|\alpha_2| \leq \tau$ then
9:         $\hat{\mathbf{G}} \leftarrow \hat{\mathbf{G}}_3(2, 3, x = B_{23}, y = B_{33})$
10:        $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T \mathbf{B}$
11:        $\mathbf{U} \leftarrow \mathbf{U}\hat{\mathbf{G}}$
12:        $SolveReducedTopLeft(\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
13:        $SortWithTopLeftSub(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
14:    else if $|\alpha_3| \leq \tau$ then
15:        $\mathbf{G} \leftarrow \mathbf{G}_3(2, 3, x = B_{22}, y = B_{23})$
16:        $\mathbf{B} \leftarrow \mathbf{B}\mathbf{G}$
17:        $\mathbf{V} \leftarrow \mathbf{V}\mathbf{G}$
18:        $\mathbf{G} \leftarrow \mathbf{G}_3(1, 3, x = B_{11}, y = B_{13})$
19:        $\mathbf{B} \leftarrow \mathbf{B}\mathbf{G}$
20:        $\mathbf{V} \leftarrow \mathbf{V}\mathbf{G}$
21:        $SolveReducedTopLeft(\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
22:        $SortWithTopLeftSub(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
23:    else if $|\alpha_1| \leq \tau$ then
24:        $\hat{\mathbf{G}} \leftarrow \hat{\mathbf{G}}_3(1, 2, x = B_{12}, y = B_{22})$
25:        $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T \mathbf{B}$
26:        $\mathbf{U} \leftarrow \mathbf{U}\hat{\mathbf{G}}$
27:        $\hat{\mathbf{G}} \leftarrow \hat{\mathbf{G}}_3(1, 3, x = B_{13}, y = B_{33})$
28:        $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T \mathbf{B}$
29:        $\mathbf{U} \leftarrow \mathbf{U}\hat{\mathbf{G}}$
30:        $SolveReducedBotRight(\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
31:        $SortWithBotRightSub(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$
32:    return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$   ▷ $\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
```

followed by $\mathbf{G} = \mathbf{G}_3(1, 3, x = B_{11}, y = B_{13})$ with $\mathbf{B} \leftarrow \mathbf{B}\mathbf{G}$ to further reduce to

$$\mathbf{B} = \begin{pmatrix} * & * & - \\ + & * & \\ & & \end{pmatrix},$$

where we just need to compute the 2×2 SVD of the top left sub-matrix and assemble back to 3D.

Case 5: $|\alpha_1| \leq \tau$**.** In this case

$$\mathbf{B} = \begin{pmatrix} & * & \\ & * & * \\ & & * \end{pmatrix}.$$

Performing an *unconventional* Givens rotation $\hat{\mathbf{G}} = \hat{\mathbf{G}}_3(1, 2, x = B_{12}, y = B_{22})$ with $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T \mathbf{B}$ reduces $\mathbf{B}$ to the form

$$\mathbf{B} = \begin{pmatrix} & - & + \\ & * & * \\ & & * \end{pmatrix}.$$

Further performing an *unconventional* Givens rotation $\hat{\mathbf{G}} = \hat{\mathbf{G}}_3(1, 3, x = B_{13}, y = B_{33})$ with $\mathbf{B} \leftarrow \hat{\mathbf{G}}^T \mathbf{B}$ reduces $\mathbf{B}$ to the form

$$\mathbf{B} = \begin{pmatrix} & & - \\ & * & * \\ & + & * \end{pmatrix},$$

where we just need to compute the 2×2 SVD of the bottom right sub-matrix and assemble back to 3D.

Algorithm 9 Solve Reduced SVD Problem

```
1: procedure SOLVEREDUCEDTOPLEFT($\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$)
2:     $\sigma_3 \leftarrow B_{33}$
3:     $\mathbf{u} \leftarrow \mathbf{G}_2(1, 2)$
4:     $\mathbf{v} \leftarrow \mathbf{G}_2(1, 2)$
5:     $(\mathbf{u}, \sigma_1, \sigma_2, \mathbf{v}) = Svd2D((TopLeftBlock(\mathbf{B}))$
6:     $\mathbf{u} \leftarrow \mathbf{G}_3(1, 2, c = \mathbf{u}.c, s = \mathbf{u}.s)$
7:     $\mathbf{v} \leftarrow \mathbf{G}_3(1, 2, c = \mathbf{v}.c, s = \mathbf{v}.s)$
8:     $\mathbf{U} \leftarrow \mathbf{U}\mathbf{u}$
9:     $\mathbf{V} \leftarrow \mathbf{V}\mathbf{v}$
10:    return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$   ▷ $\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_1 \geq |\sigma_2|$
11: procedure SOLVEREDUCEDBOTRIGHT($\mathbf{B}, \mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$)
12:    $\sigma_1 \leftarrow B_{11}$
13:    $\mathbf{u} \leftarrow \mathbf{G}_2(1, 2)$
14:    $\mathbf{v} \leftarrow \mathbf{G}_2(1, 2)$
15:    $(\mathbf{u}, \sigma_2, \sigma_3, \mathbf{v}) = Svd2D((BotRightBlock(\mathbf{B}))$
16:    $\mathbf{u} \leftarrow \mathbf{G}_3(2, 3, c = \mathbf{u}.c, s = \mathbf{u}.s)$
17:    $\mathbf{v} \leftarrow \mathbf{G}_3(2, 3, c = \mathbf{v}.c, s = \mathbf{v}.s)$
18:    $\mathbf{U} \leftarrow \mathbf{U}\mathbf{u}$
19:    $\mathbf{V} \leftarrow \mathbf{V}\mathbf{v}$
20:    return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$   ▷ $\mathbf{U}, \mathbf{V}$ are rotations, $\sigma_2 \geq |\sigma_3|$
```

4.3.2 Sorting

After processing the reduced 2×2 cases, the full $\mathbf{\Sigma}$ needs to be carefully sorted to obey our sign convention. We give out the pseudocode for the full postprocessing (including sorting) in Algorithm 8, 9, and 10.

5 Performance

Our algorithm is implemented on top of the *Eigen* C++ template library for its automatic vectorization of certain matrix operations. The fast inverse square root function is the only explicit dependency of SSE intrinsics, which is pretty standard on modern CPUs. We test our code on a 12-core 3.47GHz Intel Xeon X5690 server using the g++ compiler for Linux, version 5.3.0. We use the following data set as our test cases:

1. Totally $1024 \times 1024 = 1,048,576$ random real matrices with each entry uniformly sampled from -3.0 to 3.0.
2. All integer entry matrices with each entry ranging from -2 to 2 (totally $5^9 = 1,953,125$ matrices).
3. Starting from the integer entry matrices, we perturb all entries with random numbers from -256ϵ to 256ϵ (where ϵ is the floating point machine precision). We generate 4 perturbed cases for each integer matrix, resulting in $4 \times 5^9 = 7,812,500$ perturbed integer matrices.
4. We perturb the all entries of an identity matrix with random numbers from -256ϵ to 256ϵ. Totally $1024 \times 1024 = 1,048,576$ matrices are generated.
5. We perturb the all entries of an identity matrix with random numbers from -0.001 to 0.001. Totally $1024 \times 1024 = 1,048,576$ matrices are generated.

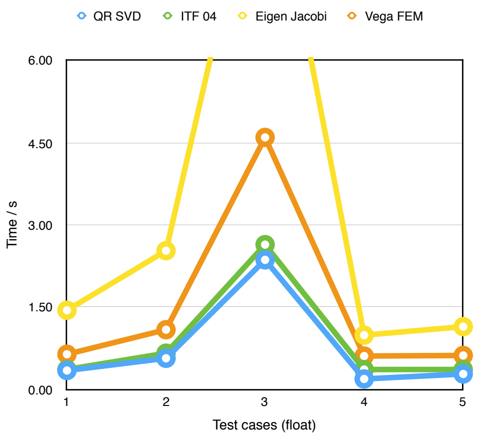
The random integer matrix and perturbed integer test cases show the performance on singular and nearly singular matrices. Matrices close to the identity are frequently obtained in simulations, particularly when warm starts are used. We run all tests with float and double precisions, and compare the results with the SVD described in [Irving et al. 2004] (implemented in the *PhysBAM* library and imported to our *Eigen* code base), Jacobi SVD (built in *Eigen*), and the one in *Vega FEM* library [Barbič et al. 2012] (See Figure 1 and 2). We don't directly compare with [McAdams et al. 2011] because their implementation is explicitly tailored to the characteristics of

Timing (float)

	QR SVD	ITF 04	Eigen Jacobi	Vega FEM
1	0.3438	0.3637	1.4422	0.6401
2	0.5669	0.6597	2.5292	1.0886
3	2.3622	2.6397	10.3638	4.5957
4	0.1899	0.3619	0.9887	0.6079
5	0.2806	0.3619	1.1439	0.6182

Timing (double)

	QR SVD	ITF 04	Eigen Jacobi	Vega FEM
1	0.6445	0.3517	1.8252	0.6367
2	1.0335	0.6354	3.1482	1.0834
3	4.4722	2.5336	12.9572	4.4881
4	0.2952	0.3481	1.0295	0.4800
5	0.6123	0.3481	1.6234	0.6141



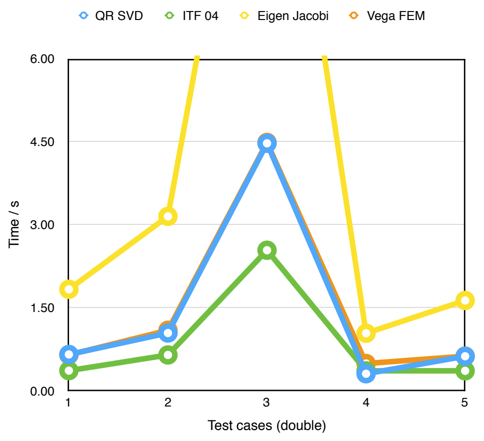


Figure 1: *Timing comparisons on all 5 tests with float and double precisions.*

Reconstruction Maximum Error (float)

	QR SVD	ITF 04	Eigen Jacobi	Vega FEM
1	4.965E-05	7.153E-07	7.153E-06	7.153E-07
2	3.123E-05	1.968E-04	4.530E-06	4.768E-07
3	4.035E-05	2.896E-04	5.007E-06	1.986E-06
4	1.542E-05	2.384E-07	1.609E-06	2.384E-07
5	1.528E-05	2.384E-07	1.907E-06	2.384E-07

Reconstruction Maximum Error (double)

	QR SVD	ITF 04	Eigen Jacobi	Vega FEM
1	8.971E-14	8.984E-10	1.332E-14	3.613E-10
2	5.351E-14	1.968E-04	8.438E-15	6.322E-08
3	7.471E-14	3.189E-04	1.021E-14	8.099E-08
4	2.850E-14	3.520E-15	2.887E-15	2.442E-15
5	2.820E-14	3.113E-14	4.219E-15	2.665E-15

Figure 2: *Maximum reconstruction error is computed as the maximum absolute value of entries in* $\mathbf{U\Sigma V}^T - \mathbf{A}$.

SIMD or vector processors, and likely runs faster for a large array of 3×3 matrices in parallel.

We observe that implicit QR SVD has consistent accuracy for different test cases. The accuracy can be adjusted further by changing the QR iteration tolerance and reach the machine precision. The algorithm of [Irving et al. 2004] becomes less accurate in certain degenerate cases even with double precision. This is largely due to the loss of information form constructing $\mathbf{A}^T\mathbf{A}$ explicitly.

For the tests we performed, implicit QR SVD is the fastest in floats, and comparable to [Barbič et al. 2012] in doubles. Notably, the Jacobi SVD built in *Eigen* is much slower than all other algorithms. This is partially due to their implementation is for general matrices with arbitrary dimension. A specifically designed algorithm such as in [McAdams et al. 2011] will probably improve it.

In summary, the Implicit QR SVD described in this document provides a nice balance between speed and accuracy. We release our C++ code together with this document and expect it to benefit many applications in computer graphics and computational solid mechanics.

Acknowledgements

The authors were partially supported by NSF CCF-1422795, ONR (N000141110719, N000141210834), DOD (W81XWH-15-1-0147), Intel STC-Visual Computing Grant (20112360) as well as a gift from Disney Research.

References

BARBIČ, J., SIN, F. S., AND SCHROEDER, D., 2012. Vega FEM Library. http://www.jernejbarbic.com/vega.

BOUAZIZ, S., MARTIN, S., LIU, T., KAVAN, L., AND PAULY, M. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans Graph 33*, 4, 154:1–154:11.

GAST, T., SCHROEDER, C., STOMAKHIN, A., JIANG, C., AND TERAN, J. 2015. Optimization integrator for large time steps. *IEEE Trans Vis Comp Graph 21*, 10, 1103–1115.

GOLUB, G. H., AND VAN LOAN, C. F. 2012. *Matrix computations*, vol. 3. JHU Press.

HEGEMANN, J., JIANG, C., SCHROEDER, C., AND TERAN, J. M. 2013. A level set method for ductile fracture. In *Proc ACM SIGGRAPH/Eurograp Symp Comp Anim*, 193–201.

IRVING, G., TERAN, J., AND FEDKIW, R. 2004. Invertible finite elements for robust simulation of large deformation. In *Proc ACM SIGGRAPH/Eurograph Symp Comp Anim*, 131–140.

KLAR, G., GAST, T., PRADHANA, A., FU, C., SCHROEDER, C., JIANG, C., AND TERAN, J. 2016. Drucker-prager elastoplasticity for sand animation. *ACM Trans Graph 35*, 4 (July).

MCADAMS, A., SELLE, A., TAMSTORF, R., TERAN, J., AND SIFAKIS, E. 2011. Computing the singular value decomposition of 3× 3 matrices with minimal branching and elementary floating point operations. Tech. rep., University of Wisconsin-Madison.

SAITO, S., ZHOU, Z.-Y., AND KAVAN, L. 2015. Computational bodybuilding: Anatomically-based modeling of human bodies. *ACM Trans Graph 34*, 4.

SIN, F., ZHU, Y., LI, Y., SCHROEDER, D., AND BARBIC, J. 2011. Invertible isotropic hyperelasticity using svd gradients. In *Proc ACM SIGGRAPH/Eurograph Symp Comp Anim (Posters)*, vol. 2.

Algorithm 10 Sorting Singular Values

```
1:  procedure SortWithTopLeftSub($\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$)    ▷ $\sigma_1 \geq |\sigma_2|$
2:      if $|\sigma_2| \geq |\sigma_3|$ then
3:          if $\sigma_2 < 0$ then
4:              $FlipSign(2, \mathbf{U}, \mathbf{\Sigma})$    ▷ sign of $\sigma_2$ and col 2 of $\mathbf{U}$
5:              $FlipSign(3, \mathbf{U}, \mathbf{\Sigma})$    ▷ sign of $\sigma_3$ and col 3 of $\mathbf{U}$
6:          return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$    ▷ $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
7:      if $\sigma_3 < 0$ then
8:          $FlipSign(2, \mathbf{U}, \mathbf{\Sigma})$
9:          $FlipSign(3, \mathbf{U}, \mathbf{\Sigma})$
10:     $swap(\sigma_2, \sigma_3)$
11:     $swap(\mathbf{U}.col(2), \mathbf{U}.col(3))$
12:     $swap(\mathbf{V}.col(2), \mathbf{V}.col(3))$
13:     if $\sigma_2 > \sigma_1$ then
14:         $swap(\sigma_1, \sigma_2)$
15:         $swap(\mathbf{U}.col(1), \mathbf{U}.col(2))$
16:         $swap(\mathbf{V}.col(1), \mathbf{V}.col(2))$
17:     else
18:         $\mathbf{U}.col(3) \leftarrow -\mathbf{U}.col(3)$
19:         $\mathbf{V}.col(3) \leftarrow -\mathbf{V}.col(3)$
20:     return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$    ▷ $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
21: procedure SortWithBotRightSub($\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$)    ▷ $\sigma_2 \geq |\sigma_3|$
22:     if $|\sigma_1| \geq |\sigma_2|$ then
23:         if $\sigma_1 < 0$ then
24:             $FlipSign(1, \mathbf{U}, \mathbf{\Sigma})$
25:             $FlipSign(3, \mathbf{U}, \mathbf{\Sigma})$
26:         return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$    ▷ $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
27:     $swap(\sigma_1, \sigma_2)$
28:     $swap(\mathbf{U}.col(1), \mathbf{U}.col(2))$
29:     $swap(\mathbf{V}.col(1), \mathbf{V}.col(2))$
30:     if $|\sigma_2| < |\sigma_3|$ then
31:         $swap(\sigma_2, \sigma_3)$
32:         $swap(\mathbf{U}.col(2), \mathbf{U}.col(3))$
33:         $swap(\mathbf{V}.col(2), \mathbf{V}.col(3))$
34:     else
35:         $\mathbf{U}.col(2) \leftarrow -\mathbf{U}.col(2)$
36:         $\mathbf{V}.col(2) \leftarrow -\mathbf{V}.col(2)$
37:     if $\sigma_2 < 0$ then
38:         $FlipSign(2, \mathbf{U}, \mathbf{\Sigma})$
39:         $FlipSign(3, \mathbf{U}, \mathbf{\Sigma})$
40:     return $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$    ▷ $\sigma_1 \geq \sigma_2 \geq |\sigma_3|$
```

STOMAKHIN, A., HOWES, R., SCHROEDER, C., AND TERAN, J. 2012. Energetically consistent invertible elasticity. In *Proc Symp Comp Anim*, 25–32.

STOMAKHIN, A., SCHROEDER, C., CHAI, L., TERAN, J., AND SELLE, A. 2013. A material point method for snow simulation. *ACM Trans Graph 32*, 4, 102:1–102:10.

STOMAKHIN, A., SCHROEDER, C., JIANG, C., CHAI, L., TERAN, J., AND SELLE, A. 2014. Augmented MPM for phase-change and varied materials. *ACM Trans Graph 33*, 4, 138:1–138:11.

TREFETHEN, L. N., AND BAU III, D. 1997. *Numerical linear algebra*, vol. 50. Siam.

XU, H., SIN, F., ZHU, Y., AND BARBIČ, J. 2015. Nonlinear material design using principal stretches. *ACM Trans Graph 34*, 4.