Fast and smooth simulation of space-time problems

Day 3

Department of Computer Science
AGH University of Science and Technology, Kraków, Poland
home.agh.edu.pl/paszynsk

Outline

- Isogeometric finite element method
- Alternating Directions Implicit (ADI) method
- Isogeometric L2 projections
- Explicit dynamics
- Example 1: Heat transfer
- Installation of IGA-ADS solver
- **Parallel shared memory explicit dynamics**
- **Example 2: Non-linear flow in heterogenous media**
- Parallel distributed memory explicit dynamics
- Example 3: Linear elasticity
- Implicit dynamics
- Example 4: Implicit heat transfer
- Example 5: Pollution problem
- Labs with implict dynamics

Software

Program Title: IGA-ADS
Code: `git clone https://github.com/marcinlos/iga-ads`
Licensing provisions: MIT license (MIT)
Programming language: C++
Nature of problem: Solving non-stationary problems in 1D, 2D and 3D
Solution method: Alternating direction solver with isogeometric finite element method

If you use this software in your work, please cite

Marcin Łoś, Maciej Woźniak, Maciej Paszyński, Andrew Lenharth, Keshav Pingali *IGA-ADS : Isogeometric Analysis FEM using ADS solver*, **Computer & Physics Communications** 217 (2017) 99-116 (available on researchgate.org)

Isogeometric L2 projections over 3D grids

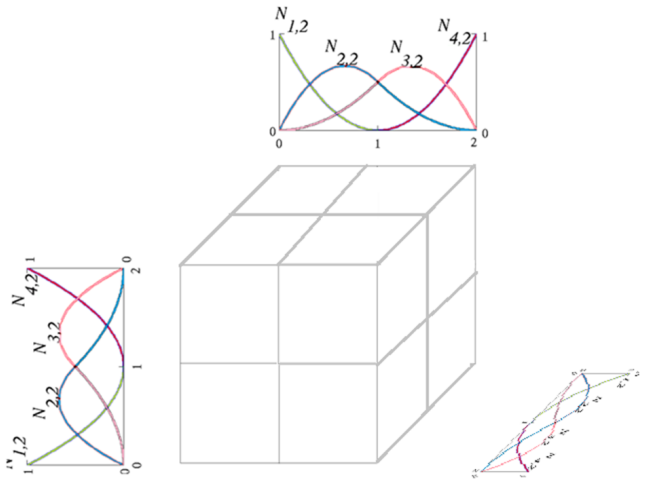


Figure: Tensor product structure of the 3D mesh

Isogeometric basis functions:

- 1D B-splines basis along x axis $B^x_{1,p}(x), \ldots, B^x_{N_x,p}(x)$
- 1D B-splines basis along y axis $B^y_{1,p}(y), \ldots, B^y_{N_y,p}(y)$
- 1D B-splines basis along z axis $B^z_{1,p}(z), \ldots, B^z_{N_z,p}(z)$
- In 3D we take tensor product basis $\{B^x_{i,p}(x)B^y_{j,p}(y)B^z_{k,p}(z)\}_{i=1,\ldots,N_x;j=1,\ldots,N_y;1,\ldots,N_z}$

Isogeometric L2 projections over 3D grids

Gram matrix of B-spline basis on 3D domain $\Omega = \Omega_x \times \Omega_y \times \Omega_z$:

$$\mathcal{M}_{ijklmn} = (B_{ijk}, B_{lmn})_{L^2} = \int_\Omega B_{ijk} B_{lmn} \, \mathrm{d}\Omega$$

$$= \int_\Omega B_i^x(x) B_j^y(y) B_k^z(z) B_l^x(x) B_m^y(y) B_n^z(z) \, \mathrm{d}\Omega$$

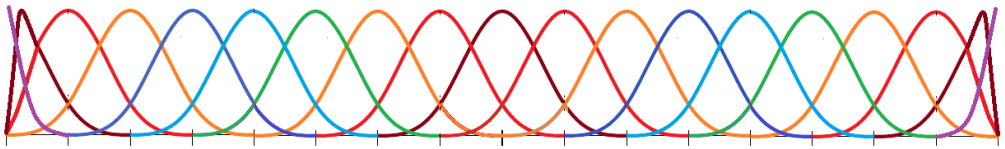
$$= \int_\Omega (B_i B_l)(x) \, (B_j B_m)(y) \, (B_k B_n)(z) \, \mathrm{d}\Omega$$

$$= \left(\int_{\Omega_x} B_i B_l \, \mathrm{d}x \right) \left(\int_{\Omega_y} B_j B_m \, \mathrm{d}y \right) \left(\int_{\Omega_z} B_k B_n \, \mathrm{d}z \right)$$

$$= \mathcal{M}_{il}^x \mathcal{M}_{jm}^y \mathcal{M}_{kn}^z$$

$$\mathcal{M} = \mathcal{M}^x \otimes \mathcal{M}^y \otimes \mathcal{M}^z \quad \text{(Kronecker product)}$$

Isogeometric L2 projections over 3D grids



B-spline basis functions have **local support** (over $p+1$ elements)

$\mathcal{M}^x$, $\mathcal{M}^y$, ... – banded structure

$\mathcal{M}^x_{ij} = 0 \iff |i-j| > 2p+1$

Exemplary basis functions and matrix for cubics

$$\begin{bmatrix}
(B_1,B_1)_{L^2} & (B_1,B_2)_{L^2} & (B_1,B_3)_{L^2} & (B_1,B_4)_{L^2} & 0 & 0 & \cdots & 0 \\
(B_2,B_1)_{L^2} & (B_2,B_2)_{L^2} & (B_2,B_3)_{L^2} & (B_2,B_4)_{L^2} & (B_2,B_5)_{L^2} & 0 & \cdots & 0 \\
(B_3,B_1)_{L^2} & (B_3,B_2)_{L^2} & (B_3,B_3)_{L^2} & (B_3,B_4)_{L^2} & (B_3,B_5)_{L^2} & (B_3,B_6)_{L^2} & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
0 & 0 & \cdots & (B_n,B_{n-3})_{L^2} & (B_n,B_{n-2})_{L^2} & (B_n,B_{n-1})_{L^2} & (B_n,B_n)_{L^2} &
\end{bmatrix}$$

Isogeometric L2 projections over 3D grids

Three steps – solving systems with **A** and **B** and **C** in different *directions*.
First, we solve along x direction, where we have $B_1^x, \cdots, B_m^x$

$$\begin{bmatrix} B_1^x B_1^x & B_1^x B_2^x & \cdots & 0 \\ B_2^x B_1^x & B_2^x B_2^x & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_m^x B_m^x \end{bmatrix} \begin{bmatrix} z_{111} & z_{211} & \cdots & z_{kl1} \\ z_{112} & z_{212} & \cdots & z_{kl2} \\ \vdots & \vdots & \ddots & \vdots \\ z_{11m} & z_{21m} & \cdots & z_{klm} \end{bmatrix} =$$

$$\begin{bmatrix} b_{111} & b_{211} & \cdots & b_{kl1} \\ b_{112} & b_{212} & \cdots & b_{kl2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{11m} & b_{21m} & \cdots & b_{klm} \end{bmatrix}$$

Isogeometric L2 projections over 3D grids

Three steps – solving systems with **A** and **B** and **C** in different *directions*.
Second, we solve along y direction, where we have $B_1^y, \cdots, B_l^y$

$$\begin{bmatrix} B_1^y B_1^y & B_1^y B_2^y & \cdots & 0 \\ B_2^y B_1^y & B_2^y B_2^y & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_l^y B_l^y \end{bmatrix} \begin{bmatrix} y_{111} & y_{211} & \cdots & y_{k1m} \\ y_{121} & y_{211} & \cdots & y_{k2m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{1l1} & y_{1l1} & \cdots & y_{klm} \end{bmatrix} =$$

$$\begin{bmatrix} z_{111} & z_{111} & \cdots & z_{k1m} \\ z_{121} & z_{211} & \cdots & z_{k2m} \\ \vdots & \vdots & \ddots & \vdots \\ z_{1l1} & z_{2l1} & \cdots & z_{klm} \end{bmatrix}$$

Isogeometric L2 projections over 3D grids

Three steps – solving systems with **A** and **B** and **C** in different *directions*.
Second, we solve along z direction, where we have $B_1^z, \cdots, B_k^z$

$$\begin{bmatrix} B_1^z B_1^z & B_1^z B_2^z & \cdots & 0 \\ B_2^z B_1^z & B_2^z B_2^z & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_k^z B_k^z \end{bmatrix} \begin{bmatrix} x_{111} & x_{121} & \cdots & x_{1lm} \\ x_{211} & x_{221} & \cdots & x_{2lm} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k11} & x_{k21} & \cdots & x_{klm} \end{bmatrix} =$$

$$\begin{bmatrix} y_{111} & y_{121} & \cdots & y_{1lm} \\ y_{211} & y_{221} & \cdots & y_{2lm} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k11} & y_{k21} & \cdots & y_{klm} \end{bmatrix}$$

Integration of the 1D mass matrix

We have $B^x_{i,p_x} i = 1, .., p_x + 1$ basis functions on 1D each element
We have $ngx = O(p_x)$ Gauss points

$(B^x_{i,p}, B^x_{j,p})_{L2} = \int_{\Omega_x} B^x_{i,p}(x) B^x_{j,p}(x) dx =$
$\sum_{E_x} \int_{E_x} B^x_{i,p}(x) B^x_{j,p}(x) dx = \sum_E \sum_{s=1,ngx} W_s B^x_{i,p}(x_s) B^x_{j,p}(x_s)$

We construct the mass matrices for each 1D element, for all the basis functions which span over the 1D element, namely $B^x_{l,p} l = 1, p_x + 1$.

Integration of the 1D mass matrix

We have $B^x_{i,p_x}\, i = 1, .., p_x + 1$ basis functions on 1D each element
We have $ngx = O(p_x)$ Gauss points

```
// Gauss integration points
```

- `for` $s = 1, ngx$

  `get Gauss point` x_s, `and weight` W_s

```
// B-spline basis functions
```

- `for` $i = 1, p_x + 1$
- `for` $j = 1, p_x + 1$
- `aggregate` $W_s * B^x_{i,p}(x_s) B^x_{j,p}(x_s)$

$p_x = p$ computational complexity $O(p^3)$, if p=9 it is 10^3

Integration of the right-hand-side

We have a mesh of $N_x \times N_y \times N_z$ elements
$nrdof = (p_x + 1)(p_y + 1)(p_z + 1)$ basis functions on each element
(p_x, p_y, p_z) denotes the B-splines order in directions x, y and z
$ngx = O(p_x)$, $ngy = O(p_y)$, $ngz = O(p_z)$ number of Gauss points

$$(F, B^x_{l,p} B^y_{m,p} B^z_{n,p})_{L2} =$$
$$\int_\Omega F(x,y,z) B^x_{l,p}(x) B^y_{m,p}(y) B^z_{n,p}(z) dxdydz =$$
$$\sum_E \int_E F(x,y,z) B^x_{l,p}(x) B^y_{m,p}(y) B^z_{n,p}(z) dxdydz =$$
$$\sum_E \sum_{s=1,ngx;t=1,ngy;w=1,ngz} W_s W_t W_z F(x_s, y_t, z_w) B^x_{l,p}(x_s) B^y_{m,p}(y_t) B^z_{n,p}(z_w)$$

We construct the right-hand-side vectors for each element, for all the basis functions which span over the element, namely
$B^x_{l,p} l = 1, p_x + 1$, $B^y_{m,p} m = 1, p_y + 1$, $B^z_{m,p} n = 1, p_z + 1$.

Integration of the right-hand-side

We have a mesh of $N_x \times N_y \times N_z$ elements
$nrdof = (p_x + 1)(p_y + 1)(p_z + 1)$ basis functions on each element
(p_x, p_y, p_z) denotes the B-splines order in directions x, y and z
$ngx = O(p_x)$, $ngy = O(p_y)$, $ngz = O(p_z)$ number of Gauss points
// Gauss integration points

- for $s = 1, ngx$
- for $t = 1, ngy$
- for $w = 1, ngz$

get Gauss point (x_s, y_t, z_w), weight $W_s W_t W_w$
// B-spline basis functions

- for $l = 1, p_x + 1$
- for $m = 1, p_y + 1$
- for $n = 1, p_z + 1$
- aggregate $W * F(x_s, y_t, z_w), B^x_{l,p}(x_s) B^y_{m,p}(x_t) B^z_{n,p}(x_w)$

$p_x = p_y = p_z = p$ computational complexity $O(p^6)$, if p=9 it is 10^6

Sequential integration of the RHS

$F = 0.d0$

```
do ex = 1,nelemx //Loop through elements
  do ey = 1,nelemy
    do ez = 1,nelemz
      J = Jx(ex)*Jy(ey)*Jz(ez) //element Jacobian
      do kx = 1,ngx //Loop through Gauss points
        do ky = 1,ngy
          do kz = 1,ngz
            W = Wx(kx)*Wy(ky)*Wz(kz) //Gauss weight
            value = fvalue(Xx(kx,ex),Xy(ky,ey),Xz(kz,ez))
            do ax = 0,px //B-splines
              do ay = 0,py
                do az = 0,pz
          call compute_index(ind,ax,ay,az,ex,ey,ez,nx,ny,nz)
                F(ind) = F(ind) +
     NNx(0,ax,kx,ex)*NNy(0,ay,ky,ey)*NNz(0,az,kz,ez)*J*W*value
```

Parallel OpenMP integration of the RHS

OpenMP = Open Multi-Processing

```
!$OMP PARALLEL DO
!$OMP& DEFAULT(SHARED)
!$OMP& FIRSTPRIVATE
    (iy,ex,ey,ez,J,kx,ky,kz,W,value,ax,ay,az,ind)
!$OMP& REDUCTION(+:nr_nonzeros)
  do iy=1,miy //Now it is 1 loop over elements
   call map_indexes(iy,ex,ey,ez)
   J = Jx(ex)*Jy(ey)*Jz(ez) //element Jacobian
   do kx = 0,ngx //loop through Gauss points
    do ky = 0,ngy
     do kz = 0,ngz
        W = Wx(kx)*Wy(ky)*Wz(kz) //Gauss weight
        value = fvalue(Xx(kx,ex),Xy(ky,ey),Xz(kz,ez))
        do ax = 0,px //B-splines along x,y,z
         do ay = 0,py
          do az = 0,pz
           call compute_index(ind,ax,ay,az,ex,ey,ez,nx,ny,nz)
           F(ind) = F(ind) +
      NNx(0,ax,kx,ex)*NNy(0,ay,ky,ey)*NNz(0,az,kz,ez)*J*W*value
!$OMP END PARALLEL DO
```

Parallel shared-memory explicit dynamics with GALOIS

Parallel version for shared-memory machines (C++ GALOIS)
Marcin Łoś, Maciej Woźniak, Maciej Paszyński, Andrew Lenharth, Keshav Pingali *IGA-ADS : Isogeometric Analysis FEM using ADS solver*, **Computer Physics Communications** 217 (2017) 99-116

Parallel shared-memory explicit dynamics with GALOIS

Explicit method with fast isogeometric L2 projections algorithm
→ Thousands of time step executed with the same matrix
→ The factorization is no longer a problem!
→ Most of the time is spent on the integration

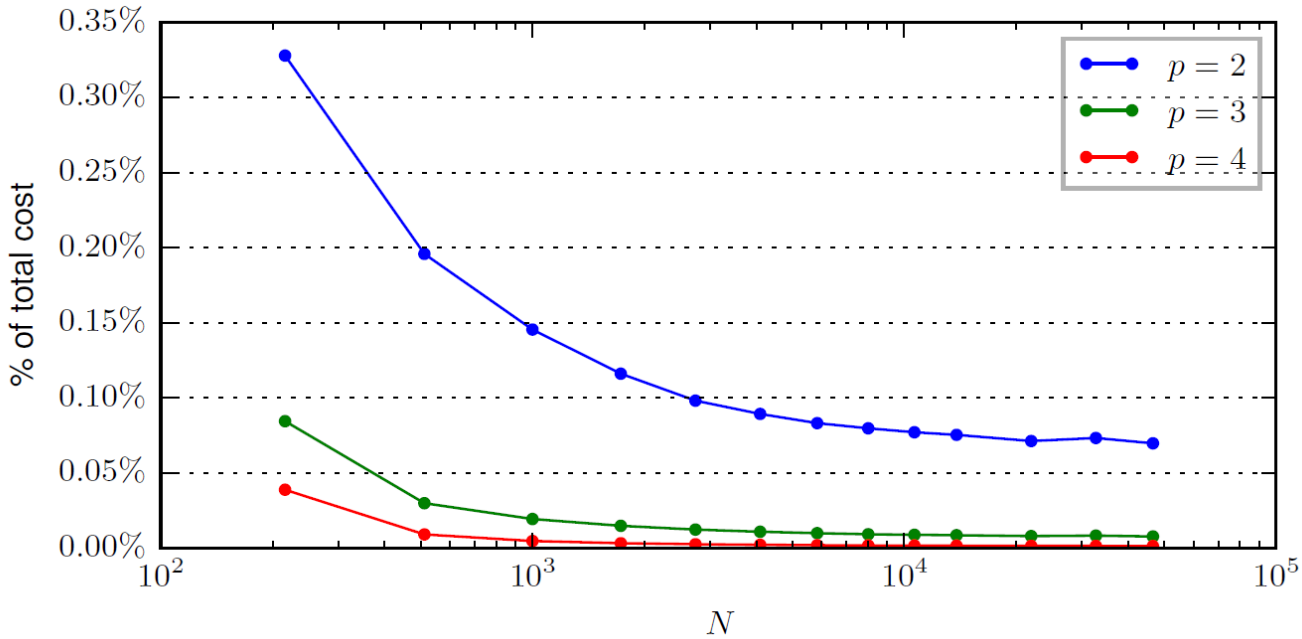


Figure: Total time = integration + factorization. Percent of the time spent on factorization is below 1 percent, for all p and N

Integration that can be speeded-up on multi-core machines

Parallel shared-memory explicit dynamics with GALOIS

for *each element* $E = [\xi_{I_x}, \xi_{I_x+1}] \times \left[\xi_{I_y}, \xi_{I_y+1}\right] \times [\xi_{I_z}, \xi_{I_z+1}]$ **do**
  for *each quadrature point* $\boldsymbol{\xi} = \left(X_{k_x}, X_{k_y}, X_{k_z}\right)$ **do**
    $\mathbf{x} \leftarrow \Psi_E(\boldsymbol{\xi})$;
    $W \leftarrow w_{k_x} w_{k_y} w_{k_z}$;
    $u, Du \leftarrow 0$;
    for $I \in \mathcal{I}(E)$ **do**
      $u \leftarrow u + U_I^{(t)} \mathcal{B}_I(\boldsymbol{\xi})$;
      $Du \leftarrow Du + U_I^{(t)} \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
    end
    for $I \in \mathcal{I}(E)$ **do**
      $v \leftarrow \mathcal{B}_I(\boldsymbol{\xi})$;
      $Dv \leftarrow \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
      $U_I^{(t+1)} \leftarrow U_I^{(t+1)} + W\,|E|\,(uv + \Delta t\, F(u, Du, v, Dv))$
    end
  end
end

Each element – independent computation
except for updating $U^{(t+1)}$ – shared state

- localize state, update once atomically
- execute element computations in parallel

Parallel shared-memory explicit dynamics with GALOIS

for *each element* $E = [\xi_{I_x}, \xi_{I_x+1}] \times \left[\xi_{I_y}, \xi_{I_y+1}\right] \times [\xi_{I_z}, \xi_{I_z+1}]$ *in parallel* **do**
  $U^{loc} \leftarrow 0$;
  for *each quadrature point* $\boldsymbol{\xi} = \left(X_{k_x}, X_{k_y}, X_{k_z}\right)$ **do**
    $\mathbf{x} \leftarrow \Psi_E(\boldsymbol{\xi})$;
    $W \leftarrow w_{k_x} w_{k_y} w_{k_z}$;
    $u, Du \leftarrow 0$;
    for $I \in \mathcal{I}(E)$ **do**
      $u \leftarrow u + U_I^{(t)} \mathcal{B}_I(\boldsymbol{\xi})$;
      $Du \leftarrow Du + U_I^{(t)} \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
    end
    for $I \in \mathcal{I}(E)$ **do**
      $v \leftarrow \mathcal{B}_I(\boldsymbol{\xi})$;
      $Dv \leftarrow \nabla \mathcal{B}_I(\boldsymbol{\xi})$;
      $U_I^{loc} \leftarrow U_I^{loc} + W\,|E|\,(uv + \Delta t\, F(u, Du, v, Dv))$;
    end
  end
  synchronized
    for $I \in \mathcal{I}(E)$ **do**
      $U_I^{(t+1)} \leftarrow U_I^{(t+1)} + U_I^{loc}$
    end
  end
end

Implementation: `Galois::for_each`, `Galois::Runtime::LL::SimpleLock`

Parallel shared-memory explicit dynamics with GALOIS

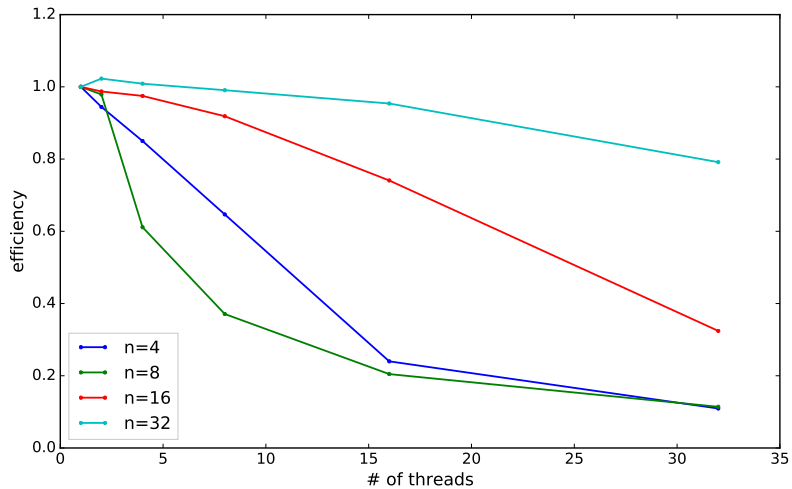


Figure: Efficiency for quadratic B-splines

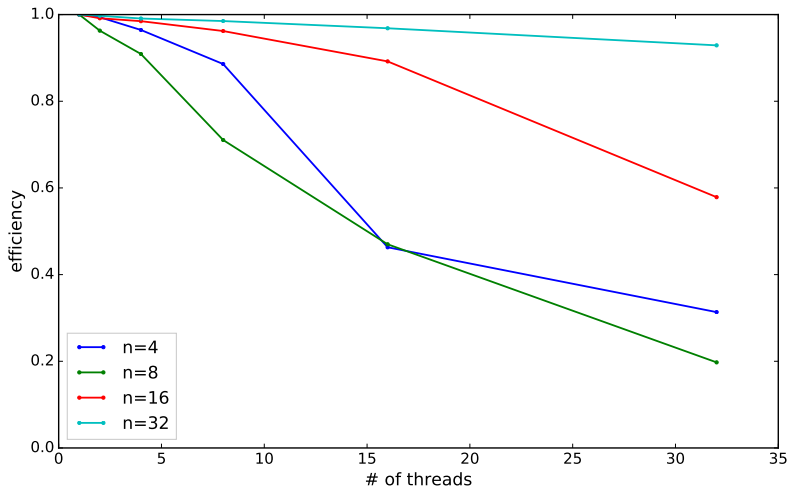


Figure: Efficiency for cubic B-splines

GALOIS framework on GILBERT shared-memory machine

Parallel shared-memory explicit dynamics with GALOIS

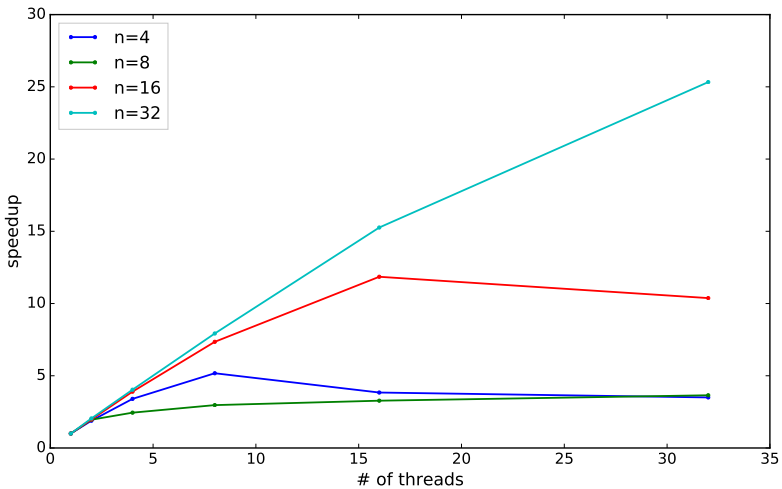


Figure: Speedup for quadratic B-splines

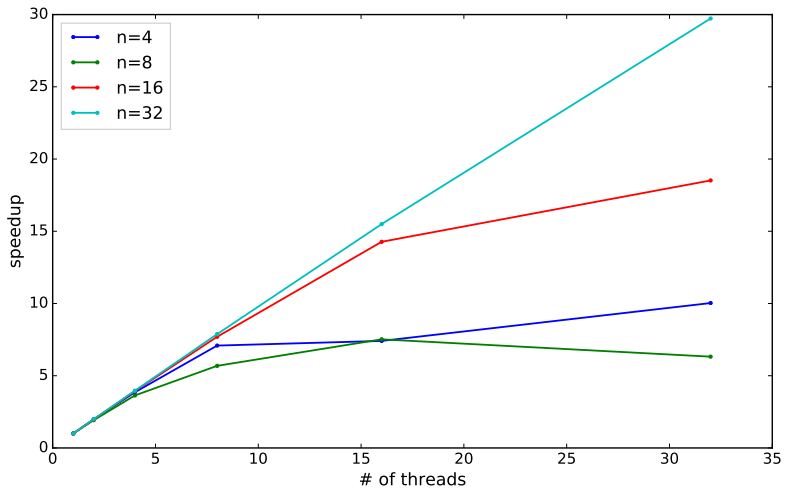


Figure: Speedup for cubic B-splines

GALOIS framework on GILBERT shared-memory machine

Parallel shared-memory explicit dynamics with GALOIS

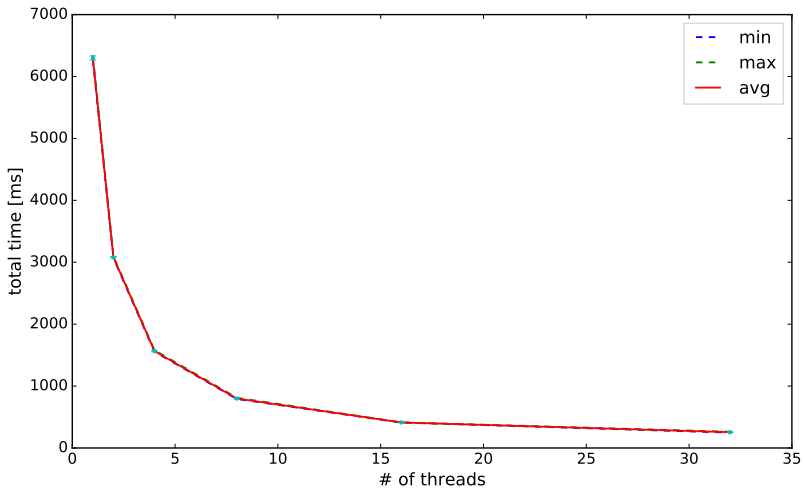


Figure: Execution time for quadratic B-splines

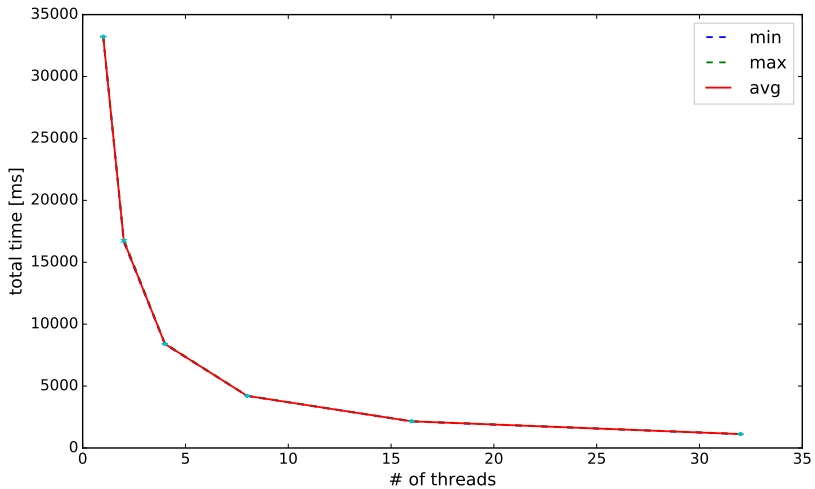
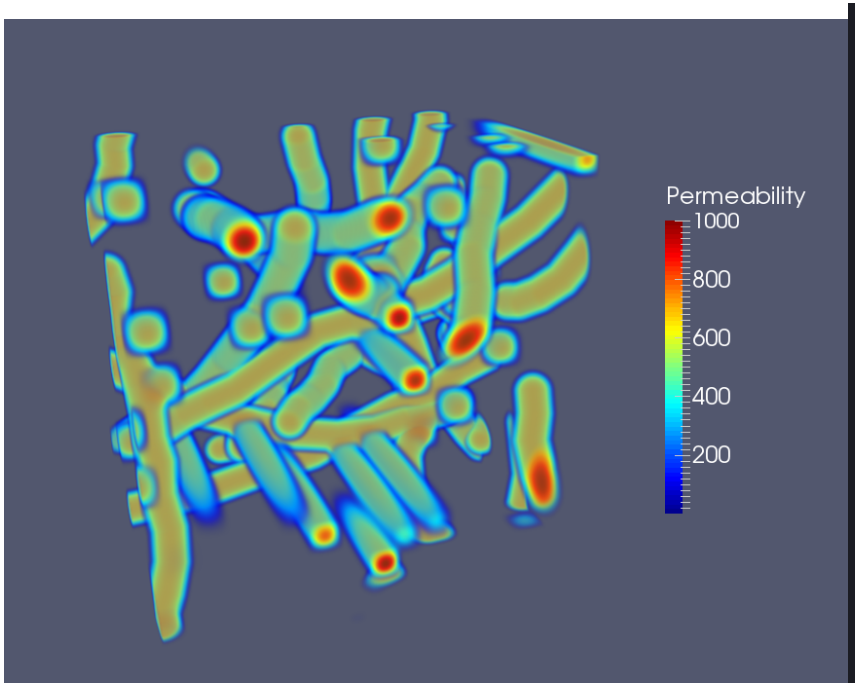


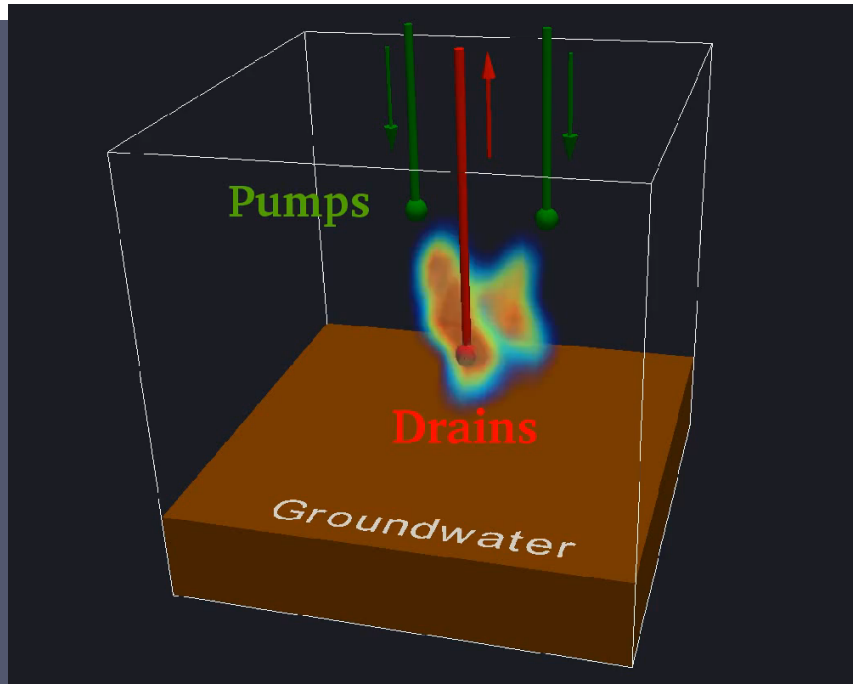
Figure: Execution time for cubic B-splines

GALOIS framework on GILBERT shared-memory machine

Example 2: Non-linear flow in heterogenous media

Hydraulic fracturing - oil/gas extraction technique consisting in high-pressure fluid injection into the deposit





Example 2: Non-linear flow in heterogenous media

Hydraulic fracturing - oil/gas extraction technique consisting in high-pressure fluid injection into the deposit
Spatial domain $= \Omega = [0,1]^3$

$$\begin{cases} \dfrac{\partial u}{\partial t} - \nabla \cdot (\kappa(\mathbf{x}, u)\, \nabla u) = h(\mathbf{x}, t) & \text{in } \Omega \times [0, T] \\ \nabla u \cdot \hat{n} = 0 & \text{on } \partial\Omega \times [0, T] \\ u(x, 0) = u_0 & \text{in } \Omega \end{cases}$$

- u – pressure
- zero Neumann boundary conditions
- initial state u_0
- κ – permeability
- h – **forcing** (induced by extraction method)

M. Alotaibi, V.M. Calo, Y. Efendiev, J. Galvis, M. Ghommem, *Global-Local Nonlinear Model Reduction for Flows in Heterogeneous Porous Media* **arXiv:1407.0782 [math.NA]**

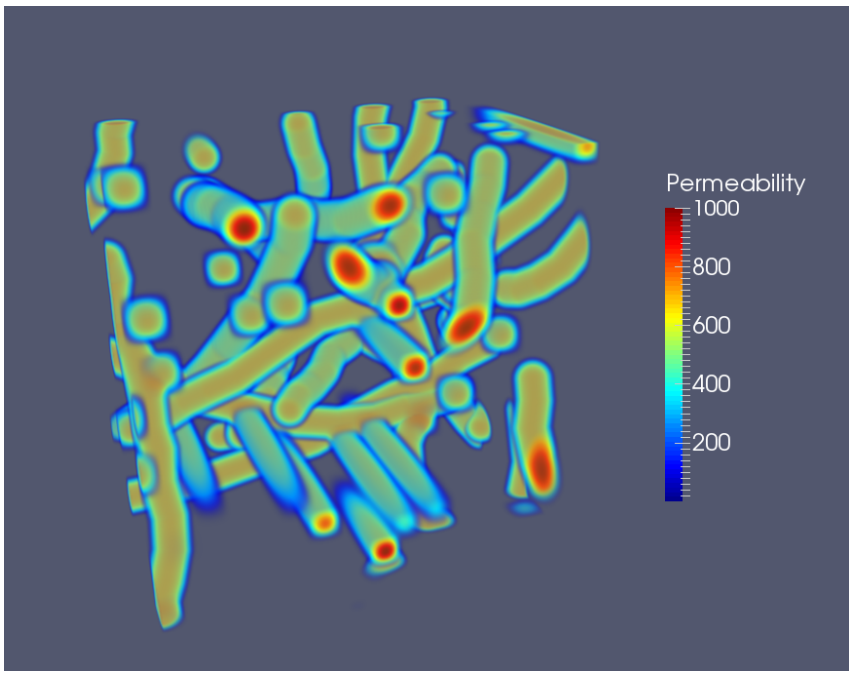
Example 2: Non-linear flow in heterogenous media

$$\kappa(\mathbf{x}, u) = K_q(x)\, b(u)$$

$$b(u) = e^{\mu u}$$

$$mu = 10$$

$K_q(\mathbf{x})$ – property of the terrain (example below)



Example 2: Non-linear flow in heterogenous media

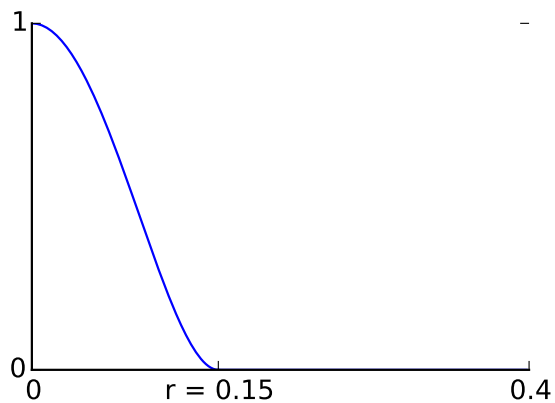
Extraction process modeled by **pumps** and **sinks**

- pump/sink has a location $\mathbf{x} \in \Omega$
- pumps locally increase the pressure u
- sinks locally decrease u (the higher, the faster)

$$h(x,t) = \sum_{p \in P} \phi\left(\|x_p - x\|\right) - \sum_{s \in S} u(x,t)\phi\left(\|x_s - x\|\right)$$

- P, S – sets of pump and sinks
- x_p, x_s – location of pump p/sink s
- ϕ – cut-off function ($r = 0.15$)

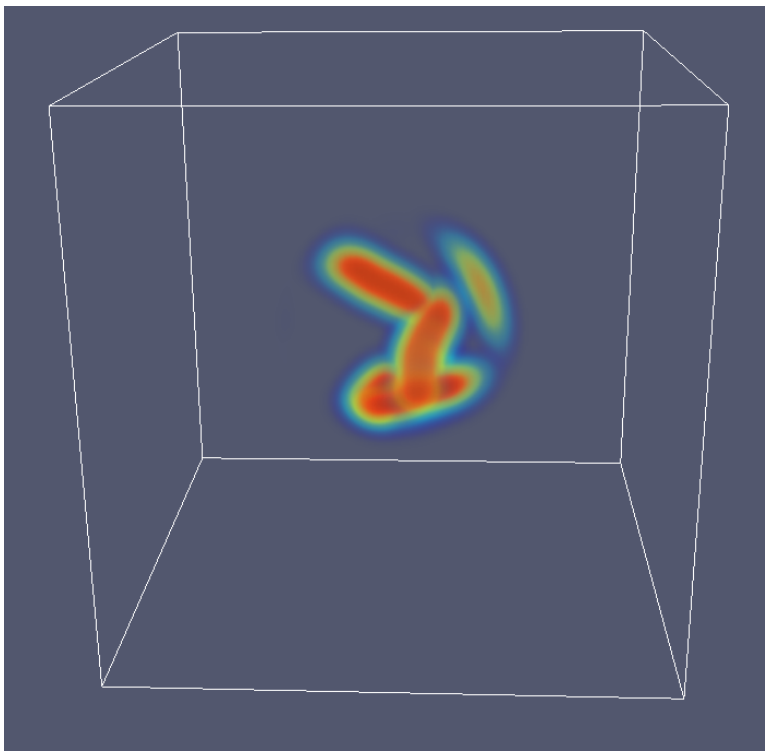
$$\phi(t) = \begin{cases} \left(\frac{t}{r} - 1\right)^2 \left(\frac{t}{r} + 1\right)^2 & \text{for } t \leq r \\ 0 & \text{for } t > r \end{cases}$$



Example 2: Non-linear flow in heterogenous media

Initial state is derived from the permeability of the material K_q

$$\tilde{K}_q(\mathbf{x}) = (K_q(\mathbf{x}) - 1)/(1000 - 1)$$
$$u_0(\mathbf{x}) = 0.1\, \tilde{K}_q(\mathbf{x})\, \theta_{0.2,0.3}(\|\mathbf{x} - \mathbf{c}\|)$$
$$\mathbf{c} = (0.5, 0.5, 0.5)$$



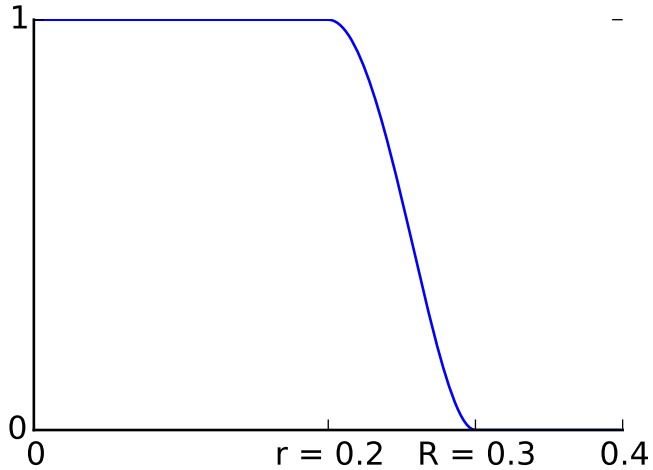


Figure: $\theta_{r,R}$

Example 2: Non-linear flow in heterogenous media

We utilize Euler time integration scheme

$$(u_{t+1}, w)_{L^2} = (u_t + h, w)_{L_2} - dt * (K_q(x)\, e^{10 * u_t} \nabla u_t, \nabla w)_{L^2} \quad (1)$$

where $K_q(x, t) = K_q(x)$ does not change with time, and it is given by the permeability map,
and $h(x, t)$ are pumps / sinks

$$h(x, t) = \sum_{p \in P} \phi\left(\|x_p - x\|\right) - \sum_{s \in S} u(x, t) \phi\left(\|x_s - x\|\right)$$

Example 2: Non-linear flow in heterogenous media

Click in the middle

Code for Example 2 (Non-linear flow)

"problems/flow/flow.cpp"

```
#include "problems/flow/flow.cpp"
using namespace ads;
using namespace ads::problems;
pilot for the simulation
int main() {
quadratic B-splines, 20 elements along axis
  dim_config dim{ 2, 20 };
5000 time steps, time step size 10^-7
  timesteps_config steps{ 10000, 1e-7 };
we will need to compute first derivatives during the computations
  int ders = 1;
some auxiliary objects for configuration and simulation
config_3d c{dim, dim, dim, steps, ders};
heat_3d sim{c};
run the simulation
sim.run();
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

```
#include "ads/simulation.hpp"
#include "ads/executor/galois.hpp"  parallel loop execution
#include "problems/flow/pumps.hpp"  pumps and sinks location
#include "problems/flow/environment.hpp" map of formation
#include "ads/output_manager.hpp"  dumpout of snapshots for graphics
class flow :  public simulation_3d {
...
implementation of the initial state
double init_state(double x, double y, double z)
executed once before the simulation starts
void before() override
executed before every simulation step
void before_step() override
implementation of the simulation step
void step() override
executed after every simulation step
void after_step() override
implementation of generation of RHS
void compute_rhs() override
executed once after the simulation ends
void after() override
```

Code for Example 2 (Non-linear flow)

"problems/flow/geometry.hpp"

this functions is called from *before* at the beginning of the simulation
the function returns the value of $u_0 = u(x, y, z)|_{t=0}$) computed at (x, y, z)

```
double init_state(double x, double y, double z) {
    double r = 0.1;
    double R = 0.5;
    return ads::bump(r, R, x, y, z);
  };

// r < R in [0, 1]
inline double bump(double r, double R, double x, double y, double
z) {     double dx = x - 0.5, dy = y - 0.5, dz = z - 0.5;
    double t = std::sqrt(dx * dx + dy * dy + dz * dz);
    return falloff(r / 2, R / 2, t);
}

inline double falloff(double r, double R, double t) {
    if (t < r) return 1.0;
    if (t > R) return 0.0;
    double h = (t - r) / (R - r);
    return std::pow((h - 1) * (h + 1), 2);
}
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

this function is called once before the simulation starts

```
void before() override {
```

preparing map of the formation

```
    fill_permeability_map();
```

performs LU factorization of three 1D systems, representing B-splines along x, y and z axes

```
    prepare_matrices();
```

pointer to *init_state* function

```
    auto init = [this](double x, double y, double z)
     { return init_state(x, y, z); };
```

preparation of the initial state

```
    projection(u, init);
```

forward and backward substitutions with multiple RHS

```
    solve(u);
```

dumpout the snapshot from this time step

```
    output.to_file(u, "out_%d.vti", 0);
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

```
void fill_permeability_map() {
    for (auto e:elements()) {  loop through elements
     for (auto q:quad_points()) {  loop through Gauss
points
      auto x = point(e, q);
      kq(e[0], e[1], e[2], q[0], q[1], q[2]) =
       env.permeability(x[0], x[1], x[2]);
     }
    }
}
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

```
void fill_permeability_map() {
for(auto e:elements()) {  loop through elements
  for(auto q:quad_points()){loop through Gauss points
   auto x = point(e, q);
   kq(e[0], e[1], e[2], q[0], q[1], q[2]) =
    env.permeability(x[0], x[1], x[2]);
  }
}
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

this function is called before every time step

```
void before_step(int /*iter*/, double /*t*/) override
{
    using std::swap;
```

swap u_t and u_{t-1}

```
    swap(u, u_prev);
}
```

this function implements every time step

```
void step(int /*iter*/, double /*t*/) override {
```

generate new RHS using u_prev

```
    compute_rhs();
```

forward and backward substitutions with multiple RHS

```
    solve(u);
}
```

Code for Example 2 (Non-linear flow)

We utilize Euler time integration scheme

$$(u_{t+1}, w)_{L^2} = (u_t + h, w)_{L_2} - dt * (K_q(x)\, e^{10 * u_t} \nabla u_t, \nabla w)_{L^2}$$

K_q is estimated from the permeability map

```
k = permeability(e, q);
```

u_t value over element e at Gauss point q

```
value_type u = eval_fun(u_prev, e, q);
```

the forcing is based on the location of pumps and sinks

```
h = forcing(x, t);
```

value of test function a over element e at Gauss point q

```
value_type v = eval_basis(e, q, a);
```

$(-K_q(x)\, e^{10 * u_t}, \nabla u_t)_{L_2} + (h, \nabla v)_{L^2}$

```
val= -k*std::exp(10*u.val)*grad_dot(u, v)+h*v.val;
```

$(u_t + h, w)_{L_2} - dt * (K_q(x)\, e^{10 * u_t} \nabla u_t, \nabla w)_{L^2}$, scaled by Jacobian*weight

```
U(aa[0],aa[1],aa[2])+=(u.val*v.val+steps.dt*val)*w*J;
```

Code for Example 2 (Non-linear flow)

```
parallel processing of loop through elements
executor.for_each(elements(), [&](index_type e) {
  double J = jacobian(e);
  for (auto q :  quad_points()) { Gauss points
weight and pointt for Gaussian quadrature
   double w = weigth(q); auto x = point(e, q);
value of permeability at Gauss point
   double k = permeability(e, q);
u_t value over element e at Gauss point q
   value_type u = eval_fun(u_prev, e, q);
the forcing is based on the location of pumps and sinks
   double h = forcing(x, t);
   for (auto a:dofs_on_element(e)) { test functions
remapping local to global index for aggregation of RHS
    auto aa = dof_global_to_local(e, a);
value of test function a over element e at Gauss point q
    value_type v = eval_basis(e, q, a);
    double val = - k*std::exp(10*u.val)*grad_dot(u, v)+h*v.val;
    U(aa[0],aa[1],aa[2])+=(u.val*v.val+steps.dt*val)*w*J;
the update of RHS must be synchronized when processed in parallel
executor.synchronized( [ & ] { update_global_rhs(rhs, U, e); });
```

Code for Example 2 (Non-linear flow)

"problems/heat/heat_3d.hpp"

this function is called once before the simulation starts

```
void before() override {
```

performs LU factorization of three 1D systems, representing B-splines along x, y and z axes

```
    prepare_matrices();
```

pointer to *init_state* function

```
    auto init = [this](double x, double y, double z)
     { return init_state(x, y, z); };
```

preparation of the initial state

```
    projection(u, init);
```

forward and backward substitutions with multiple RHS

```
    solve(u);
    }
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

```
void after_step(int iter, double /*t*/) override {
  if (iter % 10 == 0) {
```

Print out the L2 energy of the preassure field in current time step

```
    std::cout « "Step " « iter « ", energy:" «
     energy(u) « std::endl;
  }
```

Dump out data for ParaView graphics every 100 time steps

```
  if ((iter + 1) % 100 == 0) {
   output.to_file(u, "out_%d.vti", iter + 1);
  }
```

Code for Example 2 (Non-linear flow)

"problems/flow/flow.hpp"

```
double permeability(index_type e, index_type q) const
{
  return kq(e[0], e[1], e[2], q[0], q[1], q[2]);
}

double forcing(point_type x, double /*t*/) const {
  using std::sin;
  double pi2 = 2 * M_PI;
  return
   1+sin(pi2*x[0])*sin(pi2*x[1])*sin(pi2*x[2]);
}
```