`inSilico` **Tutorial**

Thomas Rüberg

November 11, 2013

Contents

Notation

Throughout the text bold-typed letters indicate a point or a vector in $\mathbb{R}^d$, e.g. $\boldsymbol{x} = (x_1, \ldots, x_d)$. Sans serif letters denote matrices (or vectors as column-matrices), e.g. $\mathsf{A} = \{A_{KL}\}$ with some index ranges for K and L. Moreover, the following abbreviations are introduced in the text and frequently used:

- FEM — Finite Element Method
- BVP — Boundary Value Problem
- IBVP — Initial Boundary Value Problem
- ODE — Ordinary Differential Equation .

1 Theoretical background

The Finite Element Method (FEM) is one of many numerical techniques for the approximate solution of boundary value problems (BVP). Such BVPs are commonly mathematical models of physical problems and a wide range of such problems has been successfully analysed by means of FEM. Even though the method's main application background is in the field of structural analysis and solid mechanics, its applicability goes far beyond these fields.

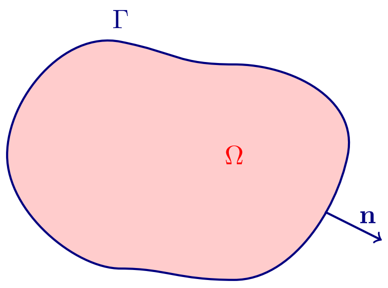

Figure 1: Model domain Ω with boundary Γ

1.1 Abstract boundary value problem and weak form

In order to go through the main characteristics of the method, some abstract notation will be used in the following. Specific examples follow in the tutorial sections 2.1 to 2.3. Let Ω denote a domain embedded in $\mathbb{R}^d$ and let $\Gamma = \partial\Omega$ be its boundary, see figure 1. Moreover, the boundary Γ is partitioned in two regions Γ^D and Γ^N such that $\Gamma = \Gamma^D \cup \Gamma^N$. We are interested in the values of a function u in Ω, where this function fulfils the following BVP

$$\begin{aligned} Lu &= f & \boldsymbol{x} &\in \Omega \\ u &= g^D & \boldsymbol{x} &\in \Gamma^D \\ T_{\boldsymbol{n}}u &= g^N & \boldsymbol{x} &\in \Gamma^N \,. \end{aligned} \tag{1}$$

Here, L and $T_{\boldsymbol{n}}$ are (linear) differential operators. f is a body force or source term function, g^D is function that prescribes u on the boundary part Γ^D (*Dirichlet* condition) and g^N is function that prescribes $T_{\boldsymbol{n}}u$ on the boundary part Γ^N (*Neumann* condition). In case of inconvenience with this notation, refer to table 1 for some common physical examples.

	u/g^D	g^N	f	Lu	$T_{\boldsymbol{n}}u$
heat	temperature θ	heat flux	heat source	$-\kappa\nabla^2\theta$	$\kappa\boldsymbol{n}\cdot\nabla\theta$
linear elasticity	displacement $\boldsymbol{u}$	traction	body force	$-\mu\nabla\cdot\nabla\boldsymbol{u} - (\lambda+\mu)\nabla\nabla\cdot\boldsymbol{u}$	$[\mu\nabla\boldsymbol{u} + (\lambda+\mu)\nabla\boldsymbol{u}^\top]\cdot\boldsymbol{n}$

Table 1: Physical examples for the notation of the abstract boundary value problem (1).

The paramount ingredient in the derivation of a finite element formulation is the *weak form* which will now be introduced. Therefore take another function v (called *test function*) which is zero on the boundary part Γ^D.[*] Now the residual of the partial differential equation in (1), i.e. $Lu - f = 0$, is multiplied (weighted) by v and the whole term is integrated over the domain Ω. Then *integration by parts*[†] gives the desired expression. This approach (*weighted residual method*) reads

$$\int\limits_\Omega (Lu - f)v \,\mathrm{d}\boldsymbol{x} = a(u,v) - \int\limits_\Omega fv \,\mathrm{d}\boldsymbol{x} - \int\limits_{\Gamma^N} (T_{\boldsymbol{n}}u)v \,\mathrm{d}s = a(u,v) - \int\limits_\Omega fv \,\mathrm{d}\boldsymbol{x} - \int\limits_{\Gamma^N} g^N v \,\mathrm{d}s = 0\,. \tag{2}$$

Note that $v = 0$ on Γ^D and thus the integral over that boundary part vanishes, and on Γ^N the value of $T_{\boldsymbol{n}}u$ is prescribed by g^N. It remains to explain the ominous object $a(u,v)$ appearing in equation (2). This is a bilinear form which means that it is linear in both arguments (e.g., $a(\alpha u_1 + \beta u_2, v) = \alpha a(u_1, v) + \beta a(u_2, v)$) and its result is a real number. Obviously these facts are not very helpful and we consider

[*]There are regularity requirements on v (and on u in the first place) which are not discussed here.
[†]Recall the divergence theorem $\int_\Omega \nabla\cdot\boldsymbol{u}\,\mathrm{d}\boldsymbol{x} = \int_\Gamma \boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}s$.

the expressions for $a(u,v)$ for the cases given in table 1

$$\text{heat:} \qquad a(\theta, v) = \int_\Omega \kappa \nabla\theta \cdot \nabla v \,\mathrm{d}\boldsymbol{x}$$

$$\text{elasticity:} \qquad a(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{u}) \,\mathrm{d}\boldsymbol{x}$$

with the stress tensor $\boldsymbol{\sigma}$ and the linear strain $\boldsymbol{\varepsilon}$. The weak form (2) is the foundation of the *variational principle*:

Find u with $u = g^D$ on Γ^D such that

$$a(u,v) = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} + \int_{\Gamma^N} g^N v \,\mathrm{d}s \qquad \text{for all } v \text{ with } v = 0 \text{ on } \Gamma^D \tag{3}$$

which is the basis of any standard finite element formulation.

1.2 Spatial discretization

The overall task is to find an approximation u_h[‡] for the solution u of the BVP (1). The most common approach in FEM is the trial

$$u_h(\boldsymbol{x}) = \sum u^K \varphi_u^K(\boldsymbol{x}), \tag{4}$$

that is a linear combination of unknown coefficients u^K and predefined functions φ_u^K. The functions φ_u^K form the essence of any FEM and will be introduced in the following.

First, the domain Ω of our mathematical model (1) is decomposed in small convex polytopes. Note that this process (commonly referred to as *triangulation*) is in general an approximation. Let τ_e denote the e-th of in total N_e polytopes. Then we have the expression

$$\Omega \approx \Omega_h = \bigcup_{e=1}^{N_e} \tau_e \tag{5}$$

which states that Ω is approximated by Ω_h which itself is the union of all polytopes τ_e. The triangulation of Ω_h is called the grid or *mesh* and the individual polytopes are referred to as *elements*. For sake of clarity, the polytopes are of the type

$d = 1$: line elements,

$d = 2$: triangle or quadrilateral elements,

$d = 3$: tetrahedron or hexahedron elements.

Recall that d is the spatial dimension of the problem.

The next step is to introduce *shape functions*. These usually serve three purposes

- representation of the geometry of Ω_h

$$\boldsymbol{x}_h = \sum \boldsymbol{x}^M \varphi_{\boldsymbol{x}}^M, \tag{6}$$

- the trial (4),

[‡]The letter h refers to a characteristic size of the elements introduced below. The notion $\lim_{h\to 0} \|u_h - u\| = 0$ is what is commonly understood as convergence of the method.

- and as test functions $v = \varphi_v^L$.

In many, but not all, circumstances it is customary to set $\varphi_u^K = \varphi_v^K$, i.e. to use the same functions for the trial (4) and as test functions. Moreover, especially in structural analysis, it is common to use for the geometry representation the same functions as for the trial, i.e. $\varphi_{\boldsymbol{x}}^K = \varphi_u^K$. This latter simplification is referred to as *isoparametric*. Unless specified otherwise, we will use the notation

$$\varphi^K := \varphi_u^K = \varphi_v^K = \varphi_{\boldsymbol{x}}^K \,. \tag{7}$$

But what are now these φ? In most cases, one introduces a so-called *reference element* $\hat{\tau}$, which can be mapped to any of the elements τ_e of the triangulation of Ω_h

$$T_e : \quad \xi \ni \hat{\tau} \to \boldsymbol{x}_h(\xi) \in \tau_e \,. \tag{8}$$

The most standard approach is to use polynomials on $\hat{\tau}$. Examples are the linear functions on line, triangle or tetrahedron elements which assume the value 1 at one vertex and 0 at the other vertices. For quadrilateral elements, the corresponding shape functions with the same feature would be bilinear. The shape function on an element τ_e result from the inverse coordinate map of shape functions $\hat{\varphi}$ defined on the reference element

$$\varphi^K(\boldsymbol{x})|_{\tau_e} = \hat{\varphi}^K(\xi(\boldsymbol{x})) = (\hat{\varphi}^K \circ T_e^{-1})(\boldsymbol{x}) \tag{9}$$

and, in general, lose their polynomial character. The precise definitions of shape functions is given in any standard text book on FEM and will not be considered here any further.

For most physical fields, it is standard to request a certain regularity of the solution u that carries over to its approximate u_h. The continuity of u_h across the boundaries of adjacent elements is a very frequently imposed condition. For the example of the linear shape functions, this condition is imposed by requiring a unique value u^K at every vertex K of the triangulation.

In the variational principle (3) it was required that $u = g^D$ and $v = 0$ on Γ^D. This way of imposing boundary conditions is called *essential*. The standard way for imposing these conditions is to set $u^K = g^D(\boldsymbol{x}^K)$ for all vertices $\boldsymbol{x}^K \in \Gamma^D$ and to discard any test function $v = \varphi^K$ for the same set of vertices. Note that this approach is very restricted since the following requirements have to be fulfilled

- $\varphi^K(\boldsymbol{x}^K) = 1$ and $\varphi^L(\boldsymbol{x}^K) = 0$ for all other $L \neq K$, and
- $\boldsymbol{x}^K \in \Gamma^D$ (note that $\boldsymbol{x}^K$ is a vertex of the approximate geometry and not necessarily lies on the true boundary Γ^D).

Whereas the latter condition is often and successfully discarded the former can be a severe restriction. We define two index sets

$$\mathbb{I} = \{K : \quad \boldsymbol{x}^K \in \Omega_h \text{ and } \boldsymbol{x}^K \notin \Gamma_h^D\} \,, \qquad \mathbb{J} = \{K : \quad \boldsymbol{x}^K \in \Gamma_h^D\} \,, \tag{10}$$

introduce the coefficients

$$A_{KL} = a(\varphi^L, \varphi^K) \,, \qquad b_K = \int\limits_{\Gamma_h^N} g^N \varphi^K \, \mathrm{d}s + \int\limits_{\Omega_h} f \varphi^K \, \mathrm{d}s \,, \tag{11}$$

and the matrices

$$\mathsf{A} = \{A_{KL}\}_{K,L\in\mathbb{I}} \,, \qquad \mathsf{x} = \{u^K\}_{K\in\mathbb{I}} \,, \qquad \mathsf{b} = \{b_K - \sum_{L\in\mathbb{J}} A_{KL} u^L\}_{K\in\mathbb{I}} \,. \tag{12}$$

Note the appearance of the prescribed coefficients u^L, $L \in \mathbb{J}$, in the definition of b. Finally, we have a system

$$\mathsf{A}\mathsf{x} = \mathsf{b} \tag{13}$$

whose solution yields the unknown coefficients u^K, $K \in \mathbb{I}$, and the trial (4) gives the approximate solution u_h at any point $\boldsymbol{x} \in \Omega_h$.

1.3 Temporal discretization

The BVP (1) does not depend on time and therefore the solution of system (13) gives the time-constant state of the physical problem to analyse. Now we consider temporal variations for the time interval $[0,T]$ in the following form[§]

$$\begin{aligned} \rho\partial_t u + Lu &= f & \boldsymbol{x} &\in \Omega\,, & t &\in [0,T] \\ u &= g_D & \boldsymbol{x} &\in \Gamma^D\,, & t &\in [0,T] \\ T_{\boldsymbol{n}} u &= g_N & \boldsymbol{x} &\in \Gamma^N\,, & t &\in [0,T] \\ u &= u_0 & \boldsymbol{x} &\in \Omega\,, & t &= 0\,. \end{aligned} \tag{14}$$

With respect to the BVP (1), the differences are: appearance of the first time derivative $\partial_t u$ multiplied by the material density ρ, the initial condition (last equation above) and the time dependency of all involved quantities. Problem (14) is usually called an *initial* boundary value problem (IBVP). Since its solution $u(\boldsymbol{x},t)$ is now a function of space and time, so is its approximate $u_h(\boldsymbol{x},t)$. There are various approaches to tackle the time variation. Here we follow the approach of first discretizing in space and then applying a finite difference method to the resulting system of ordinary differential equations (ODE). The starting point of this approach is rewrite the trial (4)

$$u_h(\boldsymbol{x},t) = \sum u^K(t)\varphi_u^K(\boldsymbol{x})\,, \tag{15}$$

with time-dependent coefficients. Following now the steps of section 1.2 leads to the system

$$\mathsf{M}\dot{\mathsf{x}} + \mathsf{A}\mathsf{x} = \mathsf{b}\,, \tag{16}$$

where the *mass* matrix has been introduced which is defined as

$$\mathsf{M} = \{M_{KL}\}_{K,L\in\mathbb{I}}\,, \quad M_{KL} = \int\limits_{\Omega_h} \rho\varphi^K\varphi^L\,\mathrm{d}\boldsymbol{x}\,. \tag{17}$$

System (16) is now represented by the simple (ODE)

$$\dot{y} = f(y,t)\,. \tag{18}$$

A huge variety of methods exists for the numerical solution of this ODE and we will restrict ourselves to the family of linear multi-step methods. Let the considered time interval $[0,T]$ be subdivided into N pieces of size Δt.[¶] The n-th multiple of Δt is the time point $t^n = n\Delta t$ and the evaluation of the (approximate) quantities at this point is denoted by using n as a superscript, $f^n = f(y^n,t^n)$ and $y^n \approx y(t^n)$. Assume that all quantities at t^m, $m \leq n$ are given and the values at t^{n+1} are now of interest. A linear multi-step method gives the equation

$$\sum_{s=0}^{A} a_s y^{n+1-s} = \Delta t \sum_{s=0}^{B} b_s f^{n+1-s}\,. \tag{19}$$

The number of terms A and B for the left and right side, respectively, and the coefficients a_s and b_s are determined by the choice of method. Expression (19) can be carried over to system (16) and gives a sequence of linear systems of equations to solve, one for every time point t^n.

1.4 `inSilico` — A generic Finite Element library

`inSilico` is a software *library*, that means it is designed to serve as a code development platform. It is *not* a FEM solver package, like `ABAQUS`. If you want to use `inSilico`, you need to program an application code yourself!

[§]Other temporal variations are possible, e.g. higher-order derivatives, varying domain $\Omega(t)$ or viscous material laws.
[¶]The size of the time steps Δt does not need to be constant but we assume this simpler form here.

`inSilico` is programmed in `C++` and makes heavily use of *templates*. In order to get an idea of templates, consider the example given in figure 2. It shows two possible template-based ways to implement the computation of the maximum of two numbers. The important feature is that neither the class nor the function knows of which specific type `T` is. Therefore, they can be employed in the software for any data type that allows the *less than* comparison based on the `operator<`.

```
template<typename T>
struct Max {
  static T apply(const T& a, const T& b)
  {
    return (a > b? a : b );
  }
};
//usage (Note: explicit instantiation)
const double maximum = Max<double>::apply( 3.1, -1.2 );
```

```
template<typename T>
T max(const T& a, const T& b)
{
  return (a > b? a : b);
}
//usage (Note: argument dependent look-up)
const int    maxInt = max( 3,   -1   );
const double maxDob = max( 3.1, -1.2 );
```

Figure 2: A template class (top) and a template function (bottom) for the computation of the maximum of two numbers; usage examples are given.

This seemingly simple example hints at the power hidden behind the concept of template-based programming: properly designed code units have to be defined only once for a certain task. In the example in the bottom of figure 2 the function is used for two different number types without the need of writing a max-function for `int` and `double` separately. Two important remarks have to be made at this point:

- The second version looks neater, since the user only has to call that function and the type of `T` is implicitly determined by the argument types of function. This concept is called *argument dependent look-up*. Note that it fails if the types cannot be uniquely determined from the argument types.
- Compilers are usually not very good in reporting a compilation error in a template-based code. Try compiling the example code for a complex number type (no <-comparison available) and enjoy the lengthy output.

Another potential of templates is the use of parameters instead of types, which, for instance, is commonly used for fixed-size arrays. An illustrative example is the template-based meta-programming as shown in figure 3. Here, the factorial of a number n is computed. Note that the factorial has two possible definitions which obviously give the same result,

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n \qquad \text{or} \qquad n! = n(n-1)!\,, \quad 0! = 1\,.$$

Note that the standard way (top of figure 3) exactly reflects the first definition of the factorial. The second, recursive definition is represented by the meta-program (bottom of the figure). The advantage of this way is that the compiler will compute the factorial and effectively write its result into the compiled program. Thereby no run-time is consumed for this computation. It has to be emphasised that

- The value of the parameter of a template has to be known at compile time. In the example, it is not possible to compute $n!$ if n is a value that is asked from the user of the program.

```
unsigned factorial( const unsigned n )
{
  unsigned result = 1;
  for ( unsigned i = 2; i <= n; i++) result *= i;
  return result;
}

const unsigned facN = factorial( N ); //usage
```

```
template<unsigned N> struct Factorial // recursive template
{
  static const unsigned value = N * Factorial<N-1>::value;
};

template<> struct Factorial<0> // recursion stop
{
  static const unsigned value = 1;
};

//usage:
const unsigned facN = Factorial<N>::value; //N is known at compile-time
```

Figure 3: Traditional way (top) and meta-program (bottom) to compute the factorial.

The library `inSilico` uses these concepts whenever possible. Consider for example the node class as displayed in figure 4. The class itself depends on the spatial dimension `DIM` which determines the size of the coordinate the node holds. The exact type used for that coordinate, defined elsewhere in the library, shall not matter. Since the spatial dimension of the physical problem is something that is known at the beginning and does not change in the computation, it is an ideal candidate for a template parameter. Moreover, the class holds two template functions for the change and the access of the coordinate. Basically, the components of the coordinate are copied from or to an iterator which goes over some storage specified by the caller. This could refer to an array or something more complicated, but does not affect the design of the node class due to the template functions.

The design rationale of `inSilico` is to separate data storage from algorithms. In line with this idea is the concept to abstract finite elements from the physical specifications. The main module of `inSilico` is the `base` module which contains all the data structures and algorithms necessary for a finite element analysis without specification of any physical model. On the other hand, the modules of `heat`, `solid`, and `fluid` contain the integral kernels as the they appear in the corresponding physical models of heat conduction (or diffusion), elasticity and viscous fluid flow. Figure 5 shows the top level structure of `inSilico` and the sub-modules given in `base`.

Next to the discussed modules, there are

`config`: Configuration of the software with machine dependent setups

`tools`: A variety of tools used in the process of the finite element analysis: mesh-file conversion, post-processing tools, etc.

```
template<unsigned DIM>
class Node
{
public:
    //! Template parameter: spatial dimension
    static const unsigned dim = DIM;

    //! Coordinate type definition
    typedef typename base::Vector<dim>::Type VecDim;

    //! Constructor
    Node()
        : id_( base::invalidInt ),
          x_(  base::invalidVector<dim>() )
    { }

    //------------------------------------------------------------------------
    //! Set the global ID
    void setID( const std::size_t id ) { id_ = id; }
    //! Set coordinates from iterator
    template<typename INPITER>
    void setX( INPITER iter )
    {
        for ( unsigned d = 0; d < dim; d ++ ) x_[d] = *iter++;
    }

    //------------------------------------------------------------------------
    //! Return global ID of node
    std::size_t getID() const { return id_; }
    //! Pass coordinates to an iterator
    template<typename OUTITER>
    void getX( OUTITER iter ) const
    {
        for ( unsigned d = 0; d < dim; d ++ ) *iter++ = x_[d];
    }

private:
    std::size_t id_; //!< Global Node ID
    VecDim      x_;  //!< Coordinate of this node
};
```

Figure 4: Reduced version of the node class in `inSilico`.

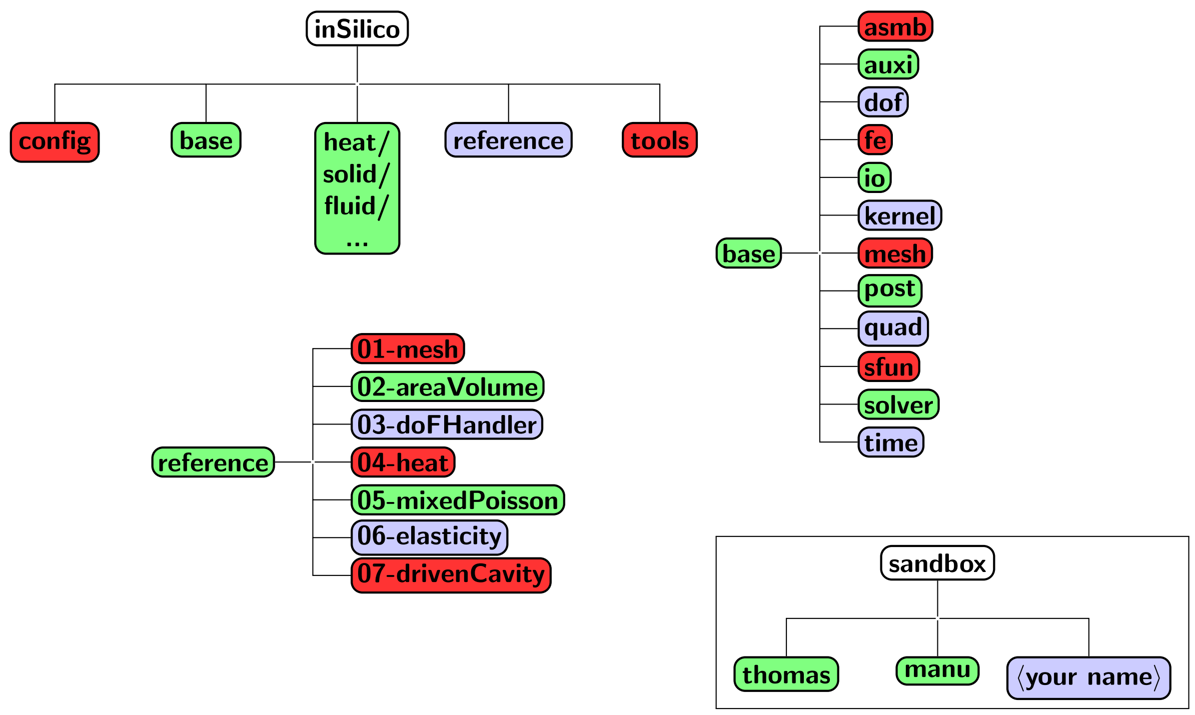

Figure 5: Directory structure of `inSilico`: top level (top left), `base` module (top right), reference applications (bottom left) and the `sandbox`.

`reference`: Reference applications which address specific parts of the library and are (partially) tested.

In addition, there is the `sandbox` in which random applications can be placed and maintained. Note that the entire library and its applications are stored within the version management system `Subversion` on a file server. A few of the implemented finite element features of `inSilico` are given in table 2.

element shapes	line (LINE), triangle (TRI), quadrilateral (QUAD), tetrahedron (TET), hexahedron (HEX)
shape functions	Lagrangian (linear and quadratic), B-Splines (any order)
quadrature	Gauß-Legendre (1- to 10-point rules), tensor-products for QUAD and HEX, Gauß-type quadrature for TRI and TET
time integration	BDF and Adamas-Moulton up to fourth order
solver	Cholesky, LU, Conjugate gradient (all via `Eigen3`)

Table 2: Some FEM features of `inSilico`.

At last, an abstract worflow of a finite element analysis with `inSilico` shall be outlined. A preliminary is that the steps in figure 6 have already been successfully carried out.

- Get a mesh file with a triangulation of the domain. The native format of `inSilico` is `smf`, but there are two alternative ways:
  - convert any other format to `smf` (a converter for `Gmsh` mesh files is provided in the `tools` module)
  - write your own input method which fills the data of the mesh class (see `base/io/smf` for a guideline)
- Run an application with that mesh file. If there is no adequate application, do the following

  - copy an existing application that you find in the `reference` module or somewhere in the `sandbox` that comes closest to your wishes
  - modify and compile

- Analyse the data. The standard way is to write a `VTK` file which can be visualised with, e.g., `ParaView`. But there are many other application-specific ways to analyse the data a program generates.

---

1. Get an account on `hermes`, read-only access to `inSilico` and read-and-write access to the `sandbox`
2. Create a local system folder where you want to place the library (e.g. `makedir ∼/inSilico`)
3. Enter that folder (`cd ∼/inSilico`) and get the `inSilico` library (`svn checkout 'https://web.hermes.cps.unizar.es/repos/m2be/InSilico'`)
4. Do the same with the sandbox (`https://web.hermes.cps.unizar.es/repos/m2be/Sandbox`)
5. Install the external libraries
   - `Eigen3` (`http://bitbucket.org/eigen/eigen/get/3.1.3.tar.gz`)
   - `boost` (`https://sourceforge.net/projects/boost/files/boost/1.53.0/boost_1_53_0.tar.gz`)
6. Adapt the settings in `config.default` file (in ∼/inSilico/config).
7. Try to compile the reference applications and conversion tools

---

Figure 6: How to get and install `inSilico`.

2 Tutorial problems

2.1 Diffusion

Consider the diffusion IBVP

$$
\begin{aligned}
\rho \partial_t c - \kappa \nabla^2 c &= f & \boldsymbol{x} \in \Omega\,, \quad & t \in [0,T] \\
c &= \bar{c} & \boldsymbol{x} \in \Gamma^D\,, \quad & t \in [0,T] \\
q := -\kappa \nabla c \cdot \boldsymbol{n} &= \bar{q} & \boldsymbol{x} \in \Gamma^N\,, \quad & t \in [0,T] \\
c &= c_0 & \boldsymbol{x} \in \Omega\,, \quad & t = 0\,.
\end{aligned}
\tag{20}
$$

This problem describes the concentration c of some substance at point $\boldsymbol{x}$ and time t due to given initial and boundary conditions. The material parameters are the density ρ and the diffusion coefficient κ. On the boundary part Γ^D, the concentration is set to $\bar{c}$ and on the remaining boundary part Γ^N the boundary flux $q = -\kappa \nabla c \cdot \boldsymbol{n}$ is prescribed as $\bar{q}$. Initially, a concentration distribution c_0 is assumed.

The domain Ω is the microfluidic device depicted in figure 7 and the boundary parts Γ^D and Γ^N are described by via coordinate functions as specified in the program. Figure 8 shows the input parameters read by the program.

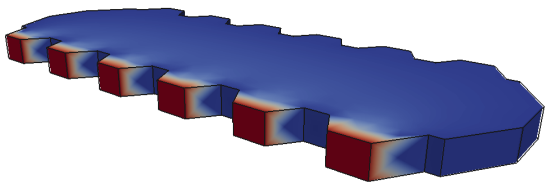

Figure 7: Microfluidic device

```
# mesh file
meshFile   microfluidic.smf

# material conductivity
kappa      1.0

# material density
rho        1.0

# time integration stuff
numSteps   50
stepSize   0.01
```

Figure 8: Input data of the program.

In this tutorial, two types of analyses are considered: steady and unsteady. In the first case, the time derivative in (20) is discarded and the parameters corresponding to a time-dependent analysis from the input file in figure 8 are ignored. In either case the class `BoundarValueProblem` as given in figure 9 is used by the code. This class defines which boundary regions are Γ^D and which are Γ^N in function of the provided coordinate. Moreover, the boundary data $\bar{c}$ and $\bar{q}$, the force function f and the initial state c_0 are provided by this class. If changes are made to `BoundarValueProblem`, the program has to be recompiled before they apply.

```
template<unsigned DIM>
struct BoundaryValueProblem
{
    static const unsigned dim = DIM;    // spatial dimension
    STATIC_ASSERT_MSG( dim == 3, "Implementation requires dim=3" );
    // convenience typedefs
    typedef typename base::Vector<DIM>::Type VecDim;
    typedef typename base::Vector<1>::Type   VecDoF;

    // boundary points with x_1 > 0.7 are on the Neumann boundary
    // or top and bottom points;
    static bool isNeumann( const VecDim & x )
    {
        const bool minX = ( std::abs( x[0] + 0.5 ) < coordTol );
        const bool maxX =  x[0] > 0.7;
        const bool maxZ = ( std::abs( x[2] - 0.    ) < coordTol ) or
                          ( std::abs( x[2] + 0.145 ) < coordTol );
        return maxX or ( maxZ and not minX );
    }
    static bool isDirichlet( const VecDim& x ) { return not(isNeumann( x ));}

    // boundary points with x_1 = -0.5 have a non-zero value
    static bool hasAppliedValue( const VecDim& x )
    {
        return std::abs( x[0] + 0.5 ) < coordTol;
    }

    // Dirichlet boundary condition
    template<typename DOF>
    static void dirichleBC( const VecDim& x, DOF* doFPtr )
    {
        const double appliedValue = 1.0;
        if ( isDirichlet( x ) ) {
            const double value = (hasAppliedValue(x) ? appliedValue : 0. );
            if ( doFPtr -> isActive(0) ) doFPtr -> constrainValue( 0, value );
        }
    }

    // Body force function
    static VecDoF forceFun( const VecDim& x )
    {
        return base::constantVector<1>( 0. );
    }

    // Flux boundary condition
    static VecDoF neumannBC( const VecDim& x, const VecDim& normal )
    {
        return base::constantVector<1>( 0. );
    }

    // Initial condition
    template<typename DOF>
    static void initialState( const VecDim& x, DOF* doFPtr )
    {
        doFPtr -> setValue( 0, 0.0 );
    }
};
```

Figure 9: Class for description of the boundary value problem's given data.

2.2 Elasticity

The equation of motion of elasticity reads in the reference configuration

$$\rho \ddot{\boldsymbol{x}} - \nabla \cdot \boldsymbol{P} = \boldsymbol{f} \,, \tag{21}$$

where $\boldsymbol{x}(\boldsymbol{X}, t)$ is the location of the material point $\boldsymbol{X}$ at time t and $\boldsymbol{P}$ the first Piola-Kirchhoff stress tensor which, in case of a hyperelastic material behaviour is given by via the strain energy W

$$\boldsymbol{P} = \frac{\partial W}{\partial \boldsymbol{F}} \,, \qquad \boldsymbol{F} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} \,. \tag{22}$$

The second quantity in (22) is the *deformation gradient* $\boldsymbol{F}$. Equation (21) has a counterpart in the spatial configuration which is not considered here. Complementing this equation with boundary and initial conditions yields the IBVP of elasticity. In the following, two special cases of this IBVP are considered: nonlinear static elasticity, linear dynamic elasticity.

2.2.1 Nonlinear static elasticity

First, we consider the time-independent version of (21) embedded in a static (nonlinear) BVP

$$\begin{aligned} -\nabla \cdot \boldsymbol{P} &= \boldsymbol{0} & \quad & \boldsymbol{X} \in \Omega_0 \\ \boldsymbol{x}(\boldsymbol{X}) - \boldsymbol{X} &= \bar{\boldsymbol{u}} & & \boldsymbol{X} \in \Omega_0^D \\ \boldsymbol{P} \cdot \boldsymbol{n} &= \boldsymbol{0} & & \boldsymbol{X} \in \Omega_0^N \,, \end{aligned} \tag{23}$$

formulated completely in the reference domain Ω_0 with outward normal vector $\boldsymbol{n}$. More specifically, a compressible Neohookean material law of the type

$$W(\boldsymbol{F}) = \frac{\lambda}{2} (\log J)^2 - \mu \log J + \frac{\mu}{2} [\mathrm{tr}(\boldsymbol{F}^\top \boldsymbol{F}) - 3] \,, \qquad J = \det(\boldsymbol{F}) \tag{24}$$

is assumed with material parameters λ and μ. The nonlinear variational principle corresponding to (23) states

$$\begin{aligned} &\text{Find } \boldsymbol{x} \text{ with } \boldsymbol{x} = \boldsymbol{X} + \bar{\boldsymbol{u}} \text{ on } \Gamma_0^D \text{ such that} \\ &\qquad \int\limits_{\Omega_0} \boldsymbol{P}(\boldsymbol{x}, \boldsymbol{X}) : \nabla \boldsymbol{v} \, \mathrm{d}\boldsymbol{X} = 0 \\ &\qquad\qquad\qquad \text{for all } \boldsymbol{v} \text{ with } \boldsymbol{v} = \boldsymbol{0} \text{ on } \Gamma_0^D . \end{aligned} \tag{25}$$

Equation (25) is often referred to as the *principle of virtual displacements* and the test function $\boldsymbol{v}$ will thus be called a virtual displacement. A Newton iteration of equation (25) has the update $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{u}$ and the increment $\boldsymbol{u}$ is the solution of the equation

$$\int\limits_{\Omega_0} (\boldsymbol{F}_k^\top \nabla \boldsymbol{v}) : \mathbb{C}_k : (\boldsymbol{F}_k^\top \nabla \boldsymbol{u}) + (\boldsymbol{F}_k \boldsymbol{P}_k) : [(\nabla \boldsymbol{v})^\top \nabla \boldsymbol{u}] \, \mathrm{d}\boldsymbol{X} = - \int\limits_{\Omega_0} \boldsymbol{P}_k : \nabla \boldsymbol{v} \, \mathrm{d}\boldsymbol{X} \,, \tag{26}$$

subject to the boundary condition on Γ_0^D. The subscript k means that the quantity is evaluated based on the current solution $\boldsymbol{x}^k$. $\mathbb{C}$ results from the linearization process and has the expression

$$\mathbb{C} = \frac{\partial^2 W}{\partial \boldsymbol{F} \partial \boldsymbol{F}} \tag{27}$$

and is usually called *elasticity tensor*. Spatial discretization of equation (26) again leads to a system of equations of the form $\mathsf{A}_k \mathsf{u} = \mathsf{b}_k$ which gives the increment over every Newton iteration. Finally, note that it is customary to apply the load (or the displacement boundary condition) incrementally (not to be confused with the above Newton increment). In the current case, we only have $\boldsymbol{x} - \boldsymbol{X} = \bar{\boldsymbol{u}}$ and thus one would work with a load increment factor $0 < \alpha \leq 1$ and apply a successively increasing amount $\alpha\bar{\boldsymbol{u}}$ until the goal of $\alpha = 1$ is reached. For every fixed value of α the above Newton method is carried out until convergence and then α is augmented. Here, simply a number N of *load steps* is prescribed and the boundary condition is increased by $\bar{\boldsymbol{u}}/N$ in every load step.

Figure 10 displays a possible combination of input parameters for the program (using the more customary material parameters E and ν instead of the above λ and μ). The outcome for these parameters is given in figure 11.

```
meshFile        quad.020.smf

# material parameter
E               100000.
nu              0.3

# distance to pull right boundary
pull            1.5

# for non-linear elasticity
maxIter         30
loadSteps       10
tolerance       1.e-8
```

Figure 10: Input data of the program.

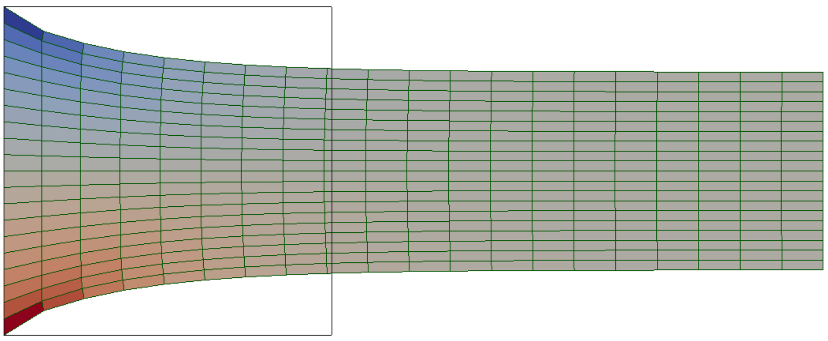

Figure 11: Initial geometry and deformed mesh, coloured by the shear stress component σ_{12}.

2.2.2 Dynamic linear elasticity

Another special case of the IBVP of elasticity is linear elasticity which has the assumptions of infinitesimal displacements and a linear material behaviour. In the dynamic case, this gives the IBVP

$$\begin{aligned} \rho\partial_t^2\boldsymbol{u} - \nabla\cdot\boldsymbol{\sigma} &= \boldsymbol{0} && \boldsymbol{x}\in\Omega\,, && t\in[0,T] \\ \boldsymbol{u} &= \boldsymbol{0} && \boldsymbol{x}\in\Gamma^D\,, && t\in[0,T] \\ \boldsymbol{\sigma}\cdot\boldsymbol{n} &= \bar{\boldsymbol{g}} && \boldsymbol{x}\in\Gamma^N\,, && t\in[0,T] \\ \boldsymbol{u} &= \boldsymbol{0} && \boldsymbol{x}\in\Omega && t<0 \end{aligned} \tag{28}$$

with the material density ρ, the prescribed (time-dependent!) traction and the Cauchy stress tensor $\boldsymbol{\sigma}$. Note that under the simplifications of linear elasticity, there is no need to distinguish between reference and current configuration nor between the various stress tensors. The used material simply follows Hooke's law

$$\boldsymbol{\sigma} = \lambda(\nabla \cdot \boldsymbol{u})\boldsymbol{I} + \mu(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^\top) \tag{29}$$

and is linear in the gradient of the displacement function $\boldsymbol{u}$. Note that in problem (28) the second time derivative occurs. Therefore, the quiescent past of the displacement function for $t < 0$ is used which implies an initial condition of zero displacement and velocity. Moreover, this second derivative does not allow a straightforward application of the linear multi-step methods as given in section 1.3. Spatial discretization of the IBVP (28) gives the system of second-order ODEs of the form

$$\mathsf{M}\ddot{\mathsf{x}} + \mathsf{A}\boldsymbol{x} = \mathsf{f}\,, \tag{30}$$

where the coefficients of the unknown x are still time-dependent. Now, the velocity $\boldsymbol{v} = \dot{\boldsymbol{u}}$ is explicitly introduced and we obtain the block system

$$\begin{pmatrix} 0 & \mathsf{M} \\ \mathsf{M} & 0 \end{pmatrix} \begin{pmatrix} \dot{\mathsf{x}} \\ \dot{\mathsf{v}} \end{pmatrix} + \begin{pmatrix} \mathsf{A} & 0 \\ 0 & -\mathsf{M} \end{pmatrix} \begin{pmatrix} \mathsf{x} \\ \mathsf{v} \end{pmatrix} = \begin{pmatrix} \mathsf{f} \\ 0 \end{pmatrix} \tag{31}$$

which is a system of first-order ODEs. Now the mechanisms of section 1.3 can be applied again.

In this problem, the only non-zero impact on the solid is a horizontal compressive load on the right side boundary whose non-zero component $\bar{g}_1$ has a temporal variation of the type

$$\bar{g}_1(0) = \begin{cases} 0 & t < t^* \\ -1 \cdot 10^4 & t \geq t^*\,, \end{cases} \tag{32}$$

that is a step function with jump at some time $t = t^*$. Figure 12 shows the input parameters of the program. The horizontal displacement u_1 and horizontal velocity v_1 components at the centre point $\boldsymbol{x} = (0.5, 0.5)$ are plotted over time in figure 13. Note that the velocity is scaled in order to have comparable amplitudes of the curves.

```
meshFile        quad.020.smf

# material parameter
E               100000.
nu              0.3

# for dynamic runs
density         1000.
numSteps        500
stepSize        0.002
```

Figure 12: Input data of the program.

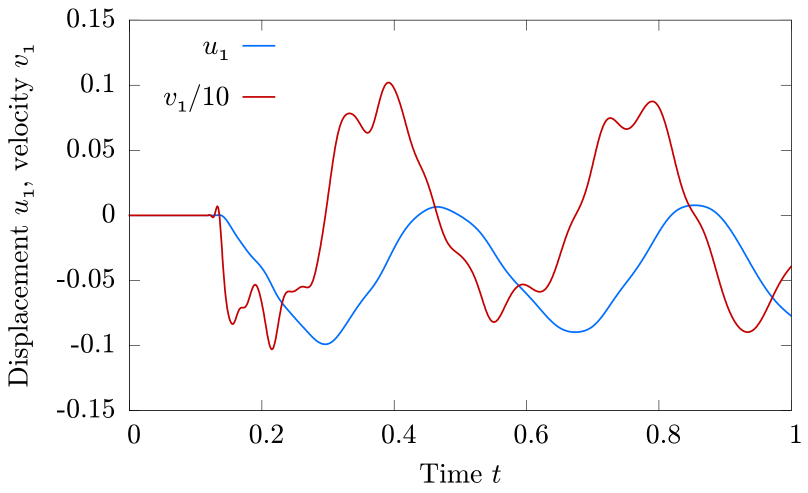

Figure 13: Plot of horizontal displacement and velocity at the centre of the domain.

2.3 Coupled fluid-diffusion

One of the main application goals of `inSilico` is multiphysics in which multiple physical problems interact. As an example, consider a tank filled with some fluid and within that fluid a certain substance is transported. We assume that the fluid mechanics of the problem are not affected by that substance and formulate the fluid IBVP via the Navier-Stokes equations

$$\begin{aligned} \nabla \cdot \boldsymbol{v} = 0\,, \quad \rho(\partial_t \boldsymbol{v} + \nabla \boldsymbol{v} \cdot \boldsymbol{v}) - \mu \nabla^2 \boldsymbol{v} + \nabla p &= \mathbf{0} & \boldsymbol{x} &\in \Omega\,, & t &\in [0,T] \\ \boldsymbol{v} &= \bar{\boldsymbol{v}} & \boldsymbol{x} &\in \Gamma\,, & t &\in [0,T] \\ \boldsymbol{v} &= \bar{0} & \boldsymbol{x} &\in \Omega\,, & t &= 0\,. \end{aligned} \tag{33}$$

The fluid dynamics are described by the velocity field $\boldsymbol{v}$ and pressure p and, in this case, only velocity boundary conditions are given on the entire boundary of the domain. Note that the momentum balance of the Navier-Stokes equations is nonlinear due to the advection term $\nabla \boldsymbol{v} \cdot \boldsymbol{v}$ which will be tackled by a simple fix point iteration and is not discussed any further. The material parameters are the mass density ρ and the fluid viscosity μ (assumption of a Newtonian fluid). With the same fluid velocity $\boldsymbol{v}$ the transport of the substance's concentration c is governed by the IBVP

$$\begin{aligned} \rho_s \partial_t c + \boldsymbol{v} \cdot \nabla c - \kappa \nabla^2 c &= \mathbf{0} & \boldsymbol{x} &\in \Omega\,, & t &\in [0,T] \\ c &= \bar{c} & \boldsymbol{x} &\in \Gamma\,, & t &\in [0,T] \\ c &= 0 & \boldsymbol{x} &\in \Omega\,, & t &= 0 \end{aligned} \tag{34}$$

with the substance's apparent density ρ_s and the diffusion constant κ. Obviously, problem (33) does not depend on c and therefore it can be solved independently. Once the velocity field $\boldsymbol{v}$ is determined, the linear problem (34) is solved for the latest values of c.

Figure 14 shows the parameter input to the program and figure 15 displays the fluid and the diffusion solutions. The chosen fluid problem is resembles the classic *driven cavity* numerical benchmark example and one can clearly see the circulation pattern in the solution. The Reynolds number is 500 for the chosen parameters. On the other hand, the concentration of the substance is set to zero at all but the top boundaries where a value of $\bar{c} > 0$ is prescribed. The contour lines in figure 15 clearly display how the advection with the fluid velocity influences the concentration field.

```
meshFile   cavity.smf
viscosity  0.002
density    1.00

# non-linear fluid iterations
maxIter    100
tolerance  1.e-5

# time stepping
numSteps   500
stepSize   0.1

# food
kappa        0.002
foodDensity  0.3
foodBoundary 0.5
```

Figure 14: Input data of the program.

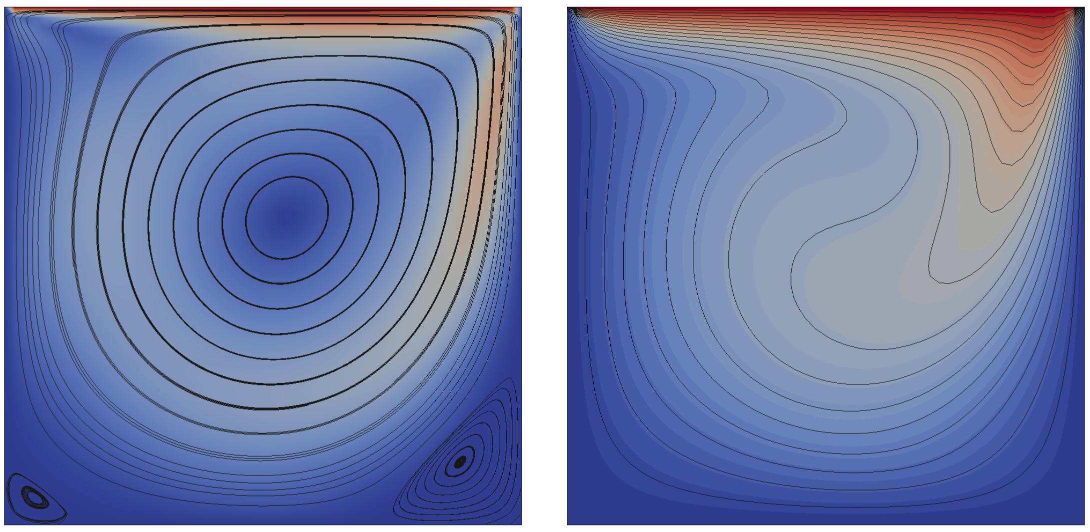

Figure 15: Fluid solution with some streamlines (left) and diffusion solution with concentration contour lines (right).