PSI

Building Blocks for Finite Element computations in IPPL

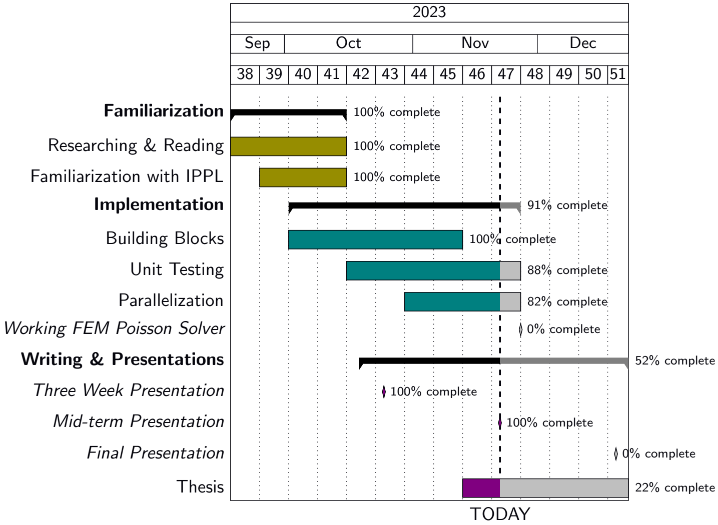
Mid-term Presentation

Lukas Bühler

Tuesday 21st November, 2023

Outline

Timeline (from 3-week presentation)



Outline

Transformation and Pullback

Transformation function (local ref. element $\hat{K}$ to global element K):

$$\mathbf{\Phi}_K : \hat{K} \mapsto K$$

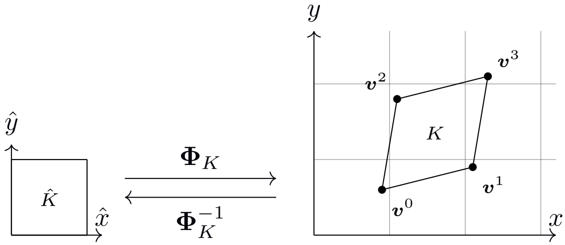
We define the pullback $\mathbf{\Phi}_K^* f$ of a function f as

$$(\mathbf{\Phi}_K^* u)(\hat{\boldsymbol{x}}) := u(\mathbf{\Phi}_K(\hat{\boldsymbol{x}})), \quad \hat{\boldsymbol{x}} \in \hat{K}$$

PSI

Affine Transformation Example

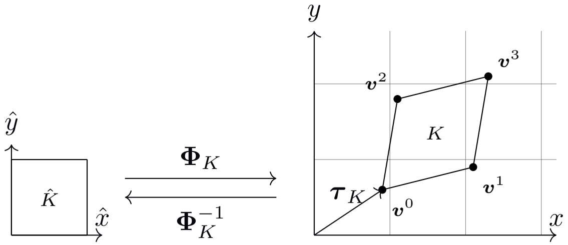
Affine Transformation: $\mathbf{\Phi}_K(\hat{\boldsymbol{x}}) = \mathbf{F}_K\hat{\boldsymbol{x}} + \boldsymbol{\tau}_K$



$$\text{with } \hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix},\ \boldsymbol{v}^i = \begin{pmatrix} \boldsymbol{v}^i_x \\ \boldsymbol{v}^i_y \end{pmatrix},\ \mathbf{F}_K = \begin{bmatrix} \boldsymbol{v}^1_x - \boldsymbol{v}^0_x & \boldsymbol{v}^2_x - \boldsymbol{v}^0_x \\ \boldsymbol{v}^1_y - \boldsymbol{v}^0_y & \boldsymbol{v}^2_y - \boldsymbol{v}^0_y \end{bmatrix},\ \boldsymbol{\tau}_K = \boldsymbol{v}^0$$

Affine Transformation Example

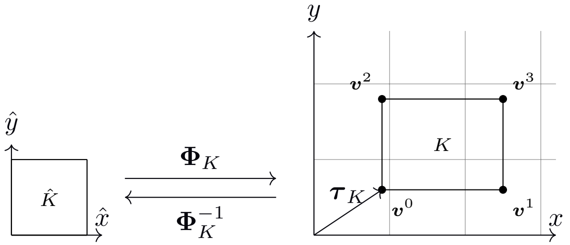
Affine Transformation: $\mathbf{\Phi}_K(\hat{\boldsymbol{x}}) = \mathbf{F}_K \hat{\boldsymbol{x}} + \boldsymbol{\tau}_K$



with $\hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix}, \; \boldsymbol{v}^i = \begin{pmatrix} \boldsymbol{v}_x^i \\ \boldsymbol{v}_y^i \end{pmatrix}, \; \mathbf{F}_K = \begin{bmatrix} \boldsymbol{v}_x^1 - \boldsymbol{v}_x^0 & \boldsymbol{v}_x^2 - \boldsymbol{v}_x^0 \\ \boldsymbol{v}_y^1 - \boldsymbol{v}_y^0 & \boldsymbol{v}_y^2 - \boldsymbol{v}_y^0 \end{bmatrix}, \; \boldsymbol{\tau}_K = \boldsymbol{v}^0$

Affine Transformation Example

Affine Transformation: $\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}) = \mathbf{F}_K \hat{\boldsymbol{x}} + \boldsymbol{\tau}_K$



with $\hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix}$, $\boldsymbol{v}^i = \begin{pmatrix} \boldsymbol{v}_x^i \\ \boldsymbol{v}_y^i \end{pmatrix}$, $\mathbf{F}_K = \begin{bmatrix} \boldsymbol{v}_x^1 - \boldsymbol{v}_x^0 & 0 \\ 0 & \boldsymbol{v}_y^2 - \boldsymbol{v}_y^0 \end{bmatrix}$, $\boldsymbol{\tau}_K = \boldsymbol{v}^0$

Applying Pullback

Rewriting integrals over elements:

$$\int_K u(\boldsymbol{x})\ d\boldsymbol{x} = \int_{\hat{K}} (\boldsymbol{\Phi}_K^* u)(\hat{\boldsymbol{x}}) |\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}})|\ d\hat{\boldsymbol{x}}$$
$$= \int_{\hat{K}} u(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}})) |\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}})|\ d\hat{\boldsymbol{x}}$$

Rewriting gradients:

$$\boldsymbol{\Phi}_K^*(\nabla_{\boldsymbol{x}}\, u)(\hat{\boldsymbol{x}}) = (\mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))^{-\top} \underbrace{(\nabla_{\hat{\boldsymbol{x}}}(\boldsymbol{\Phi}^* u))(\hat{\boldsymbol{x}})}_{=(\nabla_{\hat{\boldsymbol{x}}} u)(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))}$$

with $\mathbf{S}^{-\top} := (\mathbf{S}^{-1})^\top = (\mathbf{S}^\top)^{-1}$

Jacobian of the Transformation (Example)

$$\mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}) = \left[\frac{\partial(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))_i}{\partial \boldsymbol{x}_j}\right]_{i,j=1}^{d} = \begin{bmatrix} \boldsymbol{v}_x^1 - \boldsymbol{v}_x^0 & 0 \\ 0 & \boldsymbol{v}_y^2 - \boldsymbol{v}_y^0 \end{bmatrix}$$

Jacobian of the Transformation (Example)

$$\mathbf{D\Phi}_K(\hat{\boldsymbol{x}}) = \left[\frac{\partial(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))_i}{\partial \boldsymbol{x}_j}\right]_{i,j=1}^{d} = \begin{bmatrix} \boldsymbol{v}_x^1 - \boldsymbol{v}_x^0 & 0 \\ 0 & \boldsymbol{v}_y^2 - \boldsymbol{v}_y^0 \end{bmatrix}$$

$$|\det \mathbf{D\Phi}_K(\hat{\boldsymbol{x}})| = |(\boldsymbol{v}_x^1 - \boldsymbol{v}_x^0)(\boldsymbol{v}_y^2 - \boldsymbol{v}_y^0)|$$

Jacobian of the Transformation (Example)

$$\mathbf{D\Phi}_K(\hat{\boldsymbol{x}}) = \left[\frac{\partial(\mathbf{\Phi}_K(\hat{\boldsymbol{x}}))_i}{\partial \boldsymbol{x}_j}\right]_{i,j=1}^{d} = \begin{bmatrix} \boldsymbol{v}_x^1 - \boldsymbol{v}_x^0 & 0 \\ 0 & \boldsymbol{v}_y^2 - \boldsymbol{v}_y^0 \end{bmatrix}$$

$$|\det \mathbf{D\Phi}_K(\hat{\boldsymbol{x}})| = |(\boldsymbol{v}_x^1 - \boldsymbol{v}_x^0)(\boldsymbol{v}_y^2 - \boldsymbol{v}_y^0)|$$

$$\mathbf{D\Phi}_K^{-1}(\hat{\boldsymbol{x}}) = \begin{bmatrix} \frac{1}{\boldsymbol{v}_x^1 - \boldsymbol{v}_x^0} & 0 \\ 0 & \frac{1}{\boldsymbol{v}_y^2 - \boldsymbol{v}_y^0} \end{bmatrix}$$

Outline

Assembly Algorithm (`evaluateAx`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

    // 1. Compute the Element matrix $\mathbf{A}_K$

    DOFs$_K \leftarrow \{4, 5, 8, 7\}$, DOFs$_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

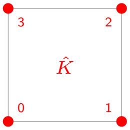
    ...

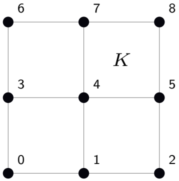
    // 2. Compute $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ contribution with $\mathbf{A}_K$

    ...

end

return $\boldsymbol{z}$





Assembly Algorithm (`evaluateAx`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element matrix` $\mathbf{A}_K$

  $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i, j \in \mathsf{DOFs}_{\hat{K}}$ **do**

    $I = \mathsf{DOFs}_K[i],\ J = \mathsf{DOFs}_K[j]$

    $$(\mathbf{A}_K)_{i,j} = \int_K \nabla b_K^J(\boldsymbol{x}) \cdot \nabla b_K^I(\boldsymbol{x})\ d\boldsymbol{x}$$

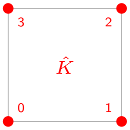
  end

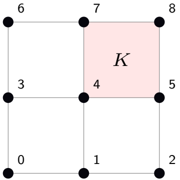
  `// 2. Compute` $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ `contribution with` $\mathbf{A}_K$

  ...

end

return $\boldsymbol{z}$





Assembly Algorithm (`evaluateAx`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element matrix` $\mathbf{A}_K$

  $\text{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\text{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i, j \in \text{DOFs}_{\hat{K}}$ **do**

    $I = \text{DOFs}_K[i],\ J = \text{DOFs}_K[j]$

    $$(\mathbf{A}_K)_{i,j} = \int_{\hat{K}} \mathbf{\Phi}_K^* \nabla b_K^J \cdot \mathbf{\Phi}_K^* \nabla b_K^I \,|\det \mathbf{D\Phi}_K|\, d\hat{\boldsymbol{x}}$$

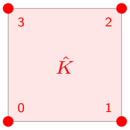
  end

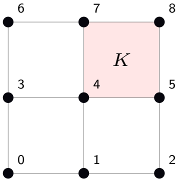
  `// 2. Compute` $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ `contribution with` $\mathbf{A}_K$

  ...

end

return $\boldsymbol{z}$





Assembly Algorithm (`evaluateAx`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element matrix` $\mathbf{A}_K$

  $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i, j \in \mathsf{DOFs}_{\hat{K}}$ **do**

$$(\mathbf{A}_K)_{i,j} = \int_{\hat{K}} (\mathbf{D}\boldsymbol{\Phi}_K)^{-\top} \nabla \hat{b}^j \cdot (\mathbf{D}\boldsymbol{\Phi}_K)^{-\top} \nabla \hat{b}^i \,|\det \mathbf{D}\boldsymbol{\Phi}_K|\, d\hat{\boldsymbol{x}}$$

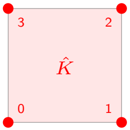
  end

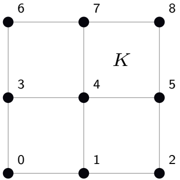
  `// 2. Compute` $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ `contribution with` $\mathbf{A}_K$

  ...

end

return $\boldsymbol{z}$





Assembly Algorithm (`evaluateAx`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element matrix` $\mathbf{A}_K$

  $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i, j \in \mathsf{DOFs}_{\hat{K}}$ **do**

$$(\mathbf{A}_K)_{i,j} \approx \sum_{k}^{N_{\mathsf{Int}}} \hat{\omega}_k (\mathbf{D\Phi}_K(\hat{\boldsymbol{q}}_k))^{-\top} \nabla \hat{b}^j(\hat{\boldsymbol{q}}_k) \cdot (\mathbf{D\Phi}(\hat{\boldsymbol{q}}_k))^{-\top} \nabla \hat{b}^i(\hat{q}_k) |\det \mathbf{D\Phi}_K(\hat{\boldsymbol{q}}_k)|$$

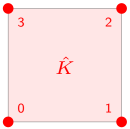
  end

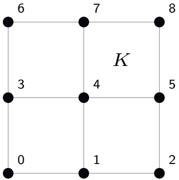
  `// 2. Compute` $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ `contribution with` $\mathbf{A}_K$

  ...

end

return $\boldsymbol{z}$





Assembly Algorithm (evaluateAx)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  // 1. Compute the Element matrix $\mathbf{A}_K$

  $\text{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\text{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i, j \in \text{DOFs}_{\hat{K}}$ **do**

$$(\mathbf{A}_K)_{i,j} \approx \sum_{k}^{N_{\text{Int}}} \hat{\omega}_k (\mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{q}}_k))^{-\top} \nabla \hat{b}^j(\hat{\boldsymbol{q}}_k) \cdot (\mathbf{D}\boldsymbol{\Phi}(\hat{\boldsymbol{q}}_k))^{-\top} \nabla \hat{b}^i(\hat{q}_k) |\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{q}}_k)|$$

  end

  // 2. Compute $\boldsymbol{z} = \mathbf{A}\boldsymbol{x}$ contribution with $\mathbf{A}_K$

  for $i, j \in \text{DOFs}_{\hat{K}}$ **do**

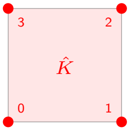
    $I = \text{DOFs}_K[i]$, $J = \text{DOFs}_K[j]$

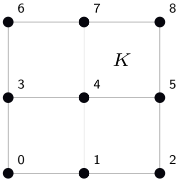
    $\boldsymbol{z}_I \leftarrow \boldsymbol{z}_I + (\mathbf{A}_K)_{i,j} \cdot \boldsymbol{x}_J$

  end

end

return $\boldsymbol{z}$





Assembly Algorithm (`evaluateLoadVector`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

    `// 1. Compute the Element vector` $\boldsymbol{b}_K$

    $\text{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\text{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

    for $i \in \text{DOFs}_{\hat{K}}$ **do**

        $I = \text{DOFs}_K[i]$

        $$(\mathbf{b}_K)_i = \int_K f(x) b_K^I(x)\, dx$$

    end

    `// 2. Compute global right-hand side vector` $\boldsymbol{b}$ `contribution`

    for $i \in \text{DOFs}_{\hat{K}}$ **do**

        $I = \text{DOFs}_K[i]$,

        $\boldsymbol{b}_I \leftarrow \boldsymbol{b}_I + (\mathbf{b}_K)_i$

    end

end

return $\underline{\boldsymbol{z}}$

Assembly Algorithm (`evaluateLoadVector`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element vector` $\boldsymbol{b}_K$

  $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

    $I = \mathsf{DOFs}_K[i]$

$$(\mathbf{b}_K)_i = \int_{\hat{K}} \underbrace{(\boldsymbol{\Phi}_K^* f)(\hat{x})}_{=f(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))} \underbrace{(\boldsymbol{\Phi}_K^* b_K^I)(\hat{x})}_{=\hat{b}^i(\hat{x})} |\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{x})| \, d\hat{x}$$

  end

  `// 2. Compute global right-hand side vector` $\boldsymbol{b}$ `contribution`

  for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

    $I = \mathsf{DOFs}_K[i]$,

    $\boldsymbol{b}_I \leftarrow \boldsymbol{b}_I + (\mathbf{b}_K)_i$

  end

end

return $\underline{\boldsymbol{z}}$

Assembly Algorithm (`evaluateLoadVector`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

  `// 1. Compute the Element vector` $\boldsymbol{b}_K$

  $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

  for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

    $$(\mathbf{b}_K)_i = \int_{\hat{K}} f(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))\hat{b}^i(\hat{x}) |\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{x})| \, d\hat{x}$$

  end

  `// 2. Compute global right-hand side vector` $\boldsymbol{b}$ `contribution`

  for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

    $I = \mathsf{DOFs}_K[i]$,

    $\boldsymbol{b}_I \leftarrow \boldsymbol{b}_I + (\mathbf{b}_K)_i$

  end

end

return $\underline{\boldsymbol{z}}$

Assembly Algorithm (`evaluateLoadVector`)

Input: $\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \mathbf{0}$

for Element K in Mesh **do**

    // 1. Compute the Element vector $\boldsymbol{b}_K$

    $\mathsf{DOFs}_K \leftarrow \{4, 5, 8, 7\}$, $\mathsf{DOFs}_{\hat{K}} \leftarrow \{0, 1, 2, 3\}$

    for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

$$(\mathbf{b}_K)_i \approx \sum_{k}^{N_{\mathsf{Int}}} \hat{\omega}_k f(\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}))\hat{b}^i(\hat{\boldsymbol{q}}_k)|\det \mathbf{D}\boldsymbol{\Phi}_K(\hat{\boldsymbol{q}}_k)|$$

    end

    // 2. Compute global right-hand side vector $\boldsymbol{b}$ contribution

    for $i \in \mathsf{DOFs}_{\hat{K}}$ **do**

        $I = \mathsf{DOFs}_K[i]$,

        $\boldsymbol{b}_I \leftarrow \boldsymbol{b}_I + (\mathbf{b}_K)_i$

    end

end

return $\underline{\boldsymbol{z}}$

Outline

Quadrature

- Quadrature rule: Approximation of the definite integral of a function.
- Midpoint rule (Rectangle rule): $\int_a^b f(x)\ dx \approx (b-a)\, f\left(\frac{a+b}{2}\right)$

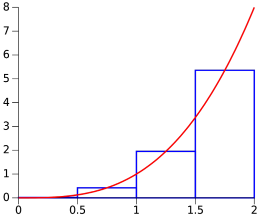


Figure: Mid-Riemann sum[1]

[1] By Qef - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=7081805

Gaussian Quadrature

n-point Gaussian quadrature rule yields exact results for polynomials of degree $2n-1$ or less with a suitable choice of nodes x_i and weights w_i, for $1 < i < n$.

$$\int_{-1}^{1} f(x)\, dx \approx \sum_{i=1}^{n} w_i\, f(x_i)$$

Examples:

- Gauss-Legendre Quadrature (x_i are the roots of the Legendre polynomial)
- Chebychev-Gauss Quadrature (x_i are the Chebychev nodes)
- Gauss-Jacobi Quadrature (x_i are the roots of the Jacobi polynomial)

Gauss-Jacobi quadrature

Jacobi polynomials are hypergeometric (a class of classical orthogonal polynomials).
Approximate integrals of the form:

$$\int_{-1}^{1} f(x)(1-x)^{\alpha}(1+x)^{\beta}\ dx$$

with $\alpha, \beta \geq -1$.
The following are special cases of the Gauss-Jacobi quadrature rule:

- Gauss-Legendre quadrature with $\alpha = \beta = 0$
- Chebychev-gauss quadrature with $|\alpha| = |\beta| = \frac{1}{2}$
- Gauss-Gegenbauer quadrature with $\alpha = \beta$

Orthogonal Polynomials and their Recurrence relation

- Recurrence relation: Equation that states that the n-th term of a sequence is equal to some combination of the previous terms.
  Example:

Orthogonal Polynomials and their Recurrence relation

- Recurrence relation: Equation that states that the n-th term of a sequence is equal to some combination of the previous terms.
  Example: Fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$

Orthogonal Polynomials and their Recurrence relation

- Recurrence relation: Equation that states that the n-th term of a sequence is equal to some combination of the previous terms. Example: Fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$
- Orthogonal Polynomials: Sequence of polynomials such that any two different polynomials in the sequence are orthogonal under some inner product.
- Recurrence relation of classical orthogonal polynomials:

$$P_n(x) = (A_n x + B_n)P_{n-1}(x) + C_n P_{n-2}(x)$$

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

All use an iterative algorithm computing the zeros of the polynomials using the recurrence relation and Newton's method:

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

All use an iterative algorithm computing the zeros of the polynomials using the recurrence relation and Newton's method:

1. Loop over the number of points/roots to compute.

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

All use an iterative algorithm computing the zeros of the polynomials using the recurrence relation and Newton's method:

1. Loop over the number of points/roots to compute.
2. For each one, make an (educated) initial guess, then apply Newton's method to find the root.

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

All use an iterative algorithm computing the zeros of the polynomials using the recurrence relation and Newton's method:

1. Loop over the number of points/roots to compute.
2. For each one, make an (educated) initial guess, then apply Newton's method to find the root.
3. With the root, compute the weight using the gamma function.

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Gauss-Jacobi Implementation: Algorithm

Compared implementations of GSL[2], deal.II[3], LehrFEM++[4]

All use an iterative algorithm computing the zeros of the polynomials using the recurrence relation and Newton's method:

1. Loop over the number of points/roots to compute.
2. For each one, make an (educated) initial guess, then apply Newton's method to find the root.
3. With the root, compute the weight using the gamma function.

This algorithm follows the implementation in LehrFEM++

[2] https://www.gnu.org/software/gsl/doc/html/integration.html
[3] https://www.dealii.org/developer/doxygen/deal.II/polynomial_8h_source.html
[4] https://github.com/craffael/lehrfempp/blob/master/lib/lf/quad/gauss_quadrature.cc

Quadrature Implementation: Tensor Product

Tensor products are used to get the quadrature nodes and weights for a reference element.

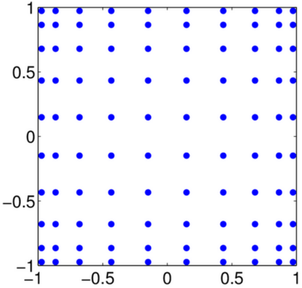


Figure: Gauss-Legendre quadrature 2D tensor product[5]

[5]Waves in Spatially-Disordered Neural Fields: A Case Study in Uncertainty Quantification

Outline

Plan for the (near) Future

- Finish Building Blocks: 3D
- FEMPoissonSolver

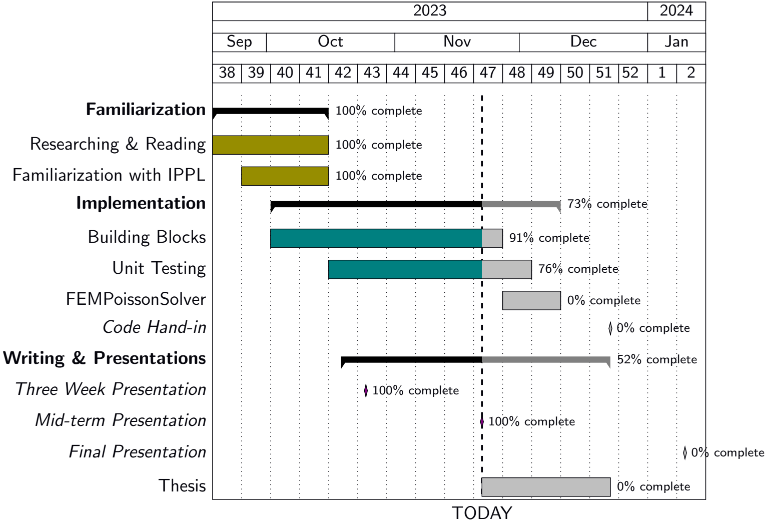
Plan for the (near) Future

- Finish Building Blocks: 3D
- FEMPoissonSolver
- Writing
- Unit testing: Gauss Jacobi, 3D implementations
- Documentation

Plan for the (near) Future

- Finish Building Blocks: 3D
- FEMPoissonSolver
- Writing
- Unit testing: Gauss Jacobi, 3D implementations
- Documentation
- Higher-order (equidistant) Lagrange

Updated Timeline



Appendix A: Global $\leftrightarrow$ Local Affine Transformations

- Local to Global: $\boldsymbol{x} = \boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}) = \mathbf{F}_K\hat{\boldsymbol{x}} + \boldsymbol{\tau}_K$
- Global to Local: $\hat{\boldsymbol{x}} = \boldsymbol{\Phi}_K^{-1}(\boldsymbol{x}) = \mathbf{F}_K^{-1}(\boldsymbol{x} - \boldsymbol{\tau}_K)$