HELMHOLTZ GEMEINSCHAFT

EUROfusion

IPP

Partly supported by SPPEXA (Software for exascale computing), project ExaDG

Matrix-free finite elements in `deal.II` with applications to the time-dependent Schrödinger equation

Katharina Kormann
Max Planck Institute for Plasma Physics
Technical University of Munich
Joint work with Martin Kronbichler

October 4-5, 2017

Outline

Operator evaluation framework

Motivation

Bottlenecks for finite elements based on sparse-matrices:

- Memory-limited, especially for high order elements (Order 6 elements in 3D between 343 and 2197 elements per line).
- Low arithmetic intensity.

Goal: change algorithm towards higher arithmetic intensity $\rightarrow$ better performance

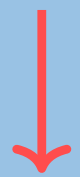
Matrix-free alternative based on elementwise integration for finite elements.

Matrix-free algorithms for higher order elements

Matrix-vector product

matrix-based:

$$\begin{cases} A = \sum_{K \in \{\text{cells}\}} P_K^T A_K P_K \quad \text{(assembly)} \\ v = Au \end{cases}$$



matrix-free:

$$v = \sum_{K \in \{\text{cells}\}} P_K^T A_K (P_K u)$$

Matrix-free algorithm:

- $v = 0$
- loop over cells
  - (i) Extract local vector values on cell: $u_K = P_K u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_K$ (**without** forming A_K)
  - (iii) Sum results from (ii) into the global solution vector: $v = v + P_K^T v_K$

Kronbichler, Kormann, A generic interface for parallel finite element operator application. *Comput. Fluids* 63:135–147 (2012)

Cell matrix-vector product: quadrature

Contribution of cell K to matrix-vector product for finite element Laplacian

$$(A_K u_K)_j = \int_K \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h d\mathbf{x} \approx \sum_q w_q \det J_q \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h \Big|_{\mathbf{x}=\mathbf{x}_q}$$

$$= \sum_q \nabla_\xi \phi_j J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}} \sum_i \nabla_\xi \phi_i u_{K,i} \Big|_{\xi=\xi_q}, \quad j = 1, \ldots, \text{cell\_dofs}$$

(a) Compute unit cell gradients $\nabla_\xi u^h = \sum (\nabla_\xi \phi_i) u_{K,i}$ on all quadrature points

(b) On each quadrature point, apply geometry $J_q^{-\mathsf{T}}$, multiply by quadrature weight and Jacobian determinant, apply geometry for test function J_q^{-1}

(c) Test by unit cell gradients of all basis functions and sum over quadrature points

Matrix notation:

$$v_K = A_K u_K$$
$$= S^{\mathsf{T}} W S u_K$$

with

$S_{qi} = \nabla_\xi \phi_i \big|_{\xi_q}$

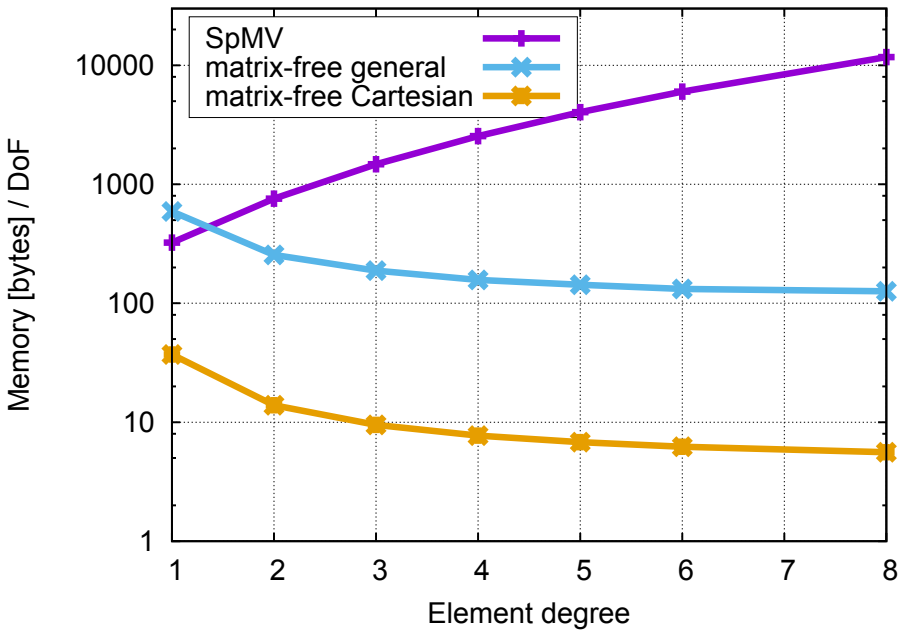
$W_{qq} = J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}}$

Memory considerations

Idea

- Matrix-free implementation reduces memory transfer
- Trade memory transfer for computations $\rightarrow$ faster?
- Naive implementation (standard abstraction in FE libraries such as deal.II, FEniCS, etc.): Dense matrix for all basis functions, $n^2_{\text{cell\_dofs}}$ operations

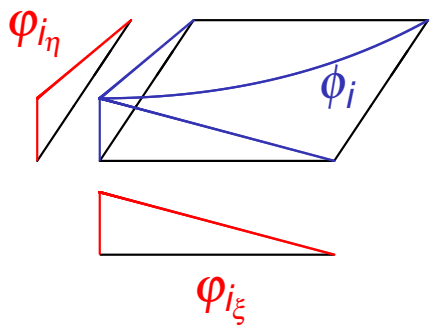
Memory per degree of freedom ($\approx$ main memory transfer)



- Bottlenecked by step (a) (unit cell gradients) and step (c) (testing and summation of quadrature points)
- In 3D: 6 to 48 times as many operations as sparse matrix
- Computations might be cheap, but they are **not for free**!

Fast evaluation of basis functions

- Tensor product form of basis functions:
  $\phi(\xi,\eta) = \varphi(\xi)\varphi(\eta)$
- Evaluate shape values on quadrature points



$$\frac{\partial u^h}{\partial \xi}(\xi_q,\eta_q) = \sum_{i \in \text{cell\_dofs}} u^{(i)} \frac{\partial \phi_i(\xi_q,\eta_q)}{\partial \xi}$$

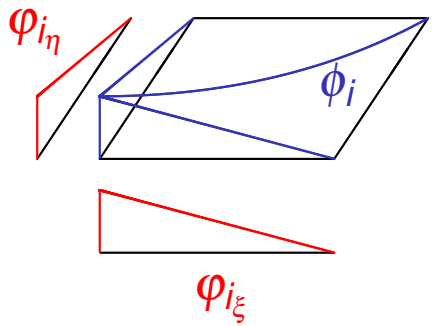
$$= \sum_{i_\xi}\sum_{i_\eta} u^{(i_\xi,i_\eta)} \varphi_{i_\xi}(y_\eta) \frac{\varphi_{i_\xi}(\xi_q)}{\partial \xi}$$

Fast evaluation of basis functions

- Tensor product form of basis functions:
  $\phi(\xi,\eta) = \varphi(\xi)\varphi(\eta)$
- Evaluate shape values on quadrature points

$$\frac{\partial u^h}{\partial \xi}(\xi_q,\eta_q) = \sum_{i \in \text{cell\_dofs}} u^{(i)} \frac{\partial \phi_i(\xi_q,\eta_q)}{\partial \xi}$$

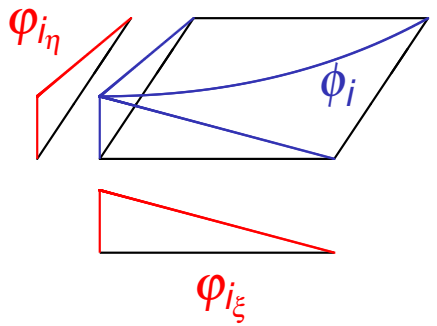
$$= \sum_{i_\xi} \sum_{i_\eta} u^{(i_\xi,i_\eta)} \varphi_{i_\xi}(y_\eta) \frac{\varphi_{i_\xi}(\xi_q)}{\partial \xi}$$



Matrix notation:
Form of shape matrix
$S = D_\xi \otimes S_\eta$

Fast evaluation of basis functions

- Tensor product form of basis functions:
  $\phi(\xi,\eta) = \varphi(\xi)\varphi(\eta)$
- Evaluate shape values on quadrature points



$$\frac{\partial u^h}{\partial \xi}(\xi_q,\eta_q) = \sum_{i \in \text{cell\_dofs}} u^{(i)} \frac{\partial \phi_i(\xi_q,\eta_q)}{\partial \xi}$$

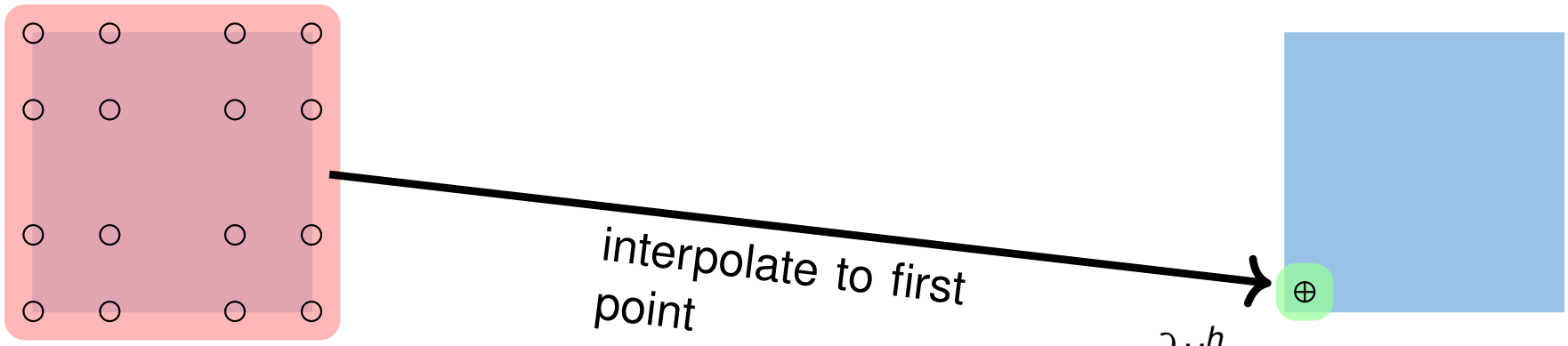
$$= \sum_{i_\xi}\sum_{i_\eta} u^{(i_\xi,i_\eta)} \varphi_{i_\xi}(y_\eta) \frac{\varphi_{i_\xi}(\xi_q)}{\partial \xi}$$

Matrix notation:
Form of shape matrix
$S = D_\xi \otimes S_\eta$

$\frac{\partial}{\partial \xi} u^h(x_q,y_q)\big|_{\text{q\_points}} = (D_\xi \otimes S_\eta)\mathbf{u}_K$
Implemented as dense matrix-matrix multiplication
$S_\eta \mathbf{U}_K D_\xi^\mathsf{T}$

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels

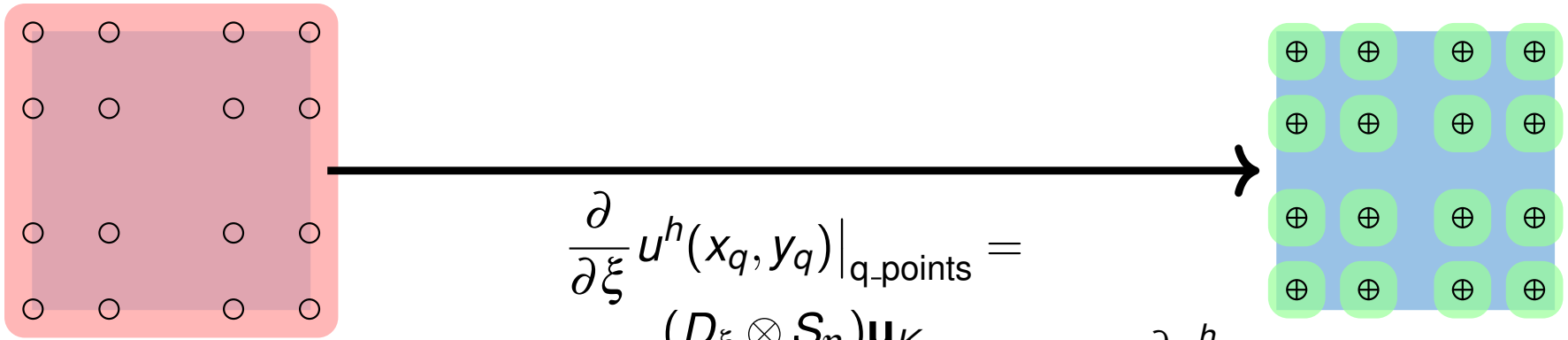


Vector values u_K on nodes

$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



Vector values u_K on nodes

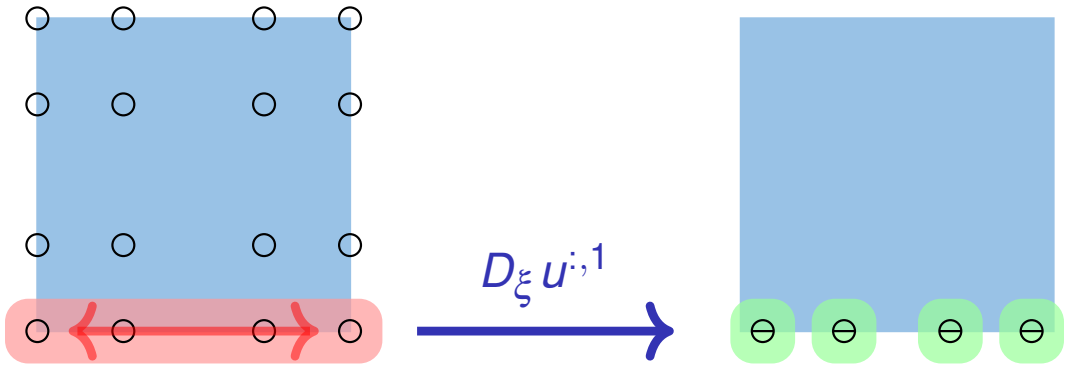
$$\frac{\partial}{\partial \xi} u^h(x_q, y_q)\big|_{\text{q\_points}} = (D_\xi \otimes S_\eta)\mathbf{u}_K$$

all points: $4^2 \times 4^2$ ops

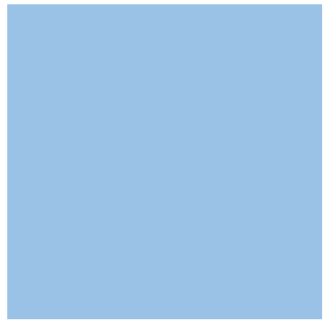
$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



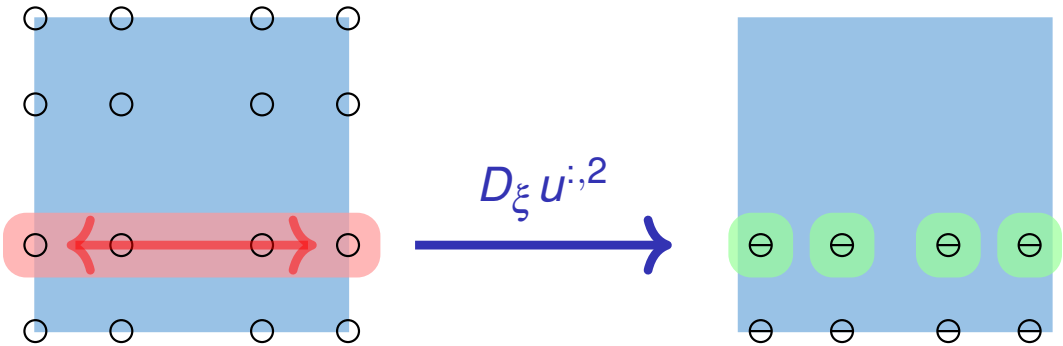
Vector values u_K on nodes



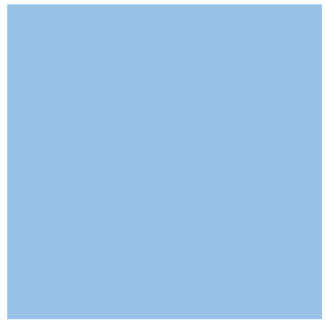
$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



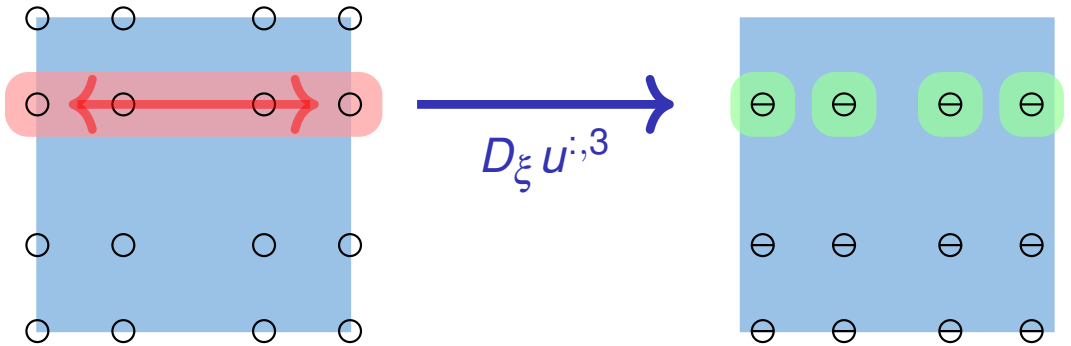
Vector values u_K on nodes



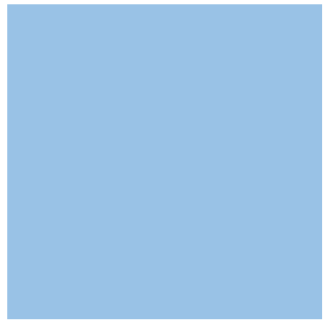
$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



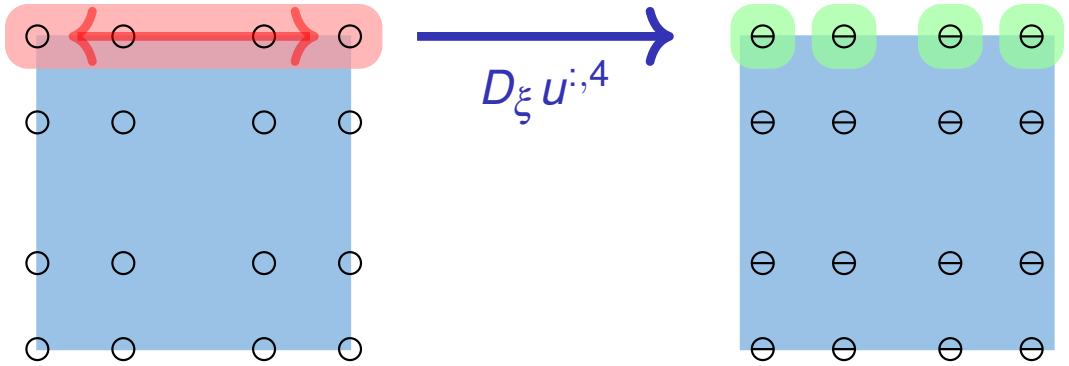
Vector values u_K on nodes



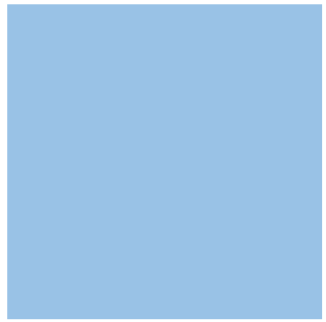
$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



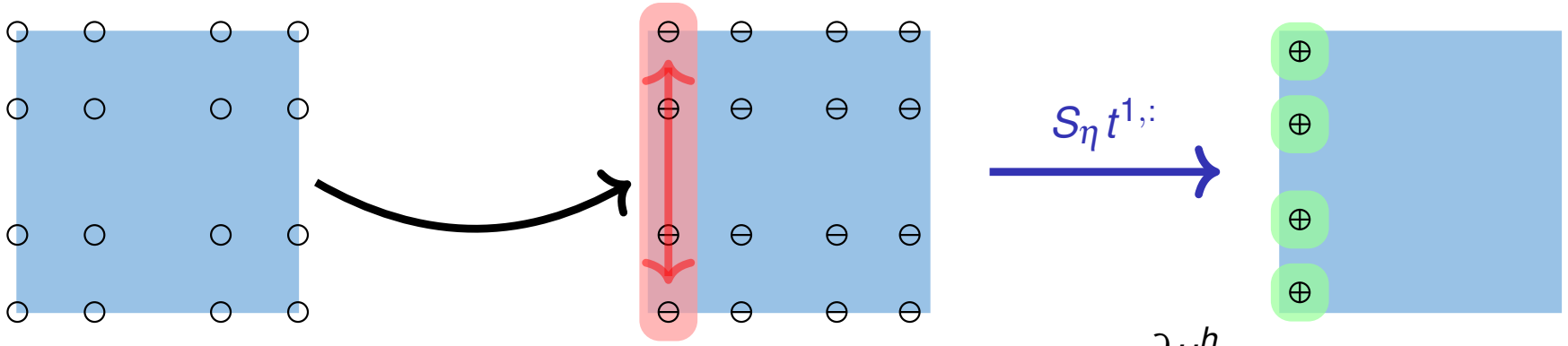
Vector values u_K on nodes



$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels

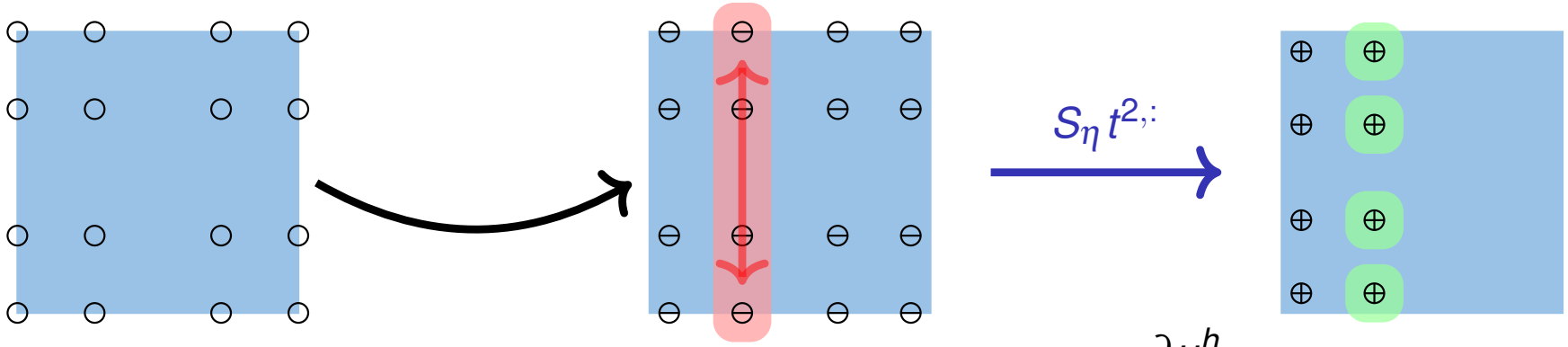


Vector values u_K on nodes

$\dfrac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels

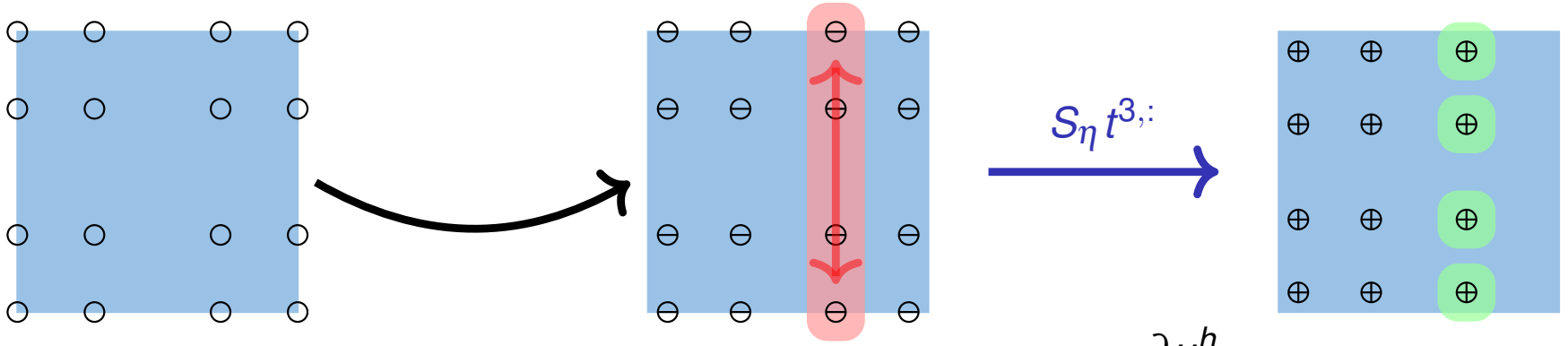


Vector values u_K on nodes

$\dfrac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels

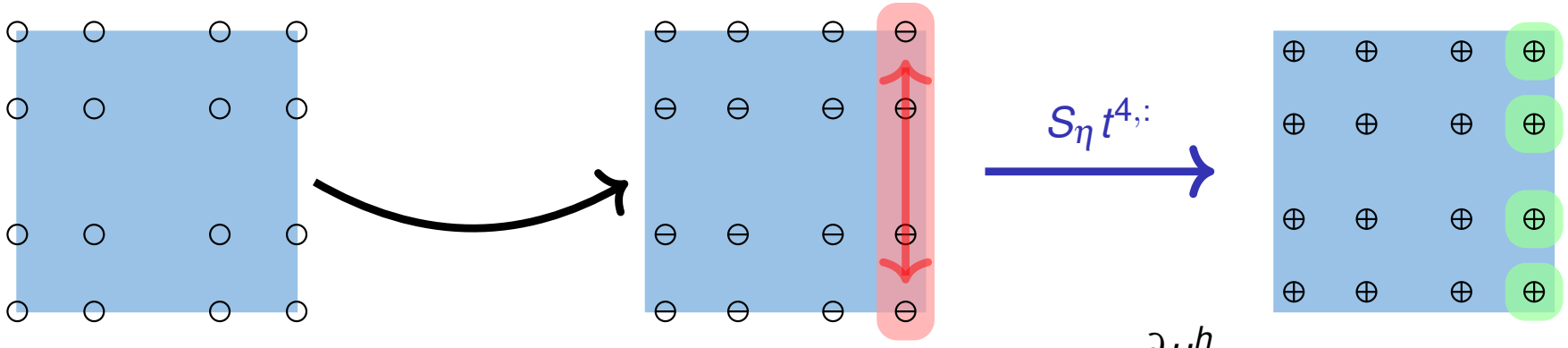


Vector values u_K on nodes

$\dfrac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels

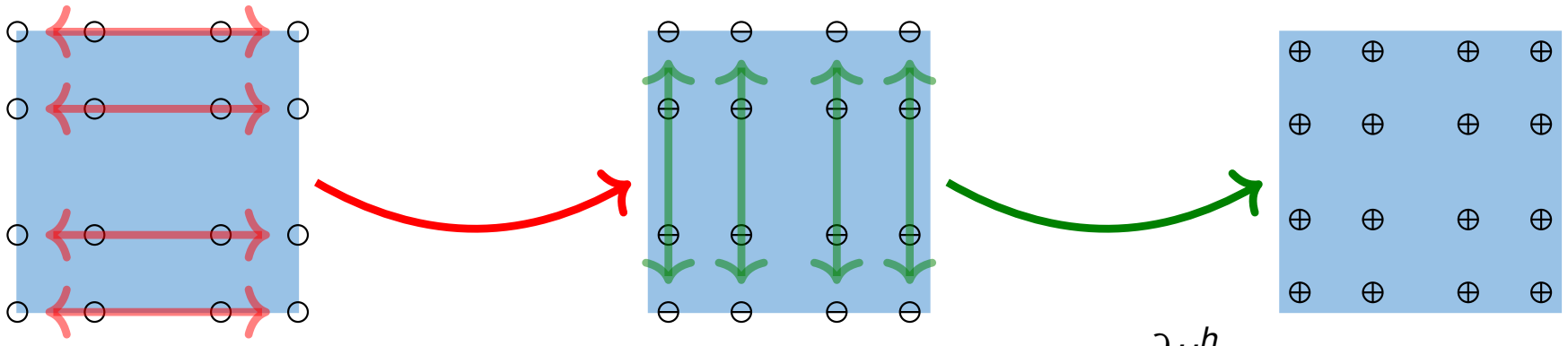


Vector values u_K on nodes

$\dfrac{\partial u^h}{\partial \xi}$ on quadrature points

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



Vector values u_K on nodes

$\frac{\partial u^h}{\partial \xi}$ on quadrature points

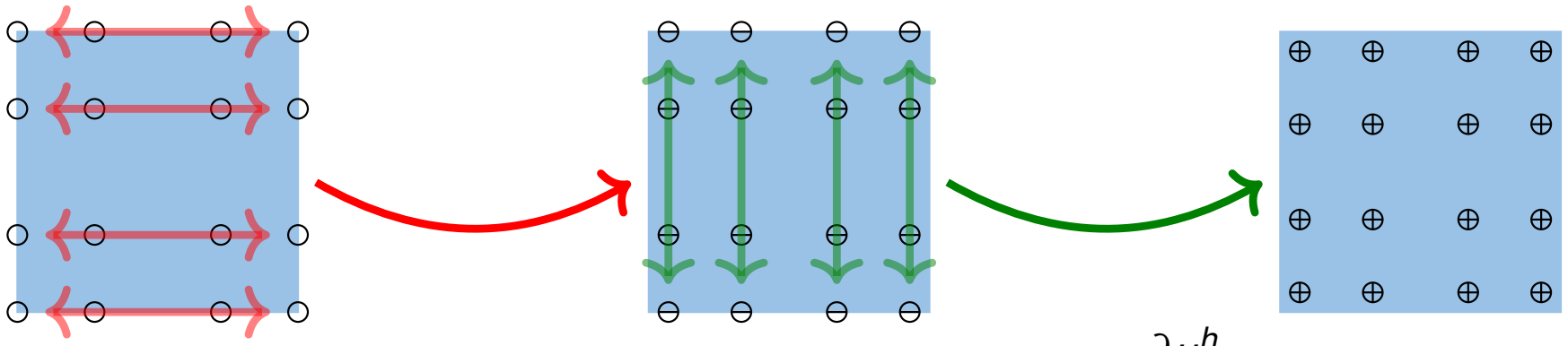
Tensor-based evaluation reduces evaluation cost from 4^4 to 2×4^3

In general for degree k and dimension d: $\mathscr{O}((k+1)^{2d})$ to $\mathscr{O}(d(k+1)^{d+1})$

Example: $d = 3$, $k = 5$: **46,456** to **3888**

Exploit tensor product structure (sum factorization)

Illustration on $\mathscr{Q}_3$ (Lagrange basis): successively apply 1D kernels



Vector values u_K on nodes

$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Tensor-based evaluation reduces evaluation cost from 4^4 to 2×4^3

In general for degree k and dimension d: $\mathscr{O}((k+1)^{2d})$ to $\mathscr{O}(d(k+1)^{d+1})$

Example: $d = 3$, $k = 5$: **46,456** to **3888**

Similarly for $\frac{\partial}{\partial \eta}$

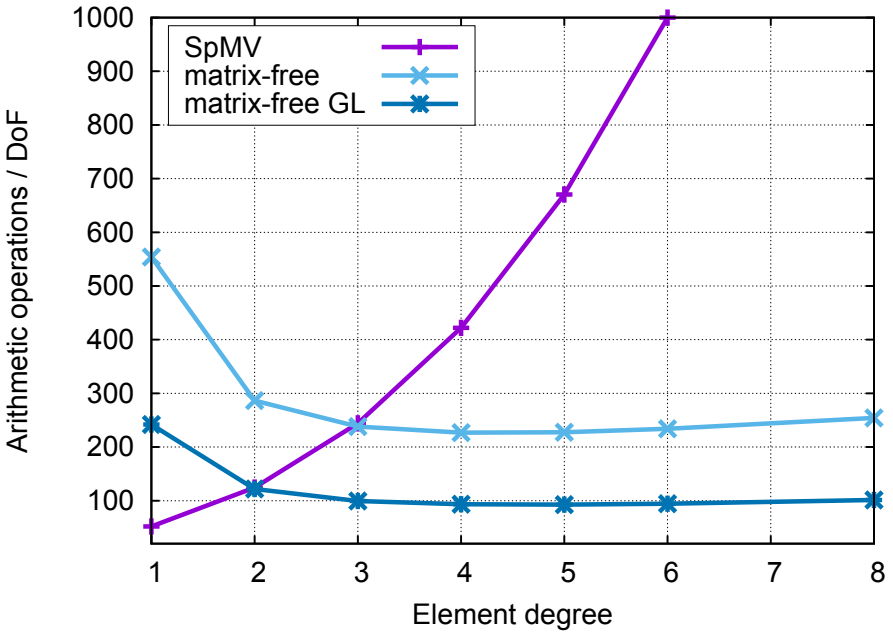
$$\frac{\partial u(\xi_q, \eta_q)}{\partial \eta}\Big|_{\text{q\_points}} = (S_\xi \otimes D_\eta)\mathbf{u}_K \simeq D_\eta U_K S_\xi^\mathsf{T}$$

and integration routines

Impact of sum factorization

- Very competitive for high orders
- Algorithms from spectral elements (1980s)
- Optimizations possible for special shape functions (picture: Gauss–Lobatto case, interpolation is identity operation)
- Previously only used for high orders $k \geq 5$
- Combination with memory transfer not considered
- **Speedup for $\mathcal{Q}_2$ and $\mathcal{Q}_3$?** Trade memory transfer for arithmetics

Operation count per degree of freedom (arithmetics)



3D Laplacian, hexahedral elements $\mathcal{Q}_k$, periodic b.c. (= no boundary), pre-evaluated Jacobians on all quadrature points, "even-odd decomposition" to decrease work for 1D kernels from $(k+1)^2$ to $(k+1)(\lfloor \frac{k}{2} \rfloor + 2)$

Code example: Cell term for Laplacian

Evaluation of weak form

$$(\nabla v, \nabla u)_{\Omega_h}$$

```
void cell(MatrixFree<dim> &data,
          Vector &dst,
          const Vector &src,
          const std::pair<unsigned int,unsigned int> &range)
{
  FEEvaluation<dim,degree> phi (data);
  for (unsigned int cell=range.first; cell<range.second; ++cell)
    {
      phi.reinit (cell);                          // set pointers to data
      phi.read_dof_values(src);                   // read from source
      phi.evaluate (/*values=*/ false,            // sum factorization
                    /*gradients=*/ true);
      for (unsigned int q=0; q<phi.n_q_points; ++q)
        phi.submit_gradient (phi.get_gradient(q), q); // geometry
      phi.integrate (/*values=*/ false,           // sum factorization
                     /*gradients=*/ true);
      phi.distribute_local_to_global (dst); // sum into destination
    }
}
```

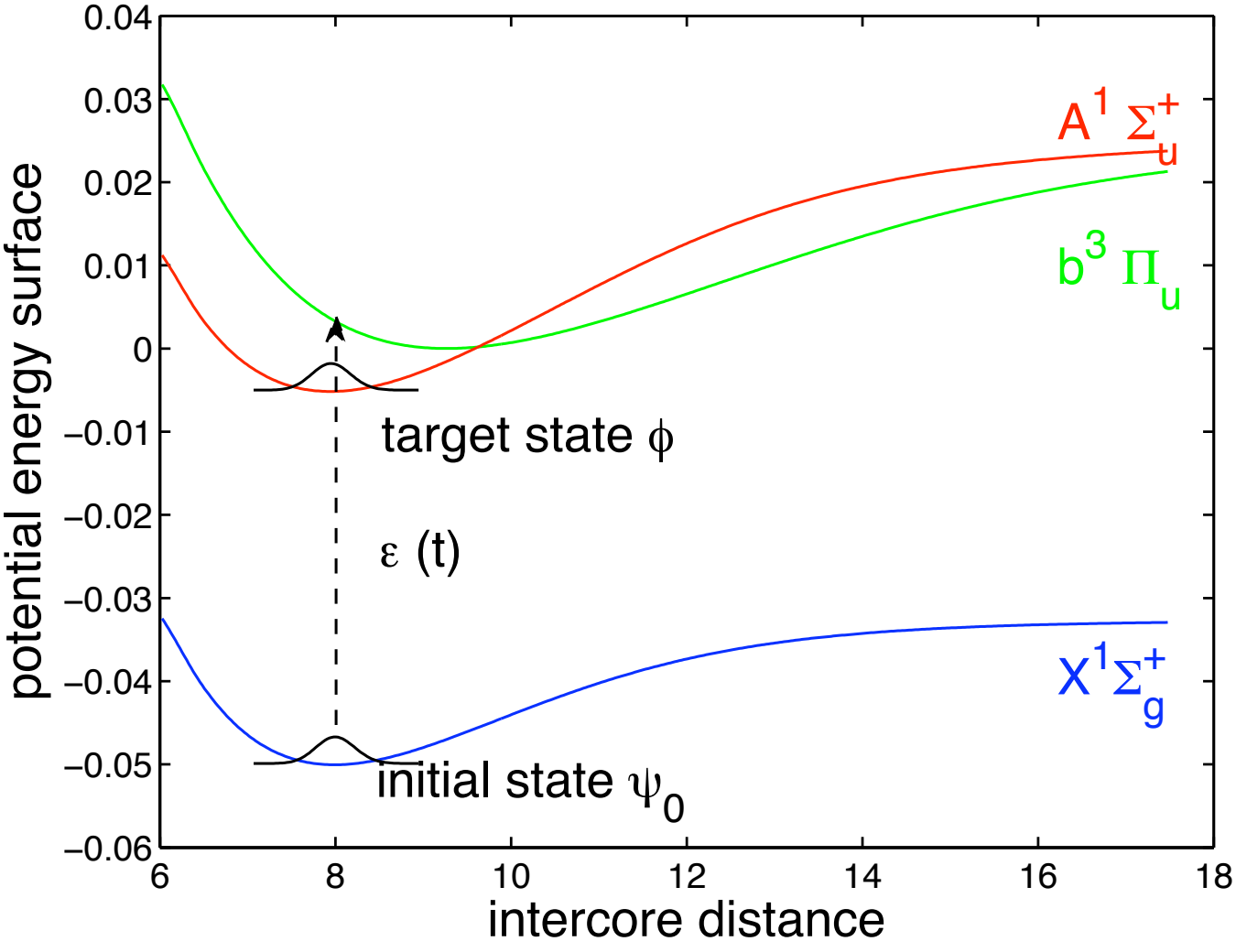
Outline

Operator evaluation framework

Time-dependent Schrödinger Equation

- Model problem
- Finite element discretization
- Time propagation
- Error Control
- Example program

Exitation by lasers



Time-Dependent Schrödinger Equation

$$\mathrm{i}\hbar\frac{\partial}{\partial t}\psi(r,t) = \hat{H}(r,\Delta r,t)\psi(r,t), \qquad \psi(r,0) = \psi_0(r).$$

Hamiltonian for three-state system: $\hat{H} = -\frac{1}{2m}\Delta + V$ with

$$V = \begin{pmatrix} V_g(r) & \mu(r)V_1(t) & 0 \\ \mu(r)V_1(t) & V_{e1}(r) & V_2(r) \\ 0 & V_2(r) & V_{e2}(r) \end{pmatrix}.$$

Example program: Two coupled harmonic oscillators

Schrödinger equation:

$$\mathrm{i}\hbar\partial_t \begin{pmatrix} \psi_a \\ \psi_b \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}\Delta + \frac{1}{2}\|x\|^2 & \mathrm{e}^{2(t-0.5)^2}\cos(t-0.5) \\ \mathrm{e}^{2(t-0.5)^2}\cos(t-0.5) & -\frac{1}{2}\Delta + \frac{1}{2}\|x\|^2 + 1 \end{pmatrix} \begin{pmatrix} \psi_a \\ \psi_b \end{pmatrix}$$

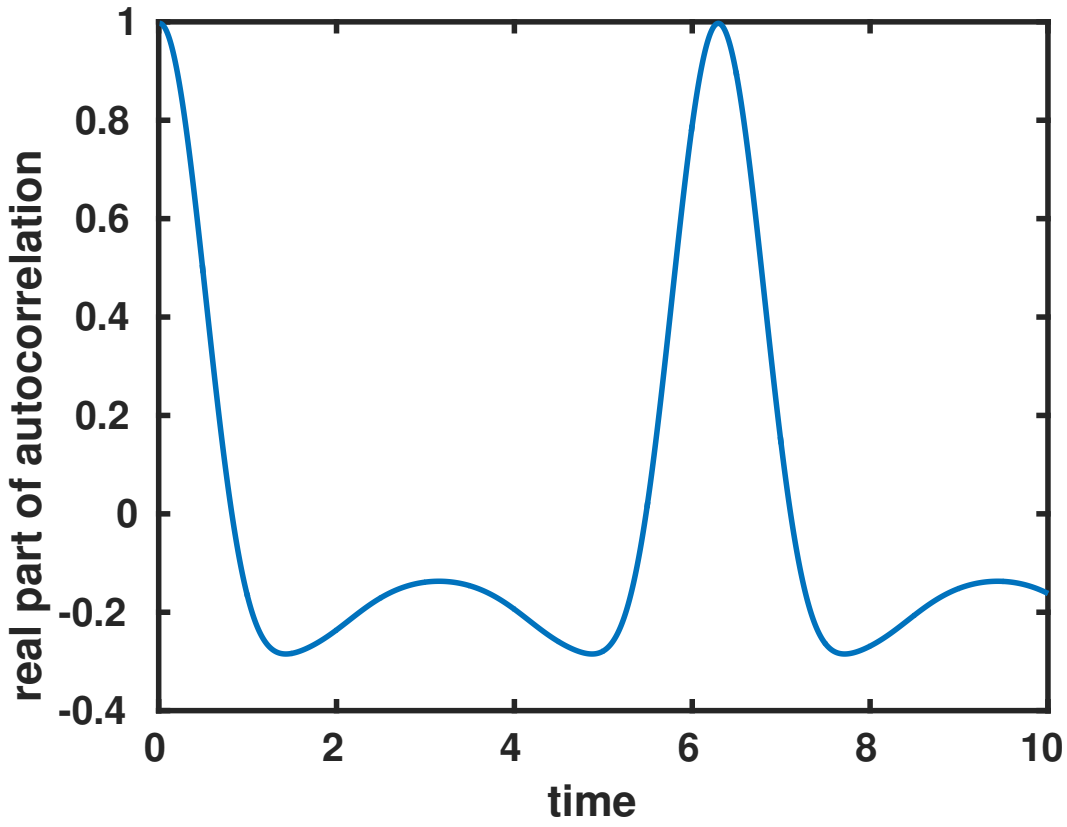
Initial value: $\psi_0(x) = \frac{1}{\pi^{d/4}}\mathrm{e}^{-0.5\|x-x_0\|_2^2}$.

Solution of the uncoupled harmonic oscillator:
$\psi(x,t) = \frac{1}{\pi^{d/4}}\mathrm{e}^{\mathrm{i}\gamma_t}\mathrm{e}^{-\mathrm{i}\sin(t)x_0^T(x-x_t)}\mathrm{e}^{-\frac{1}{2}\|x-x_0\|_2^2}$ with $x_t = x_0\cos(t)$ and $\gamma_t = -dt - x_0^T x_t$.

Implementation of test case data: `initial_ho.h`.
Main program: `schrodinger_ho.cc`.

Autocorrelation

Autocorrelation of pure harmonic oscillator example



Postprocessing script to plot autocorrelation:

`diagnostics/auto_ho.py`.

Spatial discretization

Spatial discretization with finite elements:

$$M\frac{\mathrm{d}}{\mathrm{d}t}u = Su. \tag{1}$$

Gauss–Lobatto finite elements:

- Nodal basis with nodes of Gauss–Lobatto quadrature points used as basis centers.
- Approximative quadrature using the same Gauss–Lobatto quadrature rule to compute integrals.
- Mass matrix becomes diagonal (*mass lumping*) which yields an explicit system.

Implementation of multi-state Hamiltonian

After applying the Laplacian to each state, the following code describes the application of the multi-state Hamiltonian. The potential terms are diagonal due to the lumping.

```
void operator() () const
  {
    const unsigned int size = dst(0).size();
    for(unsigned int j=0;j<size;++j)
      for(unsigned int k = 0;k<2*no_states;++k)
        dst[k].local_element(j) = dst[k].local_element(j)
          *hamiltonian_data.mass_factor*
          hamiltonian_data.system_lmm_inv.local_element(j)+
          src[k].local_element(j)*
          hamiltonian_data.potential[k/2].local_element(j)
          +src[(k+no_states)%(2*no_states)].local_element(j)*
          hamiltonian_data.pulse;
  }
```

Implementation of the matrix-free Hamiltonian: `matrix_free_ho.h`.

Exponential Propagator

Evolution operator for time-independent Hamiltonian:

$$U(t_0, t_f) = \exp\left(-\frac{i}{\hbar} H(t_f - t_0)\right)$$

Approximation for time-dependent case:

$$\psi(r, t + \Delta t) = \exp\left(-\frac{i}{\hbar} \overline{H} \Delta t\right) \psi(t)$$

Two questions:

- How to choose $\overline{H}$?
  Magnus expansion
- How to compute the matrix exponential efficiently?
  Lanczos algorithm

Lanczos algorithm

Let $A \in \mathbb{C}^{n\times n}$ be hermitian and $q \in \mathbb{C}^n$. The **Lanczos algorithm** computes in $m+1$ steps a orthonormal basis Q_m of the Krylov subspace $\text{span}\{q, Aq, \ldots, A^m q\}$ and a tridiagonal matrix

$$A_m = \begin{pmatrix} \alpha_0 & \beta_0 & 0 & \ldots & 0 \\ \beta_0 & \alpha_1 & \beta_1 & 0 & \\ 0 & \ddots & \ddots & \ddots & 0 \\ & 0 & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ 0 & \ldots & 0 & \beta_{m-1} & \alpha_m \end{pmatrix}$$

such that $AQ_m = Q_m A_m + h_{m+1,m} q_{m+1} e_m^T$.

Matrix exponential with Lanczos

- Compute Krylov subspace induced by solution at previous time step: $q = u(t_{n-1})$.
- Use Krylov approximation of the matrix to compute matrix exponential:

$$\exp(\Delta t A)u(t_{n-1}) \approx \exp(\Delta t Q_m A_m Q_m^H)u(t_{n-1}) = Q_m \exp(\Delta t A_m)e_1.$$

- Solve eigenvalue problem for tridiagonal matrix to compute $\exp(\Delta t A_m)$.
- **Error estimate**: $\Delta t \beta_m |\exp(\Delta t A_m)|_{m+1,1}$ [1]

Implementation: `expo_int.h`.

[1] Hochbruck, Lubich, Selhofer: SIAM J. Sci.Comput. 19, 1998.

Magnus expansion

$$U(t_{n+1}, t_n) = \mathrm{e}^{\sum_{k=1}^{\infty} A_k(t_{n+1}, t_n)},$$

where

$$\begin{aligned} A_1(t_{n+1}, t_n) &= -\frac{\mathrm{i}}{\hbar} \int_{t_n}^{t_{n+1}} H(\tau_1)\mathrm{d}\tau_1 \\ A_2(t_{n+1}, t_n) &= -\frac{1}{2}\left(\frac{\mathrm{i}}{\hbar}\right)^2 \int_{t_n}^{t_{n+1}} \int_{t_n}^{\tau_1} [H(\tau_1), H(\tau_2)]\mathrm{d}\tau_2\mathrm{d}\tau_1 \\ &\dots \end{aligned}$$

Truncate for numerical propagation since

$$A_k = \mathscr{O}\left((\Delta t)^{2k-1}\right)$$

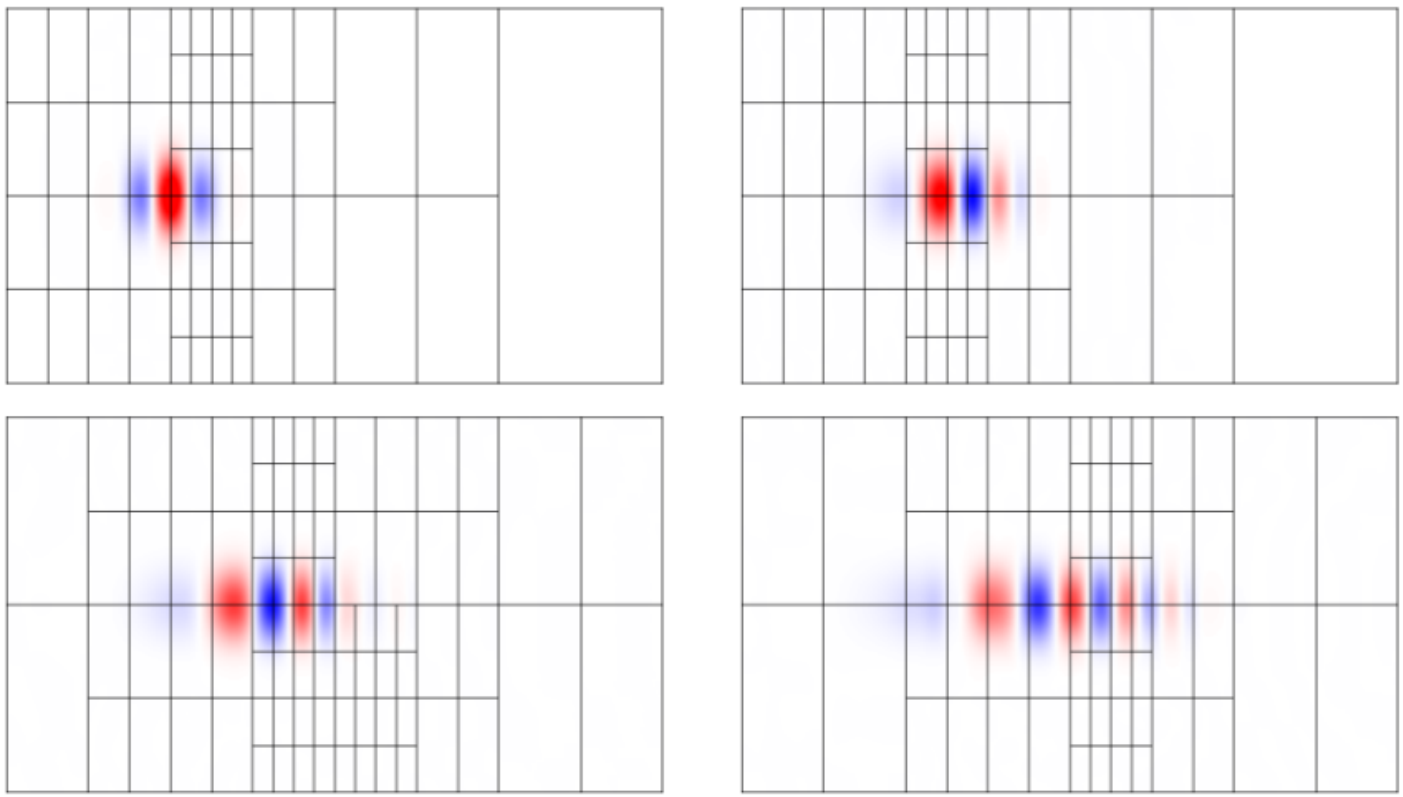
Adaptive Magnus–Lanczos: Idea

- Choose **step size** according to error in the **Magnus expansion**
- Choose **size of the Krylov space** according to error in the **Lanczos algorithm**[1]
- Use **a posteriori estimate**[2] to **connect local and global error**
- Use **special properties** of the Schrödinger equation to **avoid solving the dual problem**

References:
[1] M. Hochbruck, C. Lubich, H. Selhofer, SIAM J. Sci. Compt. **19**, 1552 (1999)
[2] Y. Cao, L. Petzold, SIAM J. Sci. Compt. **26**, 359 (2004)

Mesh Refinement

A posteriori error estimate can also be derived for spatial adaptivity.
Example 1: Dynamical mesh.

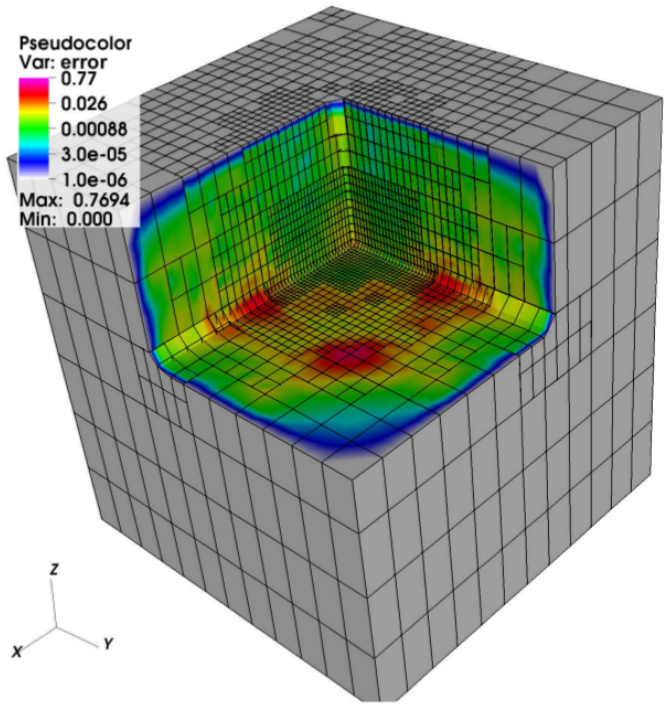


Reference: Kormann, A time-space adaptive method for the Schrödinger equation, Commun. Comput. Phys., 20:60–85, 2016.

Mesh Refinement

A posteriori error estimate can also be derived for spatial adaptivity.

Example 2: OClO example: Error control for target state.



Reference: Kormann, A time-space adaptive method for the Schrödinger equation, Commun. Comput. Phys., 20:60–85, 2016.

Test code

- Test code for coupled harmonic oscillators in folder `schrodinger_fem` (can be downloaded from `https://gitlab.lrz.de/ne65nib/schrodinger_fem`).
- Build the code and run it with:
  `mpirun -n 1 ./schrodinger_ho schrodinger_ho.prm`
- Use the python script `auto_ho.py` to visualize the autocorrelation. In the folder diagnostics type: `python auto_ho.py`
- Modify the parameter file.
  - Modify the coupling strength and check the effect on the peak of the autocorrelation function.
  - Modify the number of Gauss–Lobatto points and the grid refinement and checkt the quality of the solution.