Supplemental Document

ROMAN POYA, Siemens Digital Industries Software, United Kingdom
ROGELIO ORTIGOSA, Technical University of Cartagena, Spain
THEODORE KIM, Yale University, United States

ACM Reference Format:
Roman Poya, Rogelio Ortigosa, and Theodore Kim. 2023. Supplemental Document. *ACM Trans. Graph.* 42, 3, Article 29 (April 2023), 11 pages. https://doi.org/10.1145/3585003

1 THE TENSOR CROSS PRODUCT APPROACH

In this section, we expand on the notion of tensor cross product as a precursor to evaluating energy gradients and Hessians. We describe their explicit matrix form and provide in-line code.

In the main article, the notion of *tensor cross product* $\times$ was introduced. Specifically, the cofactor tensor $\boldsymbol{H}$ of the deformation gradient tensor $\boldsymbol{F}$ was defined as:

$$\boldsymbol{H} = J\boldsymbol{F}^{-T} = \frac{1}{2}\boldsymbol{F}\times\boldsymbol{F}.$$

This product turns the inverse of $\boldsymbol{F}$ into a standard product, which is extremely powerful when deriving gradients and Hessians. The cofactor tensor $\boldsymbol{H}$ also corresponds to $\frac{\partial J}{\partial \boldsymbol{F}}$. Many physics simulations have relied on nested vector cross products to implement the derivatives of J [Jeong et al. 2009; Kraus et al. 2019; Neff 2006; Schröder et al. 2011; Smith et al. 2018, 2019]. However, it was first de Boer [1982] who formalised the notation and Bonet et al. [2015, 2016] who introduced its associated algebra. The tensor cross product between two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ can be written as:

$$[\boldsymbol{A}\times\boldsymbol{B}] = \begin{bmatrix} [\boldsymbol{A}\times\boldsymbol{B}]_{xx} & [\boldsymbol{A}\times\boldsymbol{B}]_{xy} & [\boldsymbol{A}\times\boldsymbol{B}]_{xz} \\ [\boldsymbol{A}\times\boldsymbol{B}]_{yx} & [\boldsymbol{A}\times\boldsymbol{B}]_{yy} & [\boldsymbol{A}\times\boldsymbol{B}]_{yz} \\ [\boldsymbol{A}\times\boldsymbol{B}]_{zx} & [\boldsymbol{A}\times\boldsymbol{B}]_{zy} & [\boldsymbol{A}\times\boldsymbol{B}]_{zz} \end{bmatrix}$$

$$\begin{aligned}
[\boldsymbol{A}\times\boldsymbol{B}]_{xx} &= A_{yy}B_{zz} - A_{yz}B_{zy} + A_{zz}B_{yy} - A_{zy}B_{yz} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{xy} &= A_{yz}B_{zx} - A_{yx}B_{zz} + A_{zx}B_{yz} - A_{zz}B_{yx} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{xz} &= A_{yx}B_{zy} - A_{yy}B_{zx} + A_{zy}B_{yx} - A_{zx}B_{yy} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{yx} &= A_{xz}B_{zy} - A_{xy}B_{zz} + A_{zy}B_{xz} - A_{zz}B_{xy} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{yy} &= A_{zz}B_{xx} - A_{zx}B_{xz} + A_{xx}B_{zz} - A_{xz}B_{zx} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{yz} &= A_{zx}B_{xy} - A_{zy}B_{xx} + A_{xy}B_{zx} - A_{xx}B_{zy} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{zx} &= A_{xy}B_{yz} - A_{xz}B_{yy} + A_{yz}B_{xy} - A_{yy}B_{xz} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{zy} &= A_{xz}B_{yx} - A_{xx}B_{yz} + A_{yx}B_{xz} - A_{yz}B_{xx} \\
[\boldsymbol{A}\times\boldsymbol{B}]_{zz} &= A_{xx}B_{yy} - A_{xy}B_{yx} + A_{yy}B_{xx} - A_{yx}B_{xy}
\end{aligned}$$

which is essentially a standard vector cross product between alternating columns. In addition, the tensor cross product admits the following properties

$$\begin{aligned}
\boldsymbol{A}\times\boldsymbol{B} &= \boldsymbol{B}\times\boldsymbol{A} \\
(\boldsymbol{A}\times\boldsymbol{B}) : \boldsymbol{C} &= (\boldsymbol{A}\times\boldsymbol{C}) : \boldsymbol{B} = (\boldsymbol{B}\times\boldsymbol{C}) : \boldsymbol{A} \\
\boldsymbol{A}\times\boldsymbol{I} &= (\mathrm{tr}\boldsymbol{A})\,\boldsymbol{I} - \boldsymbol{A}^T \\
(\boldsymbol{A}\times\boldsymbol{A}) : \boldsymbol{A} &= 6\,\det\boldsymbol{A} \\
\mathrm{cof}\boldsymbol{A} &= \frac{1}{2}\boldsymbol{A}\times\boldsymbol{A}
\end{aligned}$$

A detailed overview of the properties is in Bonet et al. [2015, 2016]. This product works like any other operator, is commutative, and can be extended to higher order tensors. It is particularly useful for computing the derivatives when inverses are present. In Listing 1 we give a Python script to perform the tensor cross product between two 3×3 matrices.

Listing 1. Python code for tensor cross product of two matrices

```
from numpy import zeros

def cross(A, B):
    C = zeros((3,3))
    C[0,0] = A[1,1]*B[2,2] - A[1,2]*B[2,1] - A[2,1]*B[1,2] + A[2,2]*B[1,1]
    C[0,1] = A[1,2]*B[2,0] - A[1,0]*B[2,2] + A[2,0]*B[1,2] - A[2,2]*B[1,0]
    C[0,2] = A[1,0]*B[2,1] - A[1,1]*B[2,0] - A[2,0]*B[1,1] + A[2,1]*B[1,0]
    C[1,0] = A[0,2]*B[2,1] - A[0,1]*B[2,2] + A[2,1]*B[0,2] - A[2,2]*B[0,1]
    C[1,1] = A[0,0]*B[2,2] - A[0,2]*B[2,0] - A[2,0]*B[0,2] + A[2,2]*B[0,0]
    C[1,2] = A[0,1]*B[2,0] - A[0,0]*B[2,1] + A[2,0]*B[0,1] - A[2,1]*B[0,0]
    C[2,0] = A[0,1]*B[1,2] - A[0,2]*B[1,1] - A[1,1]*B[0,2] + A[1,2]*B[0,1]
    C[2,1] = A[0,2]*B[1,0] - A[0,0]*B[1,2] + A[1,0]*B[0,2] - A[1,2]*B[0,0]
    C[2,2] = A[0,0]*B[1,1] - A[0,1]*B[1,0] - A[1,0]*B[0,1] + A[1,1]*B[0,0]
    return C
```

With the above definition of $\boldsymbol{H}$ we can derive

$$\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}} = \frac{1}{2}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{F}}\times\boldsymbol{F} + \boldsymbol{F}\times\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{F}}\right)$$

Authors' addresses: Roman Poya, roman.poya@siemens.com, Siemens Digital Industries Software, Francis House, Cambridge, United Kingdom, CB2 1PH; Rogelio Ortigosa, rogelio.ortigosa@upct.es, Technical University of Cartagena, Cartagena, Campus Muralla del Mar, 30202, Spain; Theodore Kim, theodore.kim@yale.edu, Yale University, Richmond, Connecticutt, CT 06520, United States.

0730-0301/2023/4-ART29 $15.00
https://doi.org/10.1145/3585003

Now $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{F}}$ is the fourth order identity tensor $\mathbb{I}$ which implies that the derivative of $\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}}$ involves only alternating components of $\boldsymbol{F}$:

$$\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}} = \frac{1}{2}(\mathbb{I} \times \boldsymbol{F} + \boldsymbol{F} \times \mathbb{I}).$$

Given the identity tensor, the summation collapses into a single tensor cross product between a fourth and a second order tensor. In general, the tensor cross product between a fourth tensor $\mathbb{A}$ and second order tensor $\boldsymbol{B}$ is

$$[\mathbb{A} \times \boldsymbol{B}]_{ijIJ} = \mathcal{E}_{jpq}\mathcal{E}_{JPQ}\mathbb{A}_{iIpP}\boldsymbol{B}_{Qq}.$$

The tensor cross product between a second order tensor and the fourth order tensor is defined as

$$[\boldsymbol{B} \times \mathbb{A}]_{ijIJ} = \mathcal{E}_{ipq}\mathcal{E}_{IPQ}\mathbb{A}_{qQjJ}\boldsymbol{B}_{Pp}$$

where $\mathcal{E}$ is Levi-Civita tensor. The implementation of the tensor cross product between some select high order tensors is available in the C++ tensor algebra package Fastor [Poya et al. 2017]. In particular, the term $\mathbb{I} \times \boldsymbol{F}$ (and $\boldsymbol{F} \times \mathbb{I}$) reduces to Eqn. 2 which can then be unpacked to a 9×9 matrix.

$$\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}} = \mathbb{I} \times \boldsymbol{F} = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & F_{33} & -F_{23} \\ -F_{33} & 0 & F_{13} \\ F_{23} & -F_{13} & 0 \end{bmatrix} & \begin{bmatrix} 0 & -F_{32} & F_{22} \\ F_{32} & 0 & -F_{12} \\ -F_{22} & F_{12} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & -F_{33} & F_{23} \\ F_{33} & 0 & -F_{13} \\ -F_{23} & F_{13} & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & F_{31} & -F_{21} \\ -F_{31} & 0 & F_{11} \\ F_{21} & -F_{11} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & F_{32} & -F_{22} \\ -F_{32} & 0 & F_{12} \\ F_{22} & -F_{12} & 0 \end{bmatrix} & \begin{bmatrix} 0 & -F_{31} & F_{21} \\ F_{31} & 0 & -F_{11} \\ -F_{21} & F_{11} & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \tag{2}$$

The C++ code for computing $\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}}$ is given in Listing 2.

Listing 2. C++ Eigen code for computing $\mathbb{I} \times \boldsymbol{F}$

```
template<typename T>
Eigen::Matrix<T, 9, 9> ICrossF(const Eigen::Matrix<T, 3, 3>& F)
{
    const Real F11 = F(0,0);
    const Real F12 = F(0,1);
    const Real F13 = F(0,2);
    const Real F21 = F(1,0);
    const Real F22 = F(1,1);
    const Real F23 = F(1,2);
    const Real F31 = F(2,0);
    const Real F32 = F(2,1);
    const Real F33 = F(2,2);

    Eigen::Matrix<T, 9, 9> IxF = Eigen::Matrix<T, 9, 9>::Zero();
    IxF(1,1) =  F33;
    IxF(1,2) = -F23;
    IxF(1,3) = -F33;
    IxF(1,5) =  F13;
    IxF(1,6) =  F23;
    IxF(1,7) = -F13;
    IxF(2,1) = -F32;
    IxF(2,2) =  F22;
    IxF(2,3) =  F32;
    IxF(2,5) = -F12;
    IxF(2,6) = -F22;
    IxF(2,7) =  F12;
    IxF(3,1) = -F33;
    IxF(3,2) =  F23;
    IxF(3,3) =  F33;
    IxF(3,5) = -F13;
    IxF(3,6) = -F23;
    IxF(3,7) =  F13;
    IxF(5,1) =  F31;
    IxF(5,2) = -F21;
    IxF(5,3) = -F31;
    IxF(5,5) =  F11;
    IxF(5,6) =  F21;
    IxF(5,7) = -F11;
    IxF(6,1) =  F32;
    IxF(6,2) = -F22;
    IxF(6,3) = -F32;
    IxF(6,5) =  F12;
    IxF(6,6) =  F22;
    IxF(6,7) = -F12;
    IxF(7,1) = -F31;
    IxF(7,2) =  F21;
    IxF(7,3) =  F31;
    IxF(7,5) = -F11;
    IxF(7,6) = -F21;
    IxF(7,7) =  F11;

    return IxF;
}
```

The derivatives of J are then

$$\boldsymbol{G}_J = \frac{\partial J}{\partial \boldsymbol{F}} = J\boldsymbol{F}^{-T} = \boldsymbol{H}$$
$$\mathbb{H}_J = \frac{\partial^2 J}{\partial \boldsymbol{F}\partial \boldsymbol{F}} = \frac{\partial \boldsymbol{H}}{\partial \boldsymbol{F}} = \mathbb{I} \times \boldsymbol{F}$$

In 2D, simplified expressions exist for the derivatives of J

$$\boldsymbol{G}_J = \text{tr}(\boldsymbol{F})\boldsymbol{I} - \boldsymbol{F}^T = \begin{bmatrix} F_{22} & -F_{21} \\ -F_{12} & F_{11} \end{bmatrix}$$
$$\mathbb{H}_J = \delta_{ij}\delta_{kl} - \delta_{ik}\delta_{jl} = \begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}.$$

1.1 The extended stretch invariants

In the main article we introduced an extended set of invariants. In particular, the $I_F = \sum \sigma_i$ of Smith et al. [2019] has the following gradient and Hessians,

$$\boldsymbol{G}_{I_F} = \frac{\partial I_F}{\partial \boldsymbol{F}} = \boldsymbol{R} \tag{3}$$

$$\mathbb{H}_{I_F} = \bar{\lambda}_i \mathbb{T}_i = \begin{cases} \bar{\lambda}\boldsymbol{T} \otimes \boldsymbol{T} & \text{2D} \\ \sum_{i=1}^{3} \bar{\lambda}_i \boldsymbol{T}_i \otimes \boldsymbol{T}_i & \text{3D} \end{cases} \tag{4}$$

where the corresponding eigenvalues $\bar{\lambda}$ are

$$\bar{\lambda} = \frac{2}{\sigma_1 + \sigma_2} \quad \text{in 2D,}$$
$$\bar{\lambda}_1 = \frac{2}{\sigma_2 + \sigma_3}, \quad \bar{\lambda}_2 = \frac{2}{\sigma_1 + \sigma_3} \quad \bar{\lambda}_3 = \frac{2}{\sigma_1 + \sigma_2}, \quad \text{in 3D}$$

We further introduced the cofactor invariants I_H and II_H. Their derivatives are then

$$\begin{aligned}
\boldsymbol{G}_{I_H} &= \frac{\partial I_H}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}(\frac{1}{2}(I_F^2 - II_F)) = I_F \boldsymbol{R} - \boldsymbol{F} \\
\mathbb{H}_{I_H} &= \frac{\partial^2 I_H}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = \boldsymbol{R} \otimes \boldsymbol{R} + I_F \mathbb{H}_{I_F} - \mathbb{I} \\
\boldsymbol{G}_{II_H} &= \frac{\partial II_H}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}(I_H^2 - 2 I_F J) \\
&= 2(I_H \boldsymbol{G}_{I_H} - J\boldsymbol{R} - I_F \boldsymbol{H}) = 2\boldsymbol{H} \times \boldsymbol{F} \\
\mathbb{H}_{II_H} &= \frac{\partial^2 II_H}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = 2\Big(\boldsymbol{G}_{I_H} \otimes \boldsymbol{G}_{I_H} + \mathbb{H}_{I_H} - (\boldsymbol{H} \otimes \boldsymbol{R} + \boldsymbol{R} \otimes \boldsymbol{H}) \\
&\quad - J\mathbb{H}_{I_F} - I_F \mathbb{I} \times \boldsymbol{F}\Big) \\
&= 2\boldsymbol{F} \times \mathbb{I} \times \boldsymbol{F}
\end{aligned}$$

The II_H derivatives can be expressed concisely in terms of the tensor cross product. The implementation of the $\boldsymbol{F} \times \mathbb{I} \times \boldsymbol{F}$ term is given in Listing 9. The $\boldsymbol{H} \times \boldsymbol{F}$ term can already be computed using Listing 2.

1.2 Twist, flip and scaling tensors

We begin by writing the explicit form of twist, flip and scaling tensors. The explicit 2D twist tensor is given by

$$\boldsymbol{T} = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \boldsymbol{V}^T$$

and the 3D version is

$$\boldsymbol{T}_1 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \boldsymbol{V}^T,$$
$$\boldsymbol{T}_2 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T,$$
$$\boldsymbol{T}_3 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T.$$

The explicit 2D flip and scaling tensors are

$$\boldsymbol{L} = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \boldsymbol{V}^T, \quad \boldsymbol{D}_1 = \boldsymbol{U} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{V}^T, \quad \boldsymbol{D}_2 = \boldsymbol{U} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{V}^T$$

and in 3D they are

$$\boldsymbol{L}_1 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \boldsymbol{V}^T, \qquad \boldsymbol{D}_1 = \boldsymbol{U} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T$$
$$\boldsymbol{L}_2 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T, \qquad \boldsymbol{D}_2 = \boldsymbol{U} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T$$
$$\boldsymbol{L}_3 = \frac{1}{\sqrt{2}} \boldsymbol{U} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{V}^T, \qquad \boldsymbol{D}_3 = \boldsymbol{U} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{V}^T.$$

Regularised invariants

$$\begin{aligned}
I_{F_r} &= I_F + d\beta \\
II_{F_r} &= II_F + 2\beta I_F + d\beta^2 \\
I_{H_r} &= I_H + 2\beta I_F + 3\beta^2 \qquad \text{3D} \\
II_{H_r} &= II_H + 2\beta(I_F\, II_F - III_F) + 2\beta^2 I_F^2 + 4\beta^3 I_F + 3\beta^4 \qquad \text{3D} \\
J_r &= \begin{cases} J + \beta I_F + \beta^2 & \text{2D} \\ J + \beta I_H + \beta^2 I_F + \beta^3 & \text{3D} \end{cases}
\end{aligned}$$

Table 1. Regularised invariants in 2D and 3D

We do not column-wise "flatten" high order tensors straight away like Smith et al. [2019] but rather perform high order tensor computations using the optimised tensor algebra framework Fastor (https://github.com/romeric/Fastor) [Poya et al. 2017]. We flatten at the last moment prior to pre- and post-multiplying with basis function derivatives.

1.3 Gradient and Hessian of regularised invariants

The gradient and Hessian of regularised invariants can be obtained by re-writing them in terms of standard invariants. The regularisation parameter β is treated constant, and for clarity we re-list the regularised invariants in Table 1. The derivatives are listed in Table 2 for 2D and Table 3 for 3D.

1.3.1 *First regularised invariant.*

$$\begin{aligned}
\boldsymbol{G}_{I_{F_r}} &= \frac{\partial I_{F_r}}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}\Big(I_F + d\beta\Big) = \boldsymbol{R} \\
\mathbb{H}_{II_{F_r}} &= \frac{\partial^2 I_{F_r}}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = \mathbb{H}_{I_F}
\end{aligned}$$

1.3.2 *Second regularised invariant.*

$$\begin{aligned}
\boldsymbol{G}_{II_{F_r}} &= \frac{\partial II_{F_r}}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}\Big(II_F + 2\beta I_F + d\beta^2\Big) = 2(\boldsymbol{F} + \beta \boldsymbol{R}) \\
\mathbb{H}_{II_{F_r}} &= \frac{\partial^2 II_{F_r}}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = 2(\mathbb{I} + \beta \mathbb{H}_{I_F})
\end{aligned}$$

1.3.3 *Third regularised invariant.*

$$\begin{aligned}
\boldsymbol{G}_{I_{H_r}} &= \frac{\partial I_{H_r}}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}\Big(I_H + 2\beta I_F + 3\beta^2\Big) = \boldsymbol{G}_{I_H} + 2\beta \boldsymbol{R} \\
\mathbb{H}_{I_{H_r}} &= \frac{\partial^2 I_{H_r}}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = \mathbb{H}_{I_H} + 2\beta \mathbb{H}_{I_F}
\end{aligned}$$

1.3.4 Fourth regularised invariant. This invariant is quartic and to compute its gradient and Hessian, we first need the derivatives of

Gradient and Hessian of regularised invariants in 2D

$$
\begin{aligned}
\boldsymbol{G}_{I_{F_r}} &= \boldsymbol{R} \\
\mathbb{H}_{I_{F_r}} &= \mathbb{H}_{I_F} \\
\\
\boldsymbol{G}_{II_{F_r}} &= 2(\boldsymbol{F} + \beta \boldsymbol{R}) \\
\mathbb{H}_{II_{F_r}} &= 2(\mathbb{I} + \beta \mathbb{H}_{I_F}) \\
\\
\boldsymbol{G}_{J_r} &= \boldsymbol{H} + \beta \boldsymbol{R} \\
\mathbb{H}_{J_r} &= \mathbb{H}_J + \beta \mathbb{H}_{I_F}
\end{aligned}
$$

Table 2. Derivatives of regularised invariants in 2D. Terms in orange indicate contributions to gradients/Hessian stabilisers that emerge as a result of regularisation of the deformation gradient $\boldsymbol{F}$/singular-values

III_F. Using the following relationship,

$$
\begin{aligned}
III_F &= I_F^3 - 3(I_F I_H - J) \\
\boldsymbol{G}_a &= \frac{\partial III_F}{\partial \boldsymbol{F}} = 3(I_F^2 - I_H)\boldsymbol{R} - 3I_F \boldsymbol{G}_{I_H} + 3\boldsymbol{H} \\
\mathbb{H}_a &= \frac{\partial^2 III_F}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = 6I_F \boldsymbol{R} \otimes \boldsymbol{R} - 3\Big(\boldsymbol{G}_{I_H} \otimes \boldsymbol{R} - \boldsymbol{R} \otimes \boldsymbol{G}_{I_H}\Big) \\
&\quad + 3\left(I_F^2 - I_H\right)\mathbb{H}_{I_F} - 3I_F \mathbb{H}_{I_H} + 3\mathbb{I} \boldsymbol{\times} \boldsymbol{F}
\end{aligned}
$$

then we derive

$$
\begin{aligned}
\boldsymbol{G}_{II_{H_r}} &= \frac{\partial II_{H_r}}{\partial \boldsymbol{F}} \\
&= \frac{\partial}{\partial \boldsymbol{F}}\Big(II_H + 2\beta(I_F II_F - III_F) + 2\beta^2 I_F^2 + 4\beta^3 I_F + 3\beta^4\Big) \\
&= \boldsymbol{G}_{II_H} + 2\beta\left(2I_F \boldsymbol{F} - \boldsymbol{G}_a\right) + \left(2\beta II_F + 4\beta^2 I_F + 4\beta^3 I_F\right)\boldsymbol{R} \\
\mathbb{H}_{II_{H_r}} &= \frac{\partial^2 II_{H_r}}{\partial \boldsymbol{F} \partial \boldsymbol{F}} = \mathbb{H}_{II_H} - 2\beta \mathbb{H}_a + 4\beta\Big(\boldsymbol{R} \otimes \boldsymbol{F} + \boldsymbol{F} \otimes \boldsymbol{R}\Big) \\
&\quad + 4\beta\Big(I_F \mathbb{I} + II_F \mathbb{H}_R\Big) + 4\beta^2(1+\beta)\Big(\boldsymbol{R} \otimes \boldsymbol{R} + I_F \mathbb{H}_{I_F}\Big).
\end{aligned}
$$

1.3.5 *Fifth regularised invariant.*

$$
\begin{aligned}
\boldsymbol{G}_{J_r} &= \frac{\partial J_r}{\partial \boldsymbol{F}} = \frac{\partial}{\partial \boldsymbol{F}}\Big(J + \beta I_H + \beta^2 I_F + \beta^3\Big) \\
&= \boldsymbol{H} + \beta \boldsymbol{G}_{I_{H_r}} + \beta^2 \boldsymbol{R} \\
\mathbb{H}_{J_r} &= \mathbb{H}_J + \beta \mathbb{H}_{I_H} + \beta^2 \mathbb{H}_{I_F}
\end{aligned}
$$

Listing 3. vec function used in element kernel

```
//! Function to flatten a matrix in to a vector
template<typename T, int N>
Vector<T, N * N>
vec(const Matrix<T, N, N>& A)
{
    // This is to flatten row-wise and not column-wise
    Matrix<T, N, N> vecA = A.transpose();
    return Eigen::Map<const Vector<T, N * N>>(vecA.data());
}
```

Gradient and Hessian of regularised invariants in 3D

$$
\begin{aligned}
\boldsymbol{G}_{I_{F_r}} &= \boldsymbol{R} \\
\mathbb{H}_{I_{F_r}} &= \mathbb{H}_{I_F} \\
\\
\boldsymbol{G}_{II_{F_r}} &= 2(\boldsymbol{F} + \beta \boldsymbol{R}) \\
\mathbb{H}_{II_{F_r}} &= 2(\mathbb{I} + \beta \mathbb{H}_{I_F}) \\
\\
\boldsymbol{G}_{I_{H_r}} &= \boldsymbol{G}_{I_H} + 2\beta \boldsymbol{R} \\
\mathbb{H}_{I_{H_r}} &= \mathbb{H}_{I_H} + 2\beta \mathbb{H}_{I_F} \\
\\
\boldsymbol{G}_{II_{H_r}} &= \boldsymbol{G}_{II_H} \\
&\quad + 2\beta(2I_F \boldsymbol{F} - \boldsymbol{G}_a) + (2\beta II_F + 4\beta^2 I_F + 4\beta^3 I_F)\boldsymbol{R} \\
\mathbb{H}_{II_{H_r}} &= \mathbb{H}_{II_H} - 2\beta \mathbb{H}_a + 4\beta\Big(\boldsymbol{R} \otimes \boldsymbol{F} + \boldsymbol{F} \otimes \boldsymbol{R}\Big) \\
&\quad + 4\beta\Big(I_F \mathbb{I} + II_F \mathbb{H}_{I_F}\Big) + 4\beta^2(1+\beta)\Big(\boldsymbol{R} \otimes \boldsymbol{R} + I_F \mathbb{H}_{I_F}\Big) \\
\\
\boldsymbol{G}_{J_r} &= \boldsymbol{H} + \beta \boldsymbol{G}_{I_{H_r}} + \beta^2 \boldsymbol{R} \\
\mathbb{H}_{J_r} &= \mathbb{H}_J + \beta \mathbb{H}_{I_H} + \beta^2 \mathbb{H}_{I_F}
\end{aligned}
$$

Table 3. Derivatives of regularised invariants in 3D. Terms in orange indicate contributions to gradients/Hessian stabilisers that emerge as a result of regularisation of the deformation gradient $\boldsymbol{F}$/singular-values

Listing 4. outer function used in element kernel

```
//! Function for outer product of vectors
template<typename T, int N>
Matrix<T,N,N> outer(const Vector<T,N> &a, const Vector<T,N> &b)
{
    return a * b.transpose();
}
```

2 GRADIENT AND HESSIAN OF REGULARISED DISTORTION ENERGIES IN 3D

In the main article, the gradients and Hessians of RMIPS, RAMIPS, RSD and RSARAP were given in 2D. We now show the 3D case. Given the gradients and Hessians of the regularised invariants, we now have everything needed to compute the energy derivatives.

2.1 Regularised Advanced MIPS (RAMIPS)

The regularised version of AMIPS is simpler as the cofactor invariants disappear. We assume $f(J_r)$ to be a generic volumetric term such that without it, Regularised MIPS (RMIPS) is recovered. The

energy, gradient and Hessian can be written as:

$$\hat{\mathcal{D}}_{\text{RAMIPS}}(II_{F_r}, J_r) = \frac{1}{3} J_r^{-2/3} II_{F_r} + f(J_r)$$

$$\boldsymbol{P}_{\text{RAMIPS}} = \frac{1}{3}\left(J_r^{-2/3}\boldsymbol{G}_{II_{F_r}} - \frac{2}{3}J_r^{-5/3} II_{F_r}\boldsymbol{G}_{J_r}\right) + f'(J_r)\boldsymbol{G}_{J_r}$$

$$\begin{aligned}\mathbb{H}_{\text{RAMIPS}} = \frac{1}{3}\Big(&J_r^{-2/3}\mathbb{H}_{II_{F_r}} - \frac{2}{3}J_r^{-5/3}(\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{II_{F_r}} + \boldsymbol{G}_{II_{F_r}}\otimes\boldsymbol{G}_{J_r}) \\ &+ \frac{10}{9}J_r^{-8/3} II_{F_r}\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{J_r} - \frac{2}{3}J_r^{-5/3} II_{F_r}\mathbb{H}_{J_r}\Big) \\ &+ f''(J_r)\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{J_r} + f'(J_r)\mathbb{H}_{J_r}\end{aligned}$$

2.2 Regularised Symmetric Dirichlet (RSD)

The cofactor invariant II_H now appears, and the energy, gradient and Hessian can be written as

$$\hat{\mathcal{D}}_{\text{RSD}}(II_{F_r}, II_{H_r}, J_r) = \frac{1}{2}(II_{F_r} + J_r^{-2} II_{H_r})$$

$$\boldsymbol{P}_{\text{RSD}} = \frac{1}{2}\left(\boldsymbol{G}_{II_{F_r}} + J_r^{-2}\boldsymbol{G}_{II_{H_r}} - 2J_r^{-3} II_{H_r}\boldsymbol{G}_{J_r}\right)$$

$$\begin{aligned}\mathbb{H}_{\text{RSD}} = \frac{1}{2}\Big(&\mathbb{H}_{II_{F_r}} - 2J_r^{-3}(\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{II_{H_r}} + \boldsymbol{G}_{II_{H_r}}\otimes\boldsymbol{G}_{J_r}) \\ &- J_r^{-2}\mathbb{H}_{II_{H_r}} + 6J_r^{-4} II_{H_r}\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{J_r} - 2J_r^{-3} II_{H_r}\mathbb{H}_{J_r}\Big)\end{aligned}$$

2.3 Regularised Symmetric ARAP (RSARAP)

Both cofactor invariants I_H and II_H appear, and RSARAP can be considered as an enhanced version of RSD

$$\hat{\mathcal{D}}_{\text{RSARAP}}(II_{F_r}, II_{H_r}, J_r) = \frac{1}{2}(II_{F_r} + J_r^{-2} II_{H_r}) - (I_F + J_r^{-1} I_H)$$

$$\begin{aligned}\boldsymbol{P}_{\text{RSARAP}} = &\frac{1}{2}\left(\boldsymbol{G}_{II_{F_r}} + J_r^{-2}\boldsymbol{G}_{II_{H_r}} - 2J_r^{-3} II_{H_r}\boldsymbol{G}_{J_r}\right) \\ &- \left(\boldsymbol{G}_{I_F} - J_r^{-2} I_H\boldsymbol{G}_{J_r} + J_r^{-1}\boldsymbol{G}_{I_H}\right)\end{aligned}$$

$$\begin{aligned}\mathbb{H}_{\text{RSARAP}} = \frac{1}{2}\Big(&\mathbb{H}_{II_{F_r}} - 2J_r^{-3}(\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{II_{H_r}} + \boldsymbol{G}_{II_{H_r}}\otimes\boldsymbol{G}_{J_r}) \\ &- J_r^{-2}\mathbb{H}_{II_{H_r}} + 6J_r^{-4} II_{H_r}\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{J_r} - 2J_r^{-3} II_{H_r}\mathbb{H}_{J_r}\Big) \\ &- \Big(\mathbb{H}_{I_F} + 2J_r^{-3} I_H\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{J_r} - J_r^{-2}(\boldsymbol{G}_{J_r}\otimes\boldsymbol{G}_{I_H} + \boldsymbol{G}_{I_H}\otimes\boldsymbol{G}_{J_r}) \\ &+ J_r^{-1}\mathbb{H}_{I_H}\Big)\end{aligned}$$

2.4 Computing the regularisation parameter

2.4.1 Computing the worst Jacobian. Computing the regularisation parameter β was presented in the main article as pseudo-code. Here we provide a Python implementation. To compute β we first need to compute the worst Jacobian $J_{\min}$ over the entire domain (Listing 5).

Listing 5. Python code for computing worst J

```python
import numpy as np

def ComputeWorstJacobian(targetMesh : Mesh, sourceMesh : Mesh):
    """Computes worst Jacobian over the entire mesh"""

    minJ = 1e307
    for elementCounter in range(len(targetMesh.T)):
        # Lagrangian coordinate
        X = sourceMesh.GetElementCoordinates(elementCounter)
        # Eulerian coordinate
        x = targetMesh.GetElementCoordinates(elementCounter)
        # Compute the deformation gradient tensor
        F = ComputeDeformationGradient(X, x)
        J = np.linalg.det(F)
        minJ = min(J, minJ)
    return minJ
```

Then the auxiliary parameter δ is computed (Listing 6).

Listing 6. Python code for computing J regulariser δ

```python
from math import sqrt

def ComputeDelta(minJ : float, eps : float):
    """Computes Jacobian regulariser"""
    if minJ > 1e-4:
        return 0
    elif minJ > 0 and minJ < 1e-4:
        return eps
    else:
        return 0.5 * sqrt(eps**2 + 0.04 * minJ**2)
```

Finally β is computed using a root finding algorithm Listing 7.

Listing 7. Python code for computing $\boldsymbol{F}$ regulariser β

```python
import numpy as np
from math import sqrt

def ComputeBeta(targetMesh : Mesh, sourceMesh : Mesh, delta : float):
    """Computes maximum singular value regulariser"""

    maxBeta = -1e307
    for elementCounter in range(len(targetMesh.T)):
        # Lagrangian coordinate
        X = sourceMesh.GetElementCoordinates(elementCounter)
        # Eulerian coordinate
        x = targetMesh.GetElementCoordinates(elementCounter)
        # Compute the deformation gradient tensor
        F = ComputeDeformationGradient(X, x)
        # Compute signed singular value of F
        U, S, V = SVD(F)
        # Get the first invariant I_F
        I_F = S.sum()
        # Get second invariant II_F
        II_F = np.trace(F.dot(F.T))
        # Compute third invariant I_H
        I_H = 0.5 * (I_F**2 - II_F)
        # Compute fifth invariant J
        J = np.linalg.det(F)
        # Compute regularised Jacobian J_r
        J_r = 0.5 * sqrt(J**2 + delta**2)
        # Find beta: get the roots of cubic polynomial
        poly = [1, I_F, I_H, -(Jr - J)]
        roots = np.roots(poly)
        beta = np.real(roots[~np.iscomplex(roots)])
        maxBeta = max(beta, maxBeta)
    return maxBeta
```

We assume the mesh data structure from Listing 8 exists.

Listing 8. Mesh data structure

```
class Mesh(object):
    V = [] # num_vertices × d
    T = [] # num_elements × (d+1)
```

3 REGULARISED MIPS (RMIPS) WITHOUT β

We will now show a case where it is possible to discard β and formulate the problem as a Jacobian regulariser. If β is considered an invariant-dependent variable (e.g. re-substituted from $\beta = \frac{1}{2}(-I_F + \sqrt{I_F^2 + 4(J_r - J)})$), a similar procedure can be carried out. In this study will also show how our regularisation helps stabilise the Hessian.

The MIPS energy [Hormann and Greiner 2000], is one of the most popular fold-over preventing energies [Fu and Liu 2016; Garanzha et al. 2021]. It can be written in terms of our regularised invariants as $\frac{II_{F_r}}{d\, J_r^{2/d}}$, or further expanded in 2D to

$$\hat{\mathcal{D}}_{\text{RMIPS}}(I_F, II_F, J) = \frac{II_F + 2(\beta I_F + \beta^2)}{2(J + \beta I_F + \beta^2)} = \frac{II_F}{2J_r} + \left(1 - \frac{J}{J_r}\right). \quad (5)$$

We have re-substituted $J_r = J_r(I_F, J)$ to highlight that β disappears and a regulariser for J is sufficient. This modified, regularised energy remains consistent, and the gradient becomes

$$\boldsymbol{P}_{\text{RMIPS}} = \frac{1}{J_r}(\boldsymbol{F} - \boldsymbol{H}) - \frac{1}{J_r\, s}\left(J - \frac{II_F}{2}\right)\boldsymbol{H} \quad (6)$$

where $s = \sqrt{J^2 + \delta^2}$. Note that, the gradient vanishes in the origin, $\boldsymbol{P}_{\text{RMIPS}}|_{\boldsymbol{F}=\boldsymbol{I}} = \boldsymbol{0}$, which is not the case with approaches like Garanzha et al. [2021], because J_r no longer corresponds to $\det \boldsymbol{F}$. With those approaches, a closed-form eigensystem is impossible. In our case however, we can write the closed-form eigensystem

$$\lambda_1 = \frac{1}{J_r} + \frac{2(II_F J_r + 2s^2)(II_F - 2J) + \alpha}{4J_r s^3},$$
$$\lambda_2 = \frac{1}{J_r} + \frac{2(II_F J_r + 2s^2)(II_F - 2J) - \alpha}{4J_r s^3},$$
$$\lambda_3 = \frac{2}{J_r} + \frac{II_F - 2J}{2J_r s}, \qquad \lambda_4 = -\frac{II_F - 2J}{2J_r s},$$

where α is a positive quantity given as

$$\begin{aligned}
\alpha &= \sqrt{II_F^2 a + II_F b + c} \\
a &= 2(J^4 + 2J^3 s - 11J^2 s^2 - 12Js^3 + 18s^4) + 4J_r^2(II_F^2 - 2J^2) \\
b &= 8(-J^5 - 2J^4 s + 9J^3 s^2 + 8J^2 s^3 - 20Js^4 + 6s^5) \\
&\quad + 4(-J^3 - 2J^2 s + Js^2 + 2s^3)(II_F^2 - 2J^2) \\
c &= 8(J^6 + 2J^5 s - 7J^4 s^2 - 4J^3 s^3 + 18J^2 s^4 - 12Js^5 + 2s^6) \\
&\quad + 4(J^4 + 2J^3 s - 3J^2 s^2 - 4Js^3 + 4s^4)(II_F^2 - 2J^2).
\end{aligned}$$

Since, $II_F - 2J = (\sigma_1 - \sigma_2)^2$ is always positive, λ_1 and λ_3 are always positive. We plotted the variation of λ_2 and λ_4 for a fixed II_F in Fig. 1 and confirmed that our regularisation stabilises the Hessian. This effectively eliminates the numerical instabilities from [Shen et al. 2019], as all singularities are clamped to behave asymptotically. The corresponding eigenmatrices are the same for all 2D energies as shown earlier with the normaliser emerging as

$$\tau = \frac{2(II_F J_r - J^2 - Js + 2s^2)(\sigma_1^2 - \sigma_2^2) - \alpha}{2(II_F(3s^2 - Js - J^2) + (J^3 + J^2 s - 3Js^2 + s^3))}.$$

Here, we obtained the closed-form eigensystem for RMIPS, but in general it is not necessary to obtain λ_1 and λ_2 analytically, as they can be obtained by solving a small $d \times d$ eigenvalue problem.

4 BOUNDS OF POLYCONVEXITY OF SYMMETRIC DIRICHLET ENERGY

In Garanzha et al. [2021], polyconvexity of AMIPS was presented. AMIPS is a classical isochoric-volumetric split for nearly incompressible solids. It is commonly used in physics simulations, as its polyconvex nature is well known [Hartmann and Neff 2003; Schröder and Neff 2003]. Here, we will first determine the exact bounds for which the Symmetric Dirichlet energy loses polyconvexity. We will then show how our regularisation shifts the convexity bounds of this energy to the negative J region.

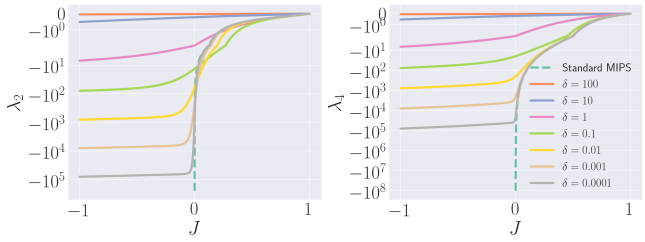

Fig. 1. Variation of λ_2 and λ_4 for MIPS and regularised MIPS (RMIPS) with fixed II_F and varying J. Our scheme projects the Hessian specially for large values of δ, but unlike in PN, the projection is *exact*. When the system goes indefinite, our regularisation adds significant stabilisation to the system. Further projecting the Hessian to SPD slightly damps Newton (in practice, local injectivity is recovered at relatively large δ) suggesting nearly quadratic convergence even for folded configurations.

Consider the 2D case $\hat{\mathcal{D}}_{\text{SD}}(II_F, J) = \mathcal{D}_{\text{SD}}(\boldsymbol{F}) = \frac{1}{2} II_{\boldsymbol{F}}(1 + J^{-2})$. We can obtain the second derivatives by starting from helper first directional derivatives

$$D\mathcal{D}_{\text{SD}}[\delta\boldsymbol{F}] = (1 + J^{-2})(\boldsymbol{F} : \delta\boldsymbol{F}), \qquad D\mathcal{D}_{\text{SD}}[\delta J] = -J^{-3} II_{\boldsymbol{F}}\, \delta J$$

and subsequently the second derivatives

$$\begin{aligned}
D^2\mathcal{D}_{\text{SD}}[\delta\boldsymbol{F}; \delta\boldsymbol{F}] &= (1 + J^{-2}) II_{\delta\boldsymbol{F}}, \\
D^2\mathcal{D}_{\text{SD}}[\delta\boldsymbol{F}; \delta J] &= -2J^{-3}\delta J(\boldsymbol{F} : \delta\boldsymbol{F}), \\
D^2\mathcal{D}_{\text{SD}}[\delta J; \delta J] &= 3J^{-4} II_{\boldsymbol{F}}\, \delta J^2
\end{aligned}$$

The determinant of $\mathbb{H}$ being non-negative can also be expressed in terms of the following characteristic equation Δ (note that polyconvexity implies positiveness of Δ)

$$\Delta = D^2\mathcal{D}_{\text{SD}}[\delta\boldsymbol{F}; \delta\boldsymbol{F}] + 2D^2\mathcal{D}_{\text{SD}}[\delta\boldsymbol{F}; \delta J] + D^2\mathcal{D}_{\text{SD}}[\delta J; \delta J]$$

which expands to

$$\Delta = J^{-4}\Big(\left(J^4 + J^2\right) II_{\delta\boldsymbol{F}} - 4J\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) + 3\delta J^2 II_{\boldsymbol{F}}\Big)$$

We can equivalently show positiveness of $\tilde{\Delta} := J^4\Delta$.

Case $\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) > 0$ and $J > 0$:

$$\begin{aligned}
\tilde{\Delta} &= \left(\sqrt{J^4 + J^2}\delta\boldsymbol{F} - \sqrt{3}\delta J\boldsymbol{F}\right) : \left(\sqrt{J^4 + J^2}\delta\boldsymbol{F} - \sqrt{3}\delta J\boldsymbol{F}\right) \\
&\quad + \left(2\sqrt{3}\sqrt{J^4 + J^2} - 4J\right)\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) \\
&\geq \left(2\sqrt{3}\sqrt{J^4 + J^2} - 4J\right)\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F})
\end{aligned}$$

Since $\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) > 0$, polyconvexity (i.e. positiveness of $\tilde{\Delta}$) is fulfilled if $f(J) := 2\sqrt{3}\sqrt{J^4 + J^2} - 4J$ (with $J > 0$) is guaranteed. It is trivial to see that this is the case if $J \in \left(1/\sqrt{3}, \infty\right)$.

Case $\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) < 0$ and $J < 0$:

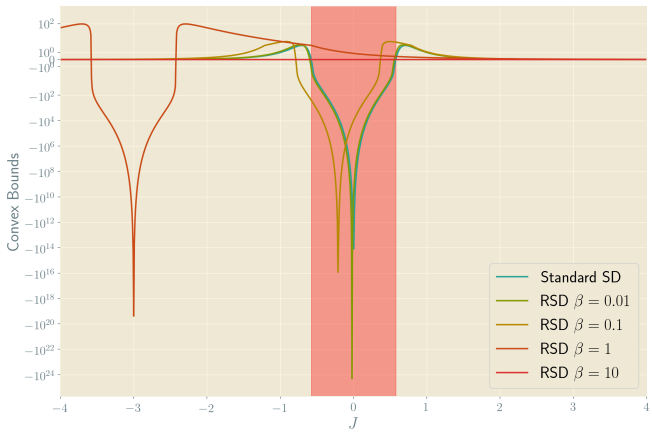

Fig. 2. Loss of polyconvexity for Symmetric Dirichlet (SD) and Regularised Symmetric Dirichlet (RSD) with different singular-value regularisation values i.e. β. Red bars indicate the region for which the Symmetric Dirichlet energy loses polyconvexity i.e. $J \in \left(-1/\sqrt{3}, 1/\sqrt{3}\right)$ whereas the region for which RSD loses polyconvexity is determined by value of β i.e. $J \in \left(-\beta(I_F+\beta) - 1/\sqrt{3}, -\beta(I_F+\beta) + 1/\sqrt{3}\right)$. With large values of β, $\boldsymbol{F}$ regularisation convexifies the energy for all negative values of J.

$$\begin{aligned}\tilde{\Delta} &= \left(\sqrt{J^4+J^2}\delta\boldsymbol{F} + \sqrt{3}\delta J\boldsymbol{F}\right) : \left(\sqrt{J^4+J^2}\delta\boldsymbol{F} + \sqrt{3}\delta J\boldsymbol{F}\right) \\ &\quad - \left(2\sqrt{3}\sqrt{J^4+J^2} + 4J\right)\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) \\ &\geq -\left(2\sqrt{3}\sqrt{J^4+J^2} + 4J\right)\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F})\end{aligned}$$

Since $\delta J\,(\boldsymbol{F} : \delta\boldsymbol{F}) < 0$, polyconvexity (i.e. positiveness of $\tilde{\Delta}$) is fulfilled if $f(J) := 2\sqrt{3}\sqrt{J^4+J^2} + 4J$ (with $J < 0$) is guaranteed. It is trivial to see that this is the case if $J \in \left(-\infty, -1/\sqrt{3}\right)$. Therefore, polyconvexity of $\mathcal{D}_{\text{SD}}$ is fulfilled provided that J belongs to the interval $J \in \left(-\infty, -1/\sqrt{3}\right) \cup \left(1/\sqrt{3}, \infty\right)$, and hence, the region for loss of strict polyconvexity is associated with the interval $J \in \left(-1/\sqrt{3}, 1/\sqrt{3}\right)$.

We have proven lack of polyconvexity for the Symmetric Dirichlet (SD) energy. Proving the same for the Regularised Symmetric Dirichlet (RSD) energy, we get bounds depending on the value of β. The region for loss of strict polyconvexity of RSD is in fact associated with the interval $J \in \left(-\beta(I_F+\beta) - 1/\sqrt{3}, -\beta(I_F+\beta) + 1/\sqrt{3}\right)$. We plot these bounds in Figure 2. As seen, our regularisation shifts the bounds of polyconvexity of SD to the negative J region and at $\beta = 10$ fully convexifies the energy. Shifting the bounds of polyconvexity to the negative J region is both good and bad. On the one hand, one gets a polyconvex function for all valid elements (for whom $J > 0$). On the other hand, for folded elements, one might encounter numerical issues arising from loss of polyconvexity in the case where a large number of elements possess J in the region $\left(-\beta(I_F+\beta) - 1/\sqrt{3}, -\beta(I_F+\beta) + 1/\sqrt{3}\right)$.

In cases where we would like to compute β for every element and treat it as a variable constrained by δ, it is also possible to show bounds of polyconvexity. Due to the complexity of the energy expression, it is cumbersome to write the analytical bounds, but we plot its variation in Figure 3. In this case, we can also use this arrangement for the J-only regularisation of Garanzha et al. [2021], whose Hessian eigensystem cannot be obtained in closed form.

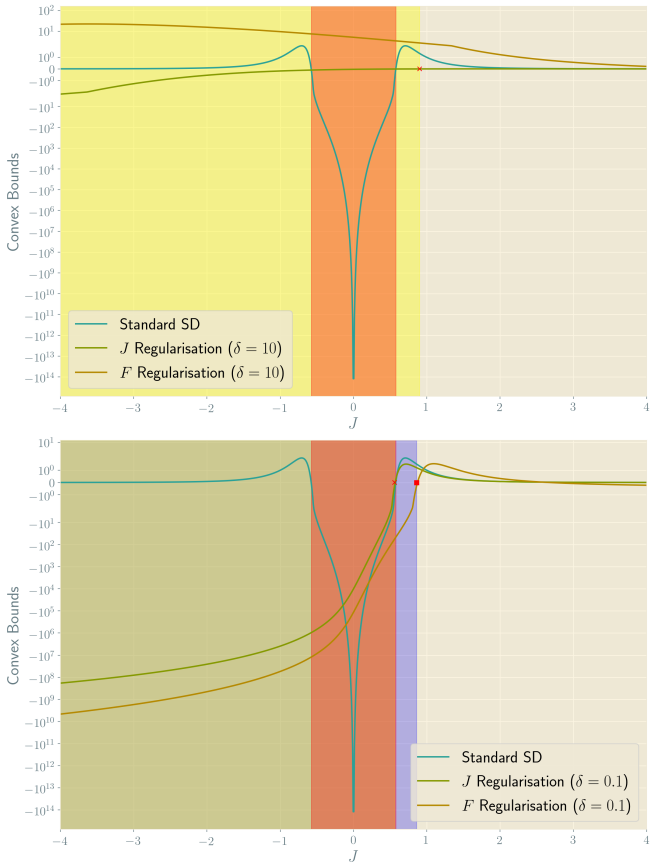

Fig. 3. Loss of polyconvexity for Symmetric Dirichlet, J-only regularised Symmetric Dirichlet and $\boldsymbol{F}$ regularised Symmetric Dirichlet (RSD) with different Jacobian regularisation values ($\delta = 10$ top and $\delta = 0.1$ bottom). Red bars indicate the region for which the Symmetric Dirichlet energy loses polyconvexity i.e. $J \in \left(-1/\sqrt{3}, 1/\sqrt{3}\right)$, while yellow bars indicate the same for J-only regularised variant and blue bars for $\boldsymbol{F}$ regularised variant of the energy. With large values of δ, $\boldsymbol{F}$ regularisation convexifies the energy for all negative values of J.

As can be seen in Figure 3 with large values of δ, $\boldsymbol{F}$ regularisation convexifies the Symmetric Dirichlet energy for the entire negative J region, while J-regularisation expands the non-convex region. However, with small values of δ, $\boldsymbol{F}$ regularisation expands the non-convex zone for positive values of J. The positive region for J is of less importance, as when $J > 0$ we plug in SD instead of RSD. Finally, loss of convexity implies that there is a combination of the invariants that lead to indefiniteness under certain situations.

5 C++ CODE FOR REGULARISED ENERGIES

In Figure 4 and Figure 5 we show two C++ Eigen implementations of RMIPS, using the approaches presented so far, where β is held

```
auto GetFuncGradProjectedHessian(
    const Matrix2d& F, /* The deformation gradient tensor */
    const Real delta   /* J regulariser, computed using the empirical formula */
)
{
    // Compute polyconvex invariants
    // Compute first minor of F - cofactor: H = tr(F)I - F^T
    Matrix2d H = cofactor(F);
    // Compute second minor of F - determinant: J
    Real J = F.determinant();

    // Compute the rotation variant SVD of F
    const auto [U, S, V] = SVD(F);
    Real s1 = S(0, 0);
    Real s2 = S(1, 1);

    // Compute the second invariant
    const Real II_F = F.squaredNorm();

    // Compute regularised J: Jr
    const Real s = std::sqrt(J * J + delta * delta);
    Real Jr = 0.5 * (J + s);

    // Compute consistently regularised energy
    Real energy  = II_F / (2 * Jr) + (1 - J / Jr);

    // Compute consistently regularised PK1
    Matrix2d P = 1. / Jr * (F - H) + 1. / (Jr * s) * (J - I2 / 2.) * H;

    // Create some temporary variables
    const Real tmp1 = s * s * (s - 2 * J + 2 * s2 * s2);
    const Real tmp2 = s * s * (s - 2 * J + 2 * s1 * s1);
    const Real tmp3 = -4 * J * J * Jr + 6*J * s * s - 2 * s * s * s;
    const Real den = s * s * s * Jr;

    // Get the entries of matrix A
    const Real a11 = ((II_F - 2*J) * Jr * s2 * s2 + tmp1) / den;
    const Real a22 = ((II_F - 2*J) * Jr * s1 * s1 + tmp2) / den;
    const Real a12 = (II_F * (2 * J * Jr - 3 * s * s) + tmp3) / (2 * den);

    // Get the eigenvalues of symmetric matrix A << a11, a12, a12, a22
    const Real disc = std::sqrt((a11 - a22) * (a11 - a22) + 4. * a12 * a12);

    // Get all the four eigenvalues of the regularised Hessian
    Real lambda1 = 0.5 * ((a11 + a22) - disc);
    Real lambda2 = 0.5 * ((a11 + a22) + disc);
    Real lambda3 =  (II_F - 2 * J) / (2 * s * Jr) + 2. / Jr;
    Real lambda4 = -(II_F - 2 * J) / (2 * s * Jr);

    // Filter the eigenvalues
    lambda1 = std::max(lambda1, 0.);
    lambda2 = std::max(lambda2, 0.);
    lambda3 = std::max(lambda3, 0.);
    lambda4 = std::max(lambda4, 0.);

    // Get all the four eigenvectors of the regularised Hessian
    const Real iSqrtTwo_v = 1. / std::sqrt(2.);
    // Twist mode
    Matrix2d T; T << 0., -1., 1., 0.;
    T = iSqrtTwo_v * (U * T * V.transpose());
    Vector4d t =  vec(T);

    // Flip mode
    Matrix2d L; L << 0.,  1., 1., 0.;
    L = iSqrtTwo_v * (U * L * V.transpose());
    Vector4d l =  vec(L);

    // Scaling modes
    Matrix2d D1; D1 << 1.,  0., 0., 0.;
    D1 = U * D1 * V.transpose();
    Vector4d d1 =  vec(D1);

    Matrix2d D2; D2 << 0.,  0., 0., 1.;
    D2 = U * D2 * V.transpose();
    Vector4d d2 =  vec(D2);

    // Coupled scaling modes
    const Real tau = (lambda1 - a22) / a12;
    const Real gamma  = std::sqrt(1. + tau * tau);
    Vector4d e1 = 1. / gamma * (tau * d1 + d2);
    Vector4d e2 = 1. / gamma * (d1 - tau * d2);

    // Build the projected regularised Hessian
    Matrix4d hessian = lambda1 * outer(e1, e1) + lambda2 * outer(e2, e2)
        + lambda3 * outer(l, l) + lambda4 * outer(t, t);

    return std::make_tuple(energy, P, hessian);
}
```

Fig. 4. C++ Eigen code for per element computation of the energy, gradient (PK1) and analytically projected Hessian of 2D RMIPS when β is computed per element: For 2D RMIPS, β rather disappears at energy level and only a Jacobian regularisation is needed to obtain a consistent formulation.

```
auto GetFuncGradProjectedHessian(
    const Matrix2d& F, /* The deformation gradient tensor */
    const Real beta   /* F regulariser, computed
        using a desired constraint ideally from the worst singular value or
        the worst Jacobian */
)
{
    // Compute polyconvex invariants
    // Compute first minor of F - cofactor: H = tr(F)I - F^T
    Matrix2d H = cofactor(F);
    // Compute second minor of F - determinant: J
    Real J = F.determinant();

    // Compute the rotation variant SVD of F
    const auto [U, S, V] = SVD(F);
    Real s1 = S(0, 0);
    Real s2 = S(1, 1);
    // Compute rotation tensor
    Matrix2d R = U * V.transpose();

    // Compute the first invariant
    const Real I_F = S.trace();

    // Compute the second invariant
    const Real II_F = F.squaredNorm();

    // Compute regularised J: Jr
    Real Jr = J + beta * I_F + beta * beta;

    // Compute consistently regularised energy
    Real energy  = II_F / (2 * Jr) + (1 - J / Jr);

    // Compute consistently regularised PK1
    Matrix2d P = 1. / Jr * (F - H) + 1. / (Jr * s) * (J - I2 / 2.) * (H + beta * R);

    // Create some temporary variables
    const Real tmp1 = (beta * beta + 2 * beta * s2);
    const Real tmp2 = (beta * beta + 2 * beta * s1);
    const Real tmp3 = 2 * beta * Jr * (s2 - s1);
    const Real tmp4 = s2 * s2 *  II_F - 2 * J * Jr + Jr * Jr;
    const Real tmp5  = s1 * s1 *  II_F - 2 * J * Jr + Jr * Jr;

    // Get the entries of matrix A
    const Real a11 = ((II_F - 2*J) * tmp1 + tmp3 + tmp4) / std::pow(Jr, 3);
    const Real a22 = ((II_F - 2*J) * tmp2 - tmp3 + tmp5) / std::pow(Jr, 3);
    const Real a12 = -II_F / Jr;

    // Get the eigenvalues of symmetric matrix A << a11, a12, a12, a22
    const Real disc = std::sqrt((a11 - a22) * (a11 - a22) + 4. * a12 * a12);

    // Get all the four eigenvalues of the regularised Hessian
    Real lambda1 = 0.5 * ((a11 + a22) - disc);
    Real lambda2 = 0.5 * ((a11 + a22) + disc);
    Real lambda3 =  (II_F - 2 * J) / (2 * Jr * Jr) + 2. / Jr;
    Real lambda4 = -(I_F + 2 * beta) * (II_F - 2 * J) / (2 * I_F * Jr * Jr);

    // Filter the eigenvalues
    lambda1 = std::max(lambda1, 0.);
    lambda2 = std::max(lambda2, 0.);
    lambda3 = std::max(lambda3, 0.);
    lambda4 = std::max(lambda4, 0.);

    // Get all the four eigenvectors of the regularised Hessian
    const Real iSqrtTwo_v = 1. / std::sqrt(2.);
    // Twist mode
    Matrix2d T; T << 0., -1., 1., 0.;
    T = iSqrtTwo_v * (U * T * V.transpose());
    Vector4d t =  vec(T);

    // Flip mode
    Matrix2d L; L << 0.,  1., 1., 0.;
    L = iSqrtTwo_v * (U * L * V.transpose());
    Vector4d l =  vec(L);

    // Scaling modes
    Matrix2d D1; D1 << 1.,  0., 0., 0.;
    D1 = U * D1 * V.transpose();
    Vector4d d1 =  vec(D1);

    Matrix2d D2; D2 << 0.,  0., 0., 1.;
    D2 = U * D2 * V.transpose();
    Vector4d d2 =  vec(D2);

    // Coupled scaling modes
    const Real tau = (lambda1 - a22) / a12;
    const Real gamma  = std::sqrt(1. + tau * tau);
    Vector4d e1 = 1. / gamma * (tau * d1 + d2);
    Vector4d e2 = 1. / gamma * (d1 - tau * d2);

    // Build the projected regularised Hessian
    Matrix4d hessian = lambda1 * outer(e1, e1) + lambda2 * outer(e2, e2)
        + lambda3 * outer(l, l) + lambda4 * outer(t, t);

    return std::make_tuple(energy, P, hessian);
}
```

Fig. 5. C++ Eigen code for per element computation of the energy, gradient (PK1) and analytically projected Hessian of 2D RMIPS when β is a constant precomputed from the worst (negative) singular-value or the worst (negative) Jacobian or any other similar constraint.

constant, and when computed per element. In Figure 7 and Figure 8 we show two analogous C++ Eigen implementations of RSD energy. The 3D Hessians eigensystems can become lengthy, so they are not provided within this document, but can be obtained from the Python script provided as supplement. However, a generic element-wise kernel for computing 3D Hessians is provided in Figure 6. The `vec` and `outer` function appearing in these element kernels are listed in Listing 3 and Listing 4. These element kernels can be embedded directly in to any existing second order solver.

REFERENCES

J. Bonet, A. J. Gil, and R. Ortigosa. 2015. A computational framework for polyconvex large strain elasticity. *Computer Methods in Applied Mechanics and Engineering* 283 (2015), 1061–1094.

```
auto GetFuncGradProjectedHessian(
    const Matrix2d& F, /* The deformation gradient tensor */
    const Real beta   /* F regulariser, computed
        using a desired constraint ideally from the worst singular value or
        the worst Jacobian */
)
{
    // Compute polyconvex invariants
    // Compute first minor of F - cofactor: H = tr(F)I - F^T
    Matrix2d H = cofactor(F);
    // Compute second minor of F - determinant: J
    Real J = F.determinant();

    // Compute the rotation variant SVD of F
    const auto [U, S, V] = SVD(F);
    Real s1 = S(0, 0);
    Real s2 = S(1, 1);
    // Compute rotation tensor
    Matrix2d R = U * V.transpose();

    // Compute the first invariant
    const Real I_F = S.trace();

    // Compute the second invariant
    const Real II_F = F.squaredNorm();

    // Compute regularised J: Jr
    Real Jr = J + beta * I_F + beta * beta;

    // Compute consistently regularised energy
    Real energy  = (1. + 1. / Jr / Jr) * II_F / 2. + (Jr + 1. / Jr - J - J / Jr / Jr);

    // Compute consistently regularised PK1
    Matrix2d Hr = H + beta * R;
    Matrix2d P = (1. + 1. / Jr / Jr) * (F - H) +
        (1. - 1. / Jr / Jr + (2*J - I2) / Jr / Jr / Jr) * Hr;

    // Create some temporary variables
    const Real Jr2 = Jr * Jr;
    const Real Jr3 = Jr2 * Jr;
    const Real Jr4 = Jr3 * Jr;
    const Real beta2 = beta * beta;

    // Get the entries of matrix A
    const Real a11 =  (Jr4 + Jr2 - 4*Jr*(J + beta*s1) + beta2*(3*II_F - 6*J + 2*Jr) +
        2*beta*s2*(3*II_F - 6*J + 4*Jr) + 3*s2*s2*(II_F - 2*J + 2*Jr)) / Jr4;
    const Real a12 =  (I_F*beta*(3*II_F - 6*J + 4*Jr) + 3*J*(II_F - 2*J + 2*Jr) - 2*Jr2 +
        Jr*(-2*I_F*beta - 3*II_F + 2*J) + beta2*(3*II_F - 6*J + 2*Jr)) / Jr4;
    const Real a22 =  (Jr4 + Jr2 - 4*Jr*(J + beta*s2) + beta2*(3*II_F - 6*J + 2*Jr) +
        2*beta*s1*(3*II_F - 6*J + 4*Jr) + 3*s1*s1*(II_F - 2*J + 2*Jr)) / Jr4;

    // Get the eigenvalues of symmetric matrix A << a11, a12, a12, a22
    const Real disc = std::sqrt((a11 - a22) * (a11 - a22) + 4. * a12 * a12);

    // Get all the four eigenvalues of the regularised Hessian
    Real lambda1 = 0.5 * ((a11 + a22) - disc);
    Real lambda2 = 0.5 * ((a11 + a22) + disc);
    Real lambda3 =  (II_F - 2*J + Jr3 + 3*Jr) / Jr3;
    Real lambda4 =  (I_F * Jr3 - I_F*Jr + I_F * (-II_F + 2*J) +
        2*beta*(-II_F + 2*J + Jr3 - Jr))/(I_F * Jr3);

    // Filter the eigenvalues
    lambda1 = std::max(lambda1, 0.);
    lambda2 = std::max(lambda2, 0.);
    lambda3 = std::max(lambda3, 0.);
    lambda4 = std::max(lambda4, 0.);

    // Get all the four eigenvectors of the regularised Hessian
    const Real iSqrtTwo_v = 1. / std::sqrt(2.);
    // Twist mode
    Matrix2d T; T << 0., -1., 1., 0.;
    T = iSqrtTwo_v * (U * T * V.transpose());
    Vector4d t =  vec(T);

    // Flip mode
    Matrix2d L; L << 0.,  1., 1., 0.;
    L = iSqrtTwo_v * (U * L * V.transpose());
    Vector4d l =  vec(L);

    // Scaling modes
    Matrix2d D1; D1 << 1.,  0., 0., 0.;
    D1 = U * D1 * V.transpose();
    Vector4d d1 =  vec(D1);

    Matrix2d D2; D2 << 0.,  0., 0., 1.;
    D2 = U * D2 * V.transpose();
    Vector4d d2 =  vec(D2);

    // Coupled scaling modes
    const Real tau = (lambda1 - a22) / a12;
    const Real gamma  = std::sqrt(1. + tau * tau);
    Vector4d e1 = 1. / gamma * (tau * d1 + d2);
    Vector4d e2 = 1. / gamma * (d1 - tau * d2);

    // Build the projected regularised Hessian
    Matrix4d hessian = lambda1 * outer(e1, e1) + lambda2 * outer(e2, e2)
        + lambda3 * outer(l, l) + lambda4 * outer(t, t);

    return std::make_tuple(energy, P, hessian);
}
```

Fig. 7. C++ Eigen code for per element computation of the energy, gradient (PK1) and analytically projected Hessian of 2D RSD when β is a constant pre-computed from the worst (negative) singular-value or the worst (negative) Jacobian or any other similar constraint.

```
auto GetFuncGradProjectedHessian(
    const Matrix2d& F, /* The deformation gradient tensor */
    const Real delta   /* J regulariser, computed using the empirical formula */
)
{
    // Compute polyconvex invariants
    // Compute first minor of F - cofactor: H = tr(F)I - F^T
    Matrix2d H = cofactor(F);
    // Compute second minor of F - determinant: J
    Real J = F.determinant();

    // Compute the rotation variant SVD of F
    const auto [U, S, V] = SVD(F);
    Real s1 = S(0, 0);
    Real s2 = S(1, 1);
    // Compute rotation tensor
    Matrix2d R = U * V.transpose();

    // Compute the first invariant
    const Real I_F = S.trace();

    // Compute the second invariant
    const Real II_F = F.squaredNorm();

    // Compute regularised J: Jr
    const Real s = std::sqrt(J * J + delta * delta);
    Real Jr = 0.5 * (J + s);

    // Compute consistently regularised energy
    Real energy  = (1. + 1. / Jr / Jr) * II_F / 2. + (Jr + 1. / Jr - J - J / Jr / Jr);

    // Compute consistently regularised PK1
    Matrix2d Hr = 0.5 * (1. + J / s) * H;
    Matrix2d P = (1. + 1. / Jr / Jr) * (F - H) +
        (1. - 1. / Jr / Jr + (2*J - I2) / Jr / Jr / Jr) * Hr;

    // Create some temporary variables
    const Real ss  = s * s;
    const Real sss = ss * s;
    const Real sssss = sss * ss;
    const Real J2 = J * J;
    const Real J3 = J2 * J;
    const Real J4 = J3 * J;
    const Real J5 = J4 * J;
    const Real JS2 = (J + s) * (J + s);
    const Real JS3 = JS2 * (J + s);
    const Real s12 = s1 * s1;
    const Real s22 = s2 * s2;

    // Get the entries of matrix A
    const Real a11 =  (1.0/2.0)*(-32*J*ss*(J + s) + 2*sss*JS3 + 8*sss*(J + s) +
    s22*(-J2*JS3 + ss*JS3 + 32*ss*(J + s) + 24*s*(I2 - 2*J)*(J + s) + 8*s*JS2 +
    8*(I2 - 2*J)*(J2 - ss) + 4*(J + s)*(J2 - ss)))/(sss*JS3);
    const Real a12 =  -1.0/2.0*(-8*I2*J2 - 16*I2*J*s + 24*I2*ss + J5 + 2*J4*s -
    J3*ss + 12*J3 - 3*J2*sss + 24*J2*s - 48*J*ss + sssss + 12*sss)/(sss*(J2 + 2*J*s + ss));
    const Real a22 =  (1.0/2.0)*(-32*J*ss*(J + s) + 2*sss*JS3 + 8*sss*(J + s)
    + s12*(-J2*JS3 + ss*JS3 + 32*ss*(J + s) + 24*s*(I2 - 2*J)*(J + s)
    + 8*s*JS2 + 8*(I2 - 2*J)*(J2 - ss) + 4*(J + s)*(J2 - ss)))/(sss*JS3);

    // Get the eigenvalues of symmetric matrix A << a11, a12, a12, a22
    const Real disc = std::sqrt((a11 - a22) * (a11 - a22) + 4. * a12 * a12);

    // Get all the four eigenvalues of the regularised Hessian
    Real lambda1 = 0.5 * ((a11 + a22) - disc);
    Real lambda2 = 0.5 * ((a11 + a22) + disc);
    Real lambda3 =  (II_F - 2*J + Jr3 + 3*Jr) / Jr3;
    Real lambda4 =  (I_F * Jr3 - I_F*Jr + I_F * (-II_F + 2*J) +
        2*beta*(-II_F + 2*J + Jr3 - Jr))/(I_F * Jr3);

    // Filter the eigenvalues
    lambda1 = std::max(lambda1, 0.);
    lambda2 = std::max(lambda2, 0.);
    lambda3 = std::max(lambda3, 0.);
    lambda4 = std::max(lambda4, 0.);

    // Get all the four eigenvectors of the regularised Hessian
    const Real iSqrtTwo_v = 1. / std::sqrt(2.);
    // Twist mode
    Matrix2d T; T << 0., -1., 1., 0.;
    T = iSqrtTwo_v * (U * T * V.transpose());
    Vector4d t =  vec(T);

    // Flip mode
    Matrix2d L; L << 0.,  1., 1., 0.;
    L = iSqrtTwo_v * (U * L * V.transpose());
    Vector4d l =  vec(L);

    // Scaling modes
    Matrix2d D1; D1 << 1.,  0., 0., 0.;
    D1 = U * D1 * V.transpose();
    Vector4d d1 =  vec(D1);

    Matrix2d D2; D2 << 0.,  0., 0., 1.;
    D2 = U * D2 * V.transpose();
    Vector4d d2 =  vec(D2);

    // Coupled scaling modes
    const Real tau = (lambda1 - a22) / a12;
    const Real gamma  = std::sqrt(1. + tau * tau);
    Vector4d e1 = 1. / gamma * (tau * d1 + d2);
    Vector4d e2 = 1. / gamma * (d1 - tau * d2);

    // Build the projected regularised Hessian
    Matrix4d hessian = lambda1 * outer(e1, e1) + lambda2 * outer(e2, e2)
        + lambda3 * outer(l, l) + lambda4 * outer(t, t);

    return std::make_tuple(energy, P, hessian);
}
```

Fig. 8. C++ Eigen code for per element computation of the energy, gradient (PK1) and analytically projected Hessian of 2D RSD when β is computed per element: For 2D RSD, β rather disappears at energy level and only a Jacobian regularisation is needed to obtain a consistent formulation.

```
auto GetFuncGradProjectedHessian(
    const Matrix3d& F, /* The deformation gradient tensor */
    const Real beta    /* F regulariser */
)
{
    // Compute polyconvex invariants
    // Compute first minor of F - cofactor: H
    Matrix3d H = cofactor(F);
    // Compute second minor of F - determinant: J
    Real J = F.determinant();

    // Compute the rotation variant SVD of F
    const auto [U, S, V] = SVD(F);
    Real s1 = S(0, 0);
    Real s2 = S(1, 1);
    Real s2 = S(2, 2);

    // Compute rotation tensor
    Matrix3d R = U * V.transpose();

    // Compute the first invariant
    const Real I_F = S.trace();
    // Compute the second invariant
    const Real II_F = F.squaredNorm();
    // Compute the third invariant
    const Real I_H = 0.5 * (I_F * I_F - II_F);

    // Compute regularised J: Jr
    Real Jr = J + beta * I_H + beta * beta * I_F + beta * beta * beta;

    // Compute energy
    Real energy; // embed energy of choice

    // Compute PK1
    Matrix3d P; // Eqn. 26 in the main article

    // Twist modes
    Matrix3d T1; T1 <<  0., 0.,  0., 0., 0., -1., 0., 1., 0.;
    Matrix3d T2; T2 <<  0., 0., -1., 0., 0.,  0., 1., 0., 0.;
    Matrix3d T3; T3 <<  0.,-1.,  0., 1., 0.,  0., 0., 0., 0.;

    T1 = iSqrtTwo_v * (U * T1 * V.transpose());
    T2 = iSqrtTwo_v * (U * T2 * V.transpose());
    T3 = iSqrtTwo_v * (U * T3 * V.transpose());

    Vector9d t1 = vec(T1);
    Vector9d t2 = vec(T2);
    Vector9d t3 = vec(T3);

    // Flip modes
    Matrix3d L1; L1 <<  0., 0., 0., 0., 0., 1., 0., 1., 0.;
    Matrix3d L2; L2 <<  0., 0., 1., 0., 0., 0., 1., 0., 0.;
    Matrix3d L3; L3 <<  0., 1., 0., 1., 0., 0., 0., 0., 0.;

    L1 = iSqrtTwo_v * (U * L1 * V.transpose());
    L2 = iSqrtTwo_v * (U * L2 * V.transpose());
    L3 = iSqrtTwo_v * (U * L3 * V.transpose());

    Vector9d l1 = vec(L1);
    Vector9d l2 = vec(L2);
    Vector9d l3 = vec(L3);

    // Scaling modes
    Matrix3d D1; D1 <<  1. , 0., 0., 0. , 0., 0., 0. , 0., 0.;
    Matrix3d D2; D2 <<  0. , 0., 0., 0. , 1., 0., 0. , 0., 0.;
    Matrix3d D3; D3 <<  0. , 0., 0., 0. , 0., 0., 0. , 0., 1.;

    D1 = U * D1 * V.transpose();
    D2 = U * D2 * V.transpose();
    D3 = U * D3 * V.transpose();

    Vector9d d1 = vec(D1);
    Vector9d d2 = vec(D2);
    Vector9d d3 = vec(D3);

    // Get entries of matrix A from the Python script
    const Real a11, a12, a13, a23, a33;

    Matrix3d A; A << a11, a12, a13, a12, a22, a23, a13, a23, a33;

    Eigen::SelfAdjointEigenSolver<Matrix3d> solver(A);
    Vector3d _eigvals  = solver.eigenvalues();
    Matrix3d _eigvecs  = solver.eigenvectors();

    Real lambda1 = _eigvals[0];
    Real lambda2 = _eigvals[1];
    Real lambda3 = _eigvals[2];

    // Get lambda4 ... lambda9 from the Python script
    Real lambda4, lambda5, lambda6, lambda7, lambda8, lambda9;

    // Filter the eigenvalues
    lambda1 = std::max(lambda1, 0.);
    lambda2 = std::max(lambda2, 0.);
    lambda3 = std::max(lambda3, 0.);
    lambda4 = std::max(lambda4, 0.);
    lambda5 = std::max(lambda5, 0.);
    lambda6 = std::max(lambda6, 0.);
    lambda7 = std::max(lambda7, 0.);
    lambda8 = std::max(lambda8, 0.);
    lambda9 = std::max(lambda9, 0.);

    Vector9d e1 = _eigvecs(0,0) * d1 + _eigvecs(1,0) * d2 + _eigvecs(2,0) * d3;
    Vector9d e2 = _eigvecs(0,1) * d1 + _eigvecs(1,1) * d2 + _eigvecs(2,1) * d3;
    Vector9d e3 = _eigvecs(0,2) * d1 + _eigvecs(1,2) * d2 + _eigvecs(2,2) * d3;

    Matrix9d hessian = lambda1 * outer(e1,e1) +
        lambda2 * outer(e2,e2) + lambda3 * outer(e3,e3) +
        lambda4 * outer(t1,t1) + lambda5 * outer(t2,t2) +
        lambda6 * outer(t3,t3) + lambda7 * outer(l1,l1) +
        lambda8 * outer(l2,l2) + lambda9 * outer(l3,l3);

    return std::make_tuple(energy, P, hessian);
}
```

Fig. 6. C++ Eigen code for per element computation of analytically projected Hessian of *any* regularised 3D energy. The entries of matrix $\boldsymbol{A}$ and eigenvalues of the 3D Hessian can be obtained from the Python script and embedded here.

J. Bonet, A. J. Gil, and R. Ortigosa. 2016. On a tensor cross product based formulation of large strain solid mechanics. *International Journal of Solids and Structures* 84 (2016), 49–63.

R. de Boer. 1982. *Vektor- und Tensorrechnung für Ingenieure.* Springer.

Xiao-Ming Fu and Yang Liu. 2016. Computing Inversion-Free Mappings by Simplex Assembly. 35, 6, Article 216 (Nov. 2016), 12 pages.

Vladimir Garanzha, Igor Kaporin, Liudmila Kudryavtseva, François Protais, Nicolas Ray, and Dmitry Sokolov. 2021. Foldover-Free Maps in 50 Lines of Code. *ACM Trans. Graph.* 40, 4, Article 102 (Jul 2021), 16 pages.

S. Hartmann and P. Neff. 2003. Polyconvexity of generalized polynomial-type hyperelastic strain energy functions for near-incompressibility. *International Journal of Solids and Structures* 40 (2003), 2767–2791.

K. Hormann and G. Greiner. 2000. MIPS: An Efficient Global Parametrization Method. In *Curve and Surface Design.* 153–162.

J. Jeong, H. Ramézani, I. Munch, and P. Neff. 2009. A numerical study for linear isotropic Cosserat elasticity with conformally invariant curvature. *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik* 89 (2009), 552–569. Issue 7.

Alex Kraus, Peter Wriggers, Nils Viebahn, and Jörg Schröder. 2019. Low order locking–free mixed finite element formulation with approximation of the minors of the deformation gradient. *Internat. J. Numer. Methods Engrg.* 120 (2019), 1011–1026. Issue 8.

P. Neff. 2006. The Cosserat couple modulus for continuous solids is zero viz the linearized Cauchy-stress tensor is symmetric. *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik* 86 (2006), 892–912. Issue 11.

Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* 216 (2017), 35 – 52.

J. Schröder and P. Neff. 2003. Invariant formulation of hyperelastic transverse isotropy based on polyconvex free energy functions. *International Journal of Solids and Structures* 40 (2003), 401–445.

Jörg Schröder, Peter Wriggers, and Daniel Balzani. 2011. A new mixed finite element based on different approximations of the minors of deformation tensors. *Computer Methods in Applied Mechanics and Engineering* 200, 49–52 (2011), 3583–3600.

Hanxiao Shen, Zhongshi Jiang, Denis Zorin, and Daniele Panozzo. 2019. Progressive Embedding. *ACM Trans. Graph.* 38, 4, Article 32 (July 2019), 13 pages.

Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable Neo-Hookean Flesh Simulation. *ACM Trans. Graph.* 37, 2, Article 12 (March 2018), 15 pages.

B. Smith, F. De Goes, and T. Kim. 2019. Analytic Eigensystems for Isotropic Distortion Energies. *ACM Trans. Graph.* 38, 1, Article 3 (Feb. 2019), 15 pages.

Listing 9. C++ Eigen code for computing $\boldsymbol{F} \times \mathbb{I} \times \boldsymbol{F}$

```
template<typename T>
Eigen::Matrix<T, 9, 9> FCrossICrossF(const Eigen::Matrix<T, 3, 3>& F)
{
    const Real F11 = F(0,0);
    const Real F12 = F(0,1);
    const Real F13 = F(0,2);
    const Real F21 = F(1,0);
    const Real F22 = F(1,1);
    const Real F23 = F(1,2);
    const Real F31 = F(2,0);
    const Real F32 = F(2,1);
    const Real F33 = F(2,2);

    Eigen::Matrix<T, 9, 9> FxIxF = Eigen::Matrix<T, 9, 9>::Zero();
    FxIxF(0,1) = -F23*F31 + F31*F32;
    FxIxF(0,2) =  F21*F23 - F21*F32;
    FxIxF(0,4) =  F13*F31 + F32*F32 + F33*F33;
    FxIxF(0,5) = -F13*F21 - F22*F32 - F23*F33;
    FxIxF(0,7) = -F12*F31 - F22*F32 - F23*F33;
    FxIxF(0,8) =  F12*F21 + F22*F22 + F23*F23;
    FxIxF(1,0) =  F23*F31 - F31*F32;
    FxIxF(1,2) = -F11*F23 + F11*F32;
    FxIxF(1,3) = -F13*F31 - F32*F32 - F33*F33;
    FxIxF(1,5) =  F11*F13 + F12*F32 + F13*F33;
    FxIxF(1,6) =  F12*F31 + F22*F32 + F23*F33;
    FxIxF(1,8) = -F11*F12 - F12*F22 - F13*F23;
    FxIxF(2,0) = -F21*F23 + F21*F32;
    FxIxF(2,1) =  F11*F23 - F11*F32;
    FxIxF(2,3) =  F13*F21 + F22*F32 + F23*F33;
    FxIxF(2,4) = -F11*F13 - F12*F32 - F13*F33;
    FxIxF(2,6) = -F12*F21 - F22*F22 - F23*F23;
    FxIxF(2,7) =  F11*F12 + F12*F22 + F13*F23;
    FxIxF(3,1) = -F23*F32 - F31*F31 - F33*F33;
    FxIxF(3,2) =  F21*F31 + F22*F23 + F23*F33;
    FxIxF(3,4) =  F13*F32 - F31*F32 ;
    FxIxF(3,5) = -F13*F22 + F22*F31;
    FxIxF(3,7) =  F11*F31 + F13*F33 + F21*F32;
    FxIxF(3,8) = -F11*F21 - F13*F23 - F21*F22;
    FxIxF(4,0) =  F23*F32 + F31*F31 + F33*F33;
    FxIxF(4,2) = -F11*F31 - F12*F23 - F13*F33 ;
    FxIxF(4,3) = -F13*F32 + F31*F32;
    FxIxF(4,5) =  F12*F13 - F12*F31;
    FxIxF(4,6) = -F11*F31 - F13*F33 - F21*F32;
    FxIxF(4,8) =  F11*F11 + F12*F21 + F13*F13;
    FxIxF(5,0) = -F21*F31 - F22*F23 - F23*F33;
    FxIxF(5,1) =  F11*F31 + F12*F23 + F13*F33;
    FxIxF(5,3) =  F13*F22 - F22*F31;
    FxIxF(5,4) = -F12*F13 + F12*F31;
    FxIxF(5,6) =  F11*F21 + F13*F23 + F21*F22;
    FxIxF(5,7) = -F11*F11 - F12*F21 - F13*F13;
    FxIxF(6,1) =  F21*F31 + F22*F32 + F32*F33;
    FxIxF(6,2) = -F21*F21 - F22*F22 - F23*F32;
    FxIxF(6,4) = -F11*F31 - F12*F32 - F31*F33;
    FxIxF(6,5) =  F11*F21 + F12*F22 + F23*F31;
    FxIxF(6,7) = -F12*F33 + F21*F33;
    FxIxF(6,8) =  F12*F23 - F21*F23;
    FxIxF(7,0) = -F21*F31 - F22*F32 - F32*F33;
    FxIxF(7,2) =  F11*F21 + F12*F22 + F13*F32;
    FxIxF(7,3) =  F11*F31 + F12*F32 + F31*F33;
    FxIxF(7,5) = -F11*F11 - F12*F12 - F13*F31;
    FxIxF(7,6) =  F12*F33 - F21*F33;
    FxIxF(7,8) = -F12*F13 + F13*F21;
    FxIxF(8,0) =  F21*F21 + F22*F22 + F23*F32;
    FxIxF(8,1) = -F11*F21 - F12*F22 - F13*F32;
    FxIxF(8,3) = -F11*F21 - F12*F22 - F23*F31;
    FxIxF(8,4) =  F11*F11 + F12*F12 + F13*F31;
    FxIxF(8,6) = -F12*F23 + F21*F23;
    FxIxF(8,7) =  F12*F13 - F13*F21;

    return FxIxF;
}
```