Tutorial

C. Straub, S. Gawlok

Time-Discretization Methods Based on Convection-Diffusion Equation

modified on February 3, 2016

Version 1.3

http://www.hiflow3.org/

Contents

Time-Discretization Methods Based on Convection-Diffusion Equation

1 Introduction

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to Use the Tutorial?

You find the example code (instationary_convdiff_tutorial.cc, instationary_convdiff_tutorial.h), a parameter file for the first numerical example (instationary_convdiff_tutorial.xml) and a Makefile, which you only need when using HiFlow^3 as a library (see 1.1.1), in the folder `/hiflow/examples/convection_diffusion`. The geometry data (*.inp, *.vtu) is stored in the folder `/hiflow/examples/data`.

1.1.1 Using HiFlow^3 as a Library

First install HiFlow^3. Therefore, follow the instructions listed on `http://www.hiflow3.org/`, see "Documentation"-"Installation". To compile and link the tutorial correctly, you may have to adapt the Makefile depending on the options you chose in the cmake set up. Make sure that the variable `HIFLOW_DIR` is set to the path, where HiFlow^3 was installed. The default value ist `/usr/local`. When you set the option `WITH_METIS` to `ON` in the cmake set up, you have to make sure to link to the metis library in the Makefile (`www.cs.umn.edu/~metis`). `-lmetis` must be added to the end of the line in the Makefile, where the target `instationary_convdiff_tutorial` is build (see second option in the Makefile, which is marked as a comment). By typing `make` in the console, in the same folder where the source-code and the Makefile is stored, you compile and link the tutorial. To execute the instationary convection-diffusion tutorial sequentially, type **./instationary_convdiff_tutorial** `/"path_to"/instationary_convdiff_tutorial.xml /"path_to_mesh_data"/` . To execute it in parallel mode with four processes type
mpirun -np 4 ./instationary_convdiff_tutorial `/"path_to"/instationary_convdiff_tutorial.xml /"path_to_mesh_data"/` .

1.1.2 Using HiFlow^3 as a Developer

First build and compile HiFlow^3. Go to the directory `/build/example/convection_diffusion`, where the binary **instationary_convdiff_tutorial** is stored. Type **./instationary_convdiff_tutorial**, to execute the program in sequential mode. To execute in parallel mode with four processes, type

mpirun -np 4 ./instationary_convdiff_tutorial. In both cases, you need to make sure that the default parameter file instationary_convdiff_tutorial.xml is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file with the name of your xml-file (first) and the path of your geometry data (second) in the comment line, i.e.
./instationary_convdiff_tutorial `/"path_to_parameterfile"/"name_of_parameterfile".xml /"path_to_geometry_data"/`.

2 Mathematical Setup

2.1 The Instationary Convection-Diffusion Equation

We now investigate the numerical solution of the instationary convection-diffusion equation

$$\begin{aligned} u_t + a \cdot \nabla u - \nabla \cdot (\nu \nabla u) &= s && \text{in } \Omega \times (0,T], \qquad (1)\\ u(\cdot,0) &= 0 && \text{in } \Omega,\\ u &= 0 && \text{on } \partial\Omega \times (0,T]. \end{aligned}$$

See tutorial "Stabilization Schemes for Advection Dominated Stationary Convection-Diffusion Equation" for more details how to solve the stationary convection-diffusion equation using stabilization schemes.
Our focus lies on the time-discretization of (1). For that purpose, we want to solve this equation with the method of lines, which means that first the spatial derivatives are discretized while the time derivative is left continuous. This leads to a system of ordinary differential equations (ODEs) in time that can be solved using time discretization algorithms.

2.2 Semidiscrete Formulation

First, we derive a weak formulation of (1). Therefore, we multiply (1) by a test function $v \in C_0^\infty(\Omega)$, integrate by parts and apply an approximation argument. Since we only have to deal with homogeneous Dirichlet boundary conditions, we look for a solution $u \in H_0^1(\Omega)$. So our weak formulation reads: Find $u \in H_0^1(\Omega)$ with

$$(u_t, v) + (a \cdot \nabla u, v) + \nu(\nabla u, \nabla v) = (s, v) \qquad \forall v \in H_0^1(\Omega). \qquad (2)$$

For the semidiscrete formulation, we make the ansatz

$$u^h(x,t) = \sum_{i=1}^{N} u_i(t)\phi_i(x),$$

where $\phi_i(x)$, $i = 1, \dots, N$, denote the basis functions to the given triangulation of Ω and the corresponding given elements and $u_i(t)$, $i = 1, \dots, N$, the time dependent coefficient to the corresponding basis function $\phi_i(x)$. Plugging in this ansatz into (2) yields

$$
\begin{aligned}
&\sum_{i=1}^{N} \partial_t u_i(t)(\phi_i(x), \phi_j(x)) \\
&+ \sum_{i=1}^{N} u_i(t)(a \cdot \nabla\phi_i(x), \phi_j(x)) \\
&+ \nu \sum_{i=1}^{N} u_i(t)(\nabla\phi_i(x), \nabla\phi_j(x)) = \sum_{i=1}^{N} s_i(t)(\phi_i(x), \phi_j(x)).
\end{aligned} \tag{3}
$$

If we introduce the matrices

$$M_{j,i} := (\phi_i(x), \phi_j(x)), \quad C_{j,i} := (a\cdot\nabla\phi_i(x), \phi_j(x)), \quad A_{j,i} := (\nabla\phi_i(x), \nabla\phi_j(x)) \quad i,j = 1,\dots,N,$$

the above equation can be written as vector valued equation

$$\frac{\partial U(t)}{\partial t} = S(t) - M^{-1}CU(t) - \nu M^{-1}AU(t), \tag{4}$$

where

$$U(t) := (u_i(t))_{i=1}^N \in \mathbb{R}^N, \quad S(t) := (s_i(t))_{i=1}^N \in \mathbb{R}^N.$$

This is our semidiscrete formulation and we have a system of N ODEs in time.

Remark 2.1. *For the implementation of the term $S(t)$, it is not absolutely necessary to compute the local mass matrix as it is written in the theoretical part of this tutorial. You can also simply write and implement*

$$S(t) = (s(x,t), \phi_j(x))_{j=1}^N.$$

We chose this formulation, because it is straight forward to implement it in the assembling routine, see section 3.4.5.

2.3 Time Discretization

For the time discretization, we divide the time interval into an equidistant grid of points. Let $T > 0$ and $\Delta t = \frac{T}{M}$, $M \in \mathbb{N}$. Then the grid is given by

$$t^n := n \cdot \Delta t, \quad n = 0, \dots, M.$$

For the numerical tests of the following time stepping schemes all given results are fixed to a maximal time $T = 100$, i.e. the number of time steps M multiplied by Δt results is $T = 100$.

2.4 θ-Family of Time Discretization Methods

If we discretize the time derivative by

$$\frac{u(t^{n+1}) - u(t^n)}{\Delta t} = \theta u_t(t^{n+1}) + (1-\theta)u_t(t^n) + O\left(\left(\frac{1}{2} - \theta\right)\Delta t, \Delta t^2\right),$$

and apply this scheme to (4), then we obtain

$$
\begin{aligned}
U(t^{n+1}) = U(t^n) + \Delta t\Big(&\theta[S(t^{n+1}) - M^{-1}CU(t^{n+1}) - \nu M^{-1}AU(t^{n+1})] \\
&+ (1-\theta)[S(t^n) - M^{-1}CU(t^n) - \nu M^{-1}AU(t^n)]\Big)
\end{aligned}
$$

$$\Longleftrightarrow$$

$$\begin{aligned}[M + \theta\Delta t\,(C + \nu A)]\,U(t^{n+1}) = MU(t^n) + \theta\Delta t MS(t^{n+1})\\ + (1-\theta)\Delta t\,[MS(t^n) - CU(t^n) - \nu AU(t^n)]\,. \end{aligned} \tag{5}$$

The last equation (5) is implemented in our code, see section 3.4.5.
For $\theta = 0$ this scheme results in the *explicit Euler method*, for $\theta = \frac{1}{2}$ *Crank-Nicolson method* and for $\theta = 1$ *implicit Euler method*. You can specify the method you want to use in the parameter file, see section 3.2.

2.4.1 Theoretical Results for Stability

For the θ-family of methods, the following stability result holds:

Theorem 2.1 (Stability of the θ-family of methods)**.** *Assume that there exist positive constants μ_0 and μ_1 such that*

$$0 < \mu_0 \le \mu(x) := \frac{1}{2}\operatorname{div}[a(x)] \le \mu_1$$

holds for almost every $x \in \Omega$ and that the map $t \to \|s(t)\|_0$ is bounded in $[0,T]$. When $0 \le \theta < \frac{1}{2}$, assume, moreover, that $\mathcal{T}_h$ is a uniform familiy of triangulations and that the time-step restriction

$$\Delta t \le \frac{h^2}{(1-2\theta)K_1}\min\left\{\frac{1}{\nu}, \frac{2\mu_0}{1+h^2\mu_1^2}\right\} \tag{6}$$

holds with a constant K_1. Then the solution $U(t^{n+1})$ of (5) *satisfies*

$$\|U(t^{n+1})\|_0 \le \|u_{0,h}\|_0 + K_2\sqrt{\frac{T}{\nu}}\max_{t\in[0,T]}\|s(t)\|_0,$$

where the positive constant K_2 depends only on Ω.

For a proof see [2], Proposition 12.2.1.
This Theorem states that the Crank-Nicolson and implicit Euler methods are *unconditionally stable* whereas for the explicit Euler method the restriction in (6) on the time-step has to be fulfilled.

2.5 The classical Runge-Kutta method

As a second time-stepping scheme we implemented the classical Runge-Kutta method. For an ODE of the form

$$u_t(t) = f(t, u(t)), \quad u(0) = u_0 \tag{7}$$

this method reads:

$$u(t^{n+1}) = u(t^n) + \Delta t\Phi(t^n, u(t^n); \Delta t),$$

where

$$\Phi(t^n, u(t^n); \Delta t) := \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

with

$$
\begin{aligned}
k_1 &:= f(t^n, u(t^n)),\\
k_2 &:= f\left(t^n + \frac{1}{2}\Delta t, u(t^n) + \frac{1}{2}\Delta t k_1\right),\\
k_3 &:= f\left(t^n + \frac{1}{2}\Delta t, u(t^n) + \frac{1}{2}\Delta t k_2\right),\\
k_4 &:= f\left(t^n + \Delta t, u(t^n) + \Delta t k_3\right).
\end{aligned}
$$

Equation (4) is of the form (7) and so the classical Runge-Kutta method can be applied.

3 The Commented Program

3.1 Preliminaries

HiFlow^3 is designed for high performance computing on massively parallel machines. So it applies the Message Passing Interface (MPI) library specification for message-passing, see sections 3.3, 3.4.3, [3], [1] .
The instationary convection-diffusion tutorial needs following two input files:

- A parameter file: The parameter file is an xml-file, which contains all parameters needed to execute the program. It is read in by the program. It is not necessary to recompile the program, when parameters in the xml-file are changed. By default the instationary convection-diffusion tutorial reads in the parameter file instationary_convdiff_tutorial.xml, see section 3.2, which contains the parameters of the numerical example, see section 5. This file is stored in `/hiflow/examples/convection_diffusion/`.
- Geometry data: The file containing the geometry is specified in the parameter file (instationary_convdiff_tutorial.xml). In the numerical example in section 5 we used **unit_square.inp**. You can find different meshes in the folder `/hiflow/examples/data` .

HiFlow^3 does not generate meshes for the domain Ω. Meshes in *.inp and *.vtu format can be read in. It is possible to extend the reader for other formats. Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand.

3.2 Parameter File

The needed parameters are set in the paramter file instationary_convdiff_tutorial.xml.

```
,
<Param>
  <Application>
    <Diffusion>1.e-5</Diffusion>
  </Application>
  <Mesh>
    <Filename>unit_square.inp</Filename>
    <RefinementLevel>5</RefinementLevel>
  </Mesh>
  <LinearAlgebra>
    <NameMatrix>CoupledMatrix</NameMatrix>
    <NameVector>CoupledVector</NameVector>
    <Platform>CPU</Platform>
    <Implementation>Naive</Implementation>
    <MatrixFormat>CSR</MatrixFormat>
  </LinearAlgebra>
  <FiniteElements>
    <Degree>1</Degree>
  </FiniteElements>
  <Instationary>
    <Method>CrankNicolson</Method>
    <MaxIts>100</MaxIts>
    <DeltaT>1</DeltaT>
  </Instationary>
  <LinearSolver>
    <Name>GMRES</Name>
    <Method>NoPreconditioning</Method>
    <MaxIterations>5000</MaxIterations>
    <AbsTolerance>1.e-14</AbsTolerance>
    <RelTolerance>1.e-8</RelTolerance>
    <DivTolerance>1.e6</DivTolerance>
    <SizeBasis>50</SizeBasis>
  </LinearSolver>
  <Output>
    <LogFilename>instationary_convdiff_tutorial_info_log</LogFilename>
    <DebugFilename>instationary_convdiff_tutorial_debug_log</DebugFilename>
  </Output>
</Param>
```

If you choose the time-discretization with Explicit/Implicit Euler, you have to modify the parameter "Method" in "Instationary" to "ExplicitEuler" respectively "ImplicitEuler". If you choose the time-discretization with the classical Runge-Kutta method, you have to uncomment the `#define RUNGE_KUTTA` pre-processor macro.

3.3 Main Function

The main function starts the simulation of the instationary convection-diffusion problem (instationary_convdiff_tutorial.cc).

,

```
int main(int argc, char** argv) {
    // Initialize MPI.
    MPI_Init(&argc, &argv);

    // Set default parameter file
    std::string param_filename(PARAM_FILENAME);
    std::string path_mesh;
    // Specify parameter file on console
    if (argc > 1) {
      param_filename = std::string(argv[1]);
    }
    // Specify path to geometry mesh on console
    if (argc > 2) {
      path_mesh = std::string(argv[2]);
    }
    try {
      // Run ConvDiff application.
      int rank = -1;
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == MASTER_RANK) {
          std::cout << "=============================================="
                 << "================================\n"
                 << "Running␣Convection-Diffusion␣tutorial...\n";
      }
      ConvectionDiffusionApp app(param_filename, path_mesh);
      app.run();
    } catch(std::exception& e) {
      std::cerr << "\nProgram␣ended␣with␣uncaught␣exception.\n";
      std::cerr << e.what() << "\n";
      return -1;
    }

    // Finalize MPI.
    MPI_Finalize();
    return 0;
}
```

3.4 Member Functions

Following member functions are defined in the class ConvectionDiffusionApp.

- run()
- read_parameters()
- read_and_distribute_mesh()
- prepare_system()
  - DirichletBC-struct
- assemble_system()
  - ConvectionDiffusionAssembler-class
    * void operator()(const Element& element, const Quadrature<double>& quadrature, LocalMatrix& lm);
    * void operator()(const Element& element, const Quadrature<double>& quadrature, LocalVector& lv);
    * double get_source_term(int q, double t);
    * std::vector<double> get_convective_velocity(int q);
- solve_system()
- visualize(int time)

3.4.1 run()

The member function run() is defined in the class ConvectionDiffusionApp (instationary_convdiff_tutorial.cc).

,

```
void ConvectionDiffusionApp::run() {
#ifndef RUNGE_KUTTA
    // Create log files for INFO and DEBUG output
    std::ofstream info_log(params_["Output"]["LogFilename"].
                              get<std::string>().c_str());
    LogKeeper::get_log("info").set_target(&info_log);
    std::ofstream debug_log(params_["Output"]["DebugFilename"].
                               get<std::string>().c_str());
    LogKeeper::get_log("debug").set_target(&debug_log);
    LOG_INFO("MPI␣Processes", num_partitions_);

    // Setup mesh and distribute it if run in parallel
    read_and_distribute_mesh();

    // Initialize space and linear algebra
    prepare_system();

    // Time discretization
    int i = 0;
    time_ = 0.0;
    method = params_["Instationary"]["Method"].get<std::string>();
    if (method == "CrankNicolson") {
       theta_ = 0.5;
    } else if (method == "ImplicitEuler") {
           theta_ = 1.;
           } else if (method == "ExplicitEuler") {
                  theta_ = 0.;
                  } else {
                      theta_ = 0.5;
                      std::cout  << "================================\n"
                      << "Unknown␣method␣for␣solving␣instationary␣problem\n"
                      << "================================\n"
                      << "Default␣method:␣Crank␣Nicolson␣method\n";
                    }

    // Number of time steps to be calculated
    Tmax_ = params_["Instationary"]["MaxIts"].get<int>();
    // Time stepp size
    delta_t_ = params_["Instationary"]["DeltaT"].get<double>();
    // Coefficient of diffusion
    nu_ = params_["Application"]["Diffusion"].get<double>();
      for (i = 0; i < Tmax_; ++i) {
        if (i > 0) {
            // Compute the stiffness matrix and right-hand side
            assemble_system();

            // Solve the linear system
            solve_system();
        }
      // Visualize the solution and the errors.
        visualize(i);
        time_ += delta_t_;
      }
#else
    // Read parameters.
    read_parameters();

    // Setup mesh and distribute it if run in parallel.
    read_and_distribute_mesh();

    // Initialize space and linear algebra.
    prepare_system();

    // Compute the stiffness matrix and right-hand side.
    assemble_system();

    // Implement time discretization.
    int i = 0;
    time_ = 0.0;
    for(i = 0; i < Tmax_; ++i)
    {
        if(i > 0)
        {
            // Solve the linear system.
            solve_system();
        }

        // Visualize the solution and the errors.
        visualize(i);

        time_ += delta_t_;
    }
#endif
}
```

3.4.2 read_parameters()

The member function read_parameters() is defined in the class ConvectionDiffusionApp (instationary_convdiff_tutorial.cc).

'

```
void ConvectionDiffusionApp::read_parameters() {
    // Parameters for application.
    refinement_level_ = params_["Mesh"]["RefinementLevel"].get<int>();
    fe_degree_ = params_["FiniteElements"]["Degree"].get<int>();
    nu_ = params_["Application"]["Diffusion"].get<double>();

    // Parameters for linear solver.
    basis_size_ = params_["LinearSolver"]["SizeBasis"].get<int>();
    maxits_ = params_["LinearSolver"]["MaxIterations"].get<int>();
    abstol_ = params_["LinearSolver"]["AbsTolerance"].get<double>();
    reltol_ = params_["LinearSolver"]["RelTolerance"].get<double>();
    divtol_ = params_["LinearSolver"]["DivTolerance"].get<double>();

    // Parameters for time-stepping
    // Number of time steps to be calculated
    Tmax_ = params_["Instationary"]["MaxIts"].get<int>();
    // Time stepping size
    delta_t_ = params_["Instationary"]["DeltaT"].get<double>();
}
```

3.4.3 read_and_distribute_mesh()

This member function reads the initial mesh, partitions and distributes mesh if indicated, and writes out the refined mesh of initial refniement level. Depending if it was compiled with or without metis (`www.cs.umn.edu/~metis`), metis or the naive implementation is used for the partitioning of the mesh when executed in parallel (instationary_convdiff_tutorial.cc).

'

```
void ConvectionDiffusionApp::read_and_distribute_mesh() {
    MeshPtr master_mesh(0);

    refinement_level_ = params_["Mesh"]["RefinementLevel"].get<int>();
    // Only master rank reads in mesh
    if (rank_ == master_rank_) {
        const std::string mesh_name =
                            params_["Mesh"]["Filename"].get<std::string>();
        std::string mesh_filename;
        if (path_mesh.empty()) {
        mesh_filename = std::string(DATADIR) + mesh_name;
        } else {
        mesh_filename = path_mesh + mesh_name;
        }
        master_mesh = read_mesh_from_file(mesh_filename, DIM, DIM, 0);
        assert(master_mesh != 0);

        // Global refinement of mesh
        for (int r = 0; r < refinement_level_; ++r) {
            master_mesh = master_mesh->refine();
        }
          std::cout << "=======================================\n"
               << "Mesh␣refinement␣level:␣"
               << refinement_level_ << "\n";
    }

    // Distribute mesh
    MeshPtr local_mesh = partition_and_distribute(master_mesh, master_rank_, comm_);
    assert(local_mesh != 0);

    // Compute ghost cells
    SharedVertexTable shared_verts;
    mesh_ = compute_ghost_cells(*local_mesh, comm_, shared_verts);
}
```

3.4.4 prepare_system()

The member function prepare_system() initializes the space and the linear algebra (instationary_convdiff_tutorial.cc). The polynomial degree of the finite element functions is set to the parameter "FeDegree" defined in the parameter file. The size and the non-zero pattern of the stiffness matrix matrix_ is initialized. The vectors for the right-hand side rhs_, and the solution sol_ are initialized and set to a 0-vector of the correct size.

'

```
void ConvectionDiffusionApp::prepare_system() {
#ifndef RUNGE_KUTTA
    // Assign degrees to each element
    fe_degree_ = params_["FiniteElements"]["Degree"].get<int>();
    std::vector<int> degrees(1, fe_degree_);

    // Initialize the VectorSpace object
    space_.Init(degrees, *mesh_);

    // Setup couplings object
    couplings_.Clear();
    couplings_.Init(comm_, space_.dof());

    // Compute the matrix graph
    SparsityStructure sparsity;
    global_asm_.compute_sparsity_structure(space_, sparsity);

    couplings_.InitializeCouplings(sparsity.off_diagonal_rows,
                                   sparsity.off_diagonal_cols);

    // Initialize system matrix, solution vector and vector of right-hand side
    CoupledMatrixFactory<Scalar> CoupMaFact;
    matrix_ = CoupMaFact.Get(
            params_["LinearAlgebra"]["NameMatrix"].get<std::string>() )->
            params(params_["LinearAlgebra"]);
    matrix_->Init(comm_, couplings_);
    CoupledVectorFactory<Scalar> CoupVecFact;
    sol_ = CoupVecFact.Get(
          params_["LinearAlgebra"]["NameVector"].get<std::string>() )->
          params(params_["LinearAlgebra"]);
    sol_->Init(comm_, couplings_);
    sol_prev_ = CoupVecFact.Get(
              params_["LinearAlgebra"]["NameVector"].get<std::string>() )->
              params(params_["LinearAlgebra"]);
    sol_prev_->Init(comm_, couplings_);

    matrix_->InitStructure(vec2ptr(sparsity.diagonal_rows),
                           vec2ptr(sparsity.diagonal_cols),
                           sparsity.diagonal_rows.size(),
                           vec2ptr(sparsity.off_diagonal_rows),
                           vec2ptr(sparsity.off_diagonal_cols),
                           sparsity.off_diagonal_rows.size());
    matrix_->Zeros();

    sol_->InitStructure();
    sol_->Zeros();

    sol_prev_->InitStructure();
    sol_prev_->Zeros();

    rhs_ = CoupVecFact.Get(
          params_["LinearAlgebra"]["NameVector"].get<std::string>() )->
          params(params_["LinearAlgebra"]);
    rhs_->Init(comm_, couplings_);
    rhs_->InitStructure();
    rhs_->Zeros();

    // Prepare boundary conditions
    dirichlet_dofs_.clear();
    dirichlet_values_.clear();

    DirichletBC bc;
    compute_dirichlet_dofs_and_values(bc, space_, 0, dirichlet_dofs_,
                                      dirichlet_values_);

    if (!dirichlet_dofs_.empty()) {
        // Correct solution with dirichlet BC
        sol_->SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
                        vec2ptr(dirichlet_values_));
    }
    // Prepare linear system
      solver_ = SolFact.Get(
            params_["LinearSolver"]["Name"].get<std::string>() )->
            params(params_["LinearSolver"]);
#else
    // Assign degrees to each element.
    std::vector<int> degrees(1, fe_degree_);

    // Initialize the VectorSpace object.
    space_.Init(degrees, *mesh_);

    // Setup couplings object.
    couplings_.Clear();
    couplings_.Init(comm_, space_.dof());
```

```
// Compute the matrix graph.
SparsityStructure sparsity;
global_asm_.compute_sparsity_structure(space_, sparsity);

couplings_.InitializeCouplings(sparsity.off_diagonal_rows,
               sparsity.off_diagonal_cols);



// Initialize matrices and vectors
mass_.Init(comm_, couplings_, la_platform, la_impl, la_matrix_format);
mass_.InitStructure(vec2ptr(sparsity.diagonal_rows),
                    vec2ptr(sparsity.diagonal_cols),
                    sparsity.diagonal_rows.size(),
                    vec2ptr(sparsity.off_diagonal_rows),
                    vec2ptr(sparsity.off_diagonal_cols),
                    sparsity.off_diagonal_rows.size());

stiff_.Init(comm_, couplings_, la_platform, la_impl, la_matrix_format);
stiff_.InitStructure(vec2ptr(sparsity.diagonal_rows),
                    vec2ptr(sparsity.diagonal_cols),
                    sparsity.diagonal_rows.size(),
                    vec2ptr(sparsity.off_diagonal_rows),
                    vec2ptr(sparsity.off_diagonal_cols),
                    sparsity.off_diagonal_rows.size());

conv_.Init(comm_, couplings_, la_platform, la_impl, la_matrix_format);
conv_.InitStructure(vec2ptr(sparsity.diagonal_rows),
                    vec2ptr(sparsity.diagonal_cols),
                    sparsity.diagonal_rows.size(),
                    vec2ptr(sparsity.off_diagonal_rows),
                    vec2ptr(sparsity.off_diagonal_cols),
                    sparsity.off_diagonal_rows.size());

source_.Init(comm_, couplings_, la_platform, la_impl);
source_.InitStructure();
source_.Zeros();

k1.Init(comm_, couplings_, la_platform, la_impl);
k1.InitStructure();
k1.Zeros();

k2.Init(comm_, couplings_, la_platform, la_impl);
k2.InitStructure();
k2.Zeros();

k3.Init(comm_, couplings_, la_platform, la_impl);
k3.InitStructure();
k3.Zeros();

k4.Init(comm_, couplings_, la_platform, la_impl);
k4.InitStructure();
k4.Zeros();

help1.Init(comm_, couplings_, la_platform, la_impl);
help1.InitStructure();
help1.Zeros();

help2.Init(comm_, couplings_, la_platform, la_impl);
help2.InitStructure();
help2.Zeros();
```

```
    help3.Init(comm_, couplings_, la_platform, la_impl);
    help3.InitStructure();
    help3.Zeros();

    help4.Init(comm_, couplings_, la_platform, la_impl);
    help4.InitStructure();
    help4.Zeros();

    s.Init(comm_, couplings_, la_platform, la_impl);
    s.InitStructure();
    s.Zeros();

    sol_.Init(comm_, couplings_, la_platform, la_impl);
    sol_.InitStructure();
    sol_.Zeros();

    rhs_.Init(comm_, couplings_, la_platform, la_impl);
    rhs_.InitStructure();
    rhs_.Zeros();

    // Prepare boundary conditions
    dirichlet_dofs_.clear();
    dirichlet_values_.clear();

    DirichletBC bc;
    compute_dirichlet_dofs_and_values(bc, space_, 0,
                                      dirichlet_dofs_, dirichlet_values_);

    if (!dirichlet_dofs_.empty()) {
        // correct solution with dirichlet BC
        sol_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
                       vec2ptr(dirichlet_values_));
    }

    // Setup linear solver
    solver_.InitControl(maxits_, abstol_, reltol_, divtol_);
    solver_.InitParameter("NoPreconditioning");
    // set the matrix to be used as the operator
    solver_.SetupOperator(mass_);
#endif
}
```

struct DirichletBC
This struct in instationary_convdiff_tutorial.cc defines the homogenous Dirichlet boundary condition.

```
,
struct DirichletBC {
    // Callback function to evaluate boundary values
    std::vector<double> evaluate(const mesh::Entity& face,
      const std::vector<Coord>& coords_on_face) const {
        return std::vector<double>(coords_on_face.size(), 0.0);
    }
};
```

3.4.5 assemble_system()

The member function assemble_system() computes the stiffness matrix and right-hand side (instationary_convdiff_tutorial.cc). The stiffness matrix, right-hand side vector and solution vector are modified to set correct Dirichlet values for the boundary DoFs.

'

```
void ConvectionDiffusionApp::assemble_system() {
#ifndef RUNGE_KUTTA
    // Assemble matrix and right-hand side vector
    ConvectionDiffusionAssembler local_asm(nu_);

    // Initialize timestepping
      local_asm.set_timestep_parameters(theta_, delta_t_);
      local_asm.set_time(time_);
      local_asm.set_prev_solution(sol_prev_);

      if (time_ == delta_t_) {
        global_asm_.assemble_matrix(space_, local_asm, *matrix_);
      }
    global_asm_.assemble_vector(space_, local_asm, *rhs_);

    if (!dirichlet_dofs_.empty()) {
        // Correct Dirichlet dofs
        matrix_->diagonalize_rows(vec2ptr(dirichlet_dofs_),
                                  dirichlet_dofs_.size(), 1.0);
        rhs_->SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
                        vec2ptr(dirichlet_values_));
    }
#else
    //assemble mass matrix
    MassAssembler mass_asm;
    global_asm_.assemble_matrix(space_, mass_asm, mass_);

    //stiffness mass matrix
    StiffnessAssembler stiff_asm;
    global_asm_.assemble_matrix(space_, stiff_asm, stiff_);

    //assemble convection matrix
    ConvectionAssembler conv_asm;
    global_asm_.assemble_matrix(space_, conv_asm, conv_);

    //assemble source vector
    SourceAssembler source_asm;
    global_asm_.assemble_vector(space_, source_asm, source_);

    if (!dirichlet_dofs_.empty()) {
        // Correct Dirichlet dofs.
        mass_.ZeroRows(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(), 1.0);
        stiff_.ZeroRows(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(), 1.0);
        conv_.ZeroRows(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(), 1.0);

        rhs_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
                       vec2ptr(dirichlet_values_));
    }
#endif
}
```

ConvectionDiffusionAssembler class

The operator for the assembling of the stiffness matrix is implemented in the class Convection-DiffusionAssembler (instationary_convdiff_tutorial.h).

'

```
void ConvectionDiffusionAssembler::operator()(const Element& element,
                                              const Quadrature<double>& quadrature,
                                              LocalMatrix& lm) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);
  // For timestepping.
  prev_ts_c_.clear();
  grad_prev_ts_c_.clear();
  evaluate_fe_function(*prev_time_sol_, 0, prev_ts_c_);
  evaluate_fe_function_gradients(*prev_time_sol_, 0, grad_prev_ts_c_);

  // Compute local matrix.
  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lm.Resize(total_dofs, total_dofs);
  lm.Zeros();

  // Loop over all quadrature points q.
  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    for (int i = 0; i < n_dofs; ++i) {       // Loop over test functions
      for (int j = 0; j < n_dofs; ++j) {     // Loop over ansatz functions
    std::vector<double> vel = get_convective_velocity(q);
    const double prod = vel[0] * grad_phi(j, q)[0] + vel[1] *
        grad_phi(j, q)[1];
    lm(dof_index(i, 0), dof_index(j, 0))
          += wq * ( phi(j, q) * phi(i, q)
                    + alpha1_ * (prod * phi(i, q)
                    + nu_ * dot(grad_phi(j, q), grad_phi(i, q))))
                * std::abs(detJ(q));
      }
    }
  }
}
```

The operator for the assembling of the right-hand side is implemented in the class Convection-DiffusionAssembler (instationary_convdiff_tutorial.h).

```
'
void ConvectionDiffusionAssembler::operator()(const Element& element,
                                          const Quadrature<double>& quadrature,
                                             LocalVector& lv) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);
  // For timestepping
  prev_ts_c_.clear();
  grad_prev_ts_c_.clear();
  evaluate_fe_function(*prev_time_sol_, 0, prev_ts_c_);
  evaluate_fe_function_gradients(*prev_time_sol_, 0, grad_prev_ts_c_);

  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lv.clear();
  lv.resize(total_dofs, 0.);

  // Compute local vector
  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    // Loop over test functions
    for(int i = 0; i < n_dofs; ++i) {
      for(int j = 0; j < n_dofs; ++j) {      // Loop over ansatz functions
    std::vector<double> vel = get_convective_velocity(q);
    const double prod = vel[0] * grad_phi(j, q)[0] + vel[1]
                            * grad_phi(j, q)[1];
```

```
       const double prev_q = prev_ts_c_[q];
       lv[dof_index(i, 0)]
         += wq * (prev_ts_c_[q] * phi(j, q) * phi(i, q)
                  + alpha1_ * get_source_term(q, time_) * phi(j, q) * phi(i, q)
                  + alpha3_ * (get_source_term(q, time_ - alpha2_)  * phi(j, q) * phi(i, q)
                                              - prev_q * prod *  phi(i, q)
                                              - nu_ * prev_q
                                              * dot(grad_phi(i, q), grad_phi(j, q)))
                  * std::abs(detJ(q));
      }
    }
  }
}
```

The function get_source_term(int q, double t) returns the value of the source for a given time.

```
'
double ConvectionDiffusionAssembler::get_source_term(int q, double t) {
    double s = 0.0;
    // Source term
    double x_q = x(q)[0] - 0.25;
    double y_q = x(q)[1] - 0.25;

    double fac = std::sqrt(x_q * x_q + y_q * y_q);
    if (fac <= 0.25) {
          s = std::exp(-pow(t, 10.0)) * std::cos(2 * M_PI * fac);
    }
    return s;
}
```

The function get_convective_velocity(int q) computes the velocity a.

```
'

std::vector<double> ConvectionDiffusionAssembler::get_convective_velocity(int q)
{
    // Vector field for rotating pulse
    std::vector<double> a(DIM, 0.0);

    double x_q = x(q)[0] - 0.5;
    double y_q = x(q)[1] - 0.5;
    double fac = std::sqrt(x_q * x_q + y_q * y_q);
    a[0] = -fac * y_q;
    a[1] = fac * x_q;

    return a;
}
```

StiffnessAssembler class

The operator for the assembling of the stiffness matrix is implemented in the class StiffnessAssembler (instationary_convdiff_tutorial.h).

```
'

void StiffnessAssembler::operator()(const Element<double>& element,
                                    const Quadrature<double>& quadrature,
                                    LocalMatrix& lm) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);

  // Compute local matrix.
  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lm.Resize(total_dofs, total_dofs);
  lm.Zeros();


  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    for(int i = 0; i < n_dofs; ++i) {        //loop over test functions
      for(int j = 0; j < n_dofs; ++j) {      //loop over ansatz functions
    lm(dof_index(i, 0), dof_index(j, 0))
          += wq * dot(grad_phi(j, q), grad_phi(i, q)) * std::abs(detJ(q));
      }
    }
  }
}
```

MassAssembler class
The operator for the assembling of the mass matrix is implemented in the class MassAssembler (instationary_convdiff_tutorial.h).

'

```
void MassAssembler::operator()(const Element<double>& element,
                               const Quadrature<double>& quadrature,
                               LocalMatrix& lm) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);

  // Compute local matrix.
  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lm.Resize(total_dofs, total_dofs);
  lm.Zeros();

  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    for(int i = 0; i < n_dofs; ++i) {        //loop over test functions
      for(int j = 0; j < n_dofs; ++j) {      //loop over ansatz functions
    lm(dof_index(i, 0), dof_index(j, 0))
          += wq * phi(j, q) * phi(i, q) * std::abs(detJ(q));
      }
    }
  }
}
```

ConvectionAssembler class
The operator for the assembling of the convection matrix is implemented in the class ConvectionAssembler (instationary_convdiff_tutorial.h).

'

```
void ConvectionAssembler::operator()(const Element<double>& element,
                                     const Quadrature<double>& quadrature,
                                     LocalMatrix& lm) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);

  // Compute local matrix.
  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lm.Resize(total_dofs, total_dofs);
  lm.Zeros();
  // Loop over all quadrature points q.
  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    for(int i = 0; i < n_dofs; ++i) {       //loop over test functions
      for(int j = 0; j < n_dofs; ++j) {     //loop over ansatz functions
    std::vector<double> vel = get_convective_velocity(q);
    const double prod = vel[0] * grad_phi(j, q)[0] + vel[1] * grad_phi(j, q)[1];
    lm(dof_index(i, 0), dof_index(j, 0)) += wq * prod * phi(i, q) * std::abs(detJ(q));
      }
    }
  }
}
```

The function get_convective_velocity(int q) computes the velocity a.

,

```
std::vector<double> ConvectionAssembler::get_convective_velocity(int q) {
  // Vector field for rotating pulse.
  std::vector<double> a(DIM, 0.0);

  // Implement the vector field for the rotating pulse!
  double x_q = x(q)[0] - 0.5;
  double y_q = x(q)[1] - 0.5;
  double fac = std::sqrt(x_q * x_q + y_q * y_q);
  a[0] = -fac * y_q;
  a[1] = fac * x_q;

  return a;
}
```

SourceAssembler class

The operator for the assembling of the source vector is implemented in the class SourceAssembler (instationary_convdiff_tutorial.h).

,

```
void SourceAssembler::operator()(const Element<double>& element,
                                 const Quadrature<double>& quadrature,
                                 LocalVector& lv) {
  AssemblyAssistant<DIM,double>::initialize_for_element(element, quadrature);

  const int num_q = num_quadrature_points();
  const int total_dofs = num_dofs_total();

  lv.clear();
  lv.resize(total_dofs, 0.);

  for (int q = 0; q < num_q; ++q) {
    const double wq = w(q);
    const int n_dofs = num_dofs(0);
    for(int i = 0; i < n_dofs; ++i) {
      lv[dof_index(i, 0)] += wq * get_source_term(q) * phi(i, q) * std::abs(detJ(q));
    }
  }
}
```

The function get_source_term(int q, double t) returns the value of the source for a given time.

,

```
double SourceAssembler::get_source_term(int q) {
  double s = 0.0;
  // Implement the source term!
  double x_q = x(q)[0] - 0.25;
  double y_q = x(q)[1] - 0.25;

  double fac = std::sqrt(x_q * x_q + y_q * y_q);
  if(fac <= 0.25)
  {
    s = std::cos(2 * M_PI * fac);
  }
  return s;
}
```

3.4.6 solve_system()

The member function solve_system() solves the linear system (instationary_convdiff_tutorial.cc). The solver is specified in the parameter file. The convection-diffusion equation is not symmetric, which means that GMRES-method is a good choice, see 3.2.

'

```
void ConvectionDiffusionApp::solve_system() {
#ifndef RUNGE_KUTTA
  // Solve linear system
  solver_->SetupOperator(*matrix_);
  solver_->Solve(*rhs_, sol_);

  // update the ghost DoFs, since in the next time step
  // they are needed in the assembly
  sol_->UpdateCouplings();
  // store old time solution (this also clones the ghost DoFs)
  sol_prev_->CloneFrom(*sol_);

#else

  // -----------------------------------------
  // setup rhs for k1
  u_.CloneFrom(sol_);
  double t = time_ - delta_t_;
  help1.Zeros();
  stiff_.VectorMult(u_, &help1);

  help2.Zeros();
  conv_.VectorMult(u_, &help2);

  help3.Zeros();
  help3.CloneFrom(help2);
  help3.Axpy(help1, nu_);

  s.Zeros();
  s.Axpy(source_, std::exp(-std::pow(t, 10.0)));

  rhs_.Zeros();
  rhs_.CloneFrom(s);
  rhs_.Axpy(help3, -1.0);
  if (!dirichlet_dofs_.empty()) {
    rhs_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
    // correct solution with dirichlet BC
    k1.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
  }
```

```
// Solve linear system for k1.
solver_.Solve(rhs_, &k1);

// ------------------------------------------
// setup rhs for k2
u_.CloneFrom(sol_);
u_.Axpy(k1, 0.5 * delta_t_);
t = time_ - 0.5 * delta_t_;

help1.Zeros();
stiff_.VectorMult(u_, &help1);

help2.Zeros();
conv_.VectorMult(u_, &help2);

help3.Zeros();
help3.CloneFrom(help2);
help3.Axpy(help1, nu_);

s.Zeros();
s.Axpy(source_, std::exp(-std::pow(t, 10.0)));

rhs_.Zeros();
rhs_.CloneFrom(s);
rhs_.Axpy(help3, -1.0);

if (!dirichlet_dofs_.empty()) {
  rhs_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
      vec2ptr(dirichlet_values_));
  // correct solution with dirichlet BC
  k2.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
      vec2ptr(dirichlet_values_));
}

// Solve linear system for k2.
solver_.Solve(rhs_, &k2);

// ------------------------------------------
// setup rhs for k3
u_.CloneFrom(sol_);
u_.Axpy(k2, 0.5 * delta_t_);
t = time_ - 0.5 * delta_t_;

help1.Zeros();
stiff_.VectorMult(u_, &help1);

help2.Zeros();
conv_.VectorMult(u_, &help2);

help3.Zeros();
help3.CloneFrom(help2);
help3.Axpy(help1, nu_);

s.Zeros();
s.Axpy(source_, std::exp(-std::pow(t, 10.0)));

rhs_.Zeros();
rhs_.CloneFrom(s);
rhs_.Axpy(help3, -1.0);

if (!dirichlet_dofs_.empty()) {
    rhs_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
    // correct solution with dirichlet BC
    k3.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
  }

  // Solve linear system for k3.
  solver_.Solve(rhs_, &k3);

  // -----------------------------------------
  // setup rhs for k4
  u_.CloneFrom(sol_);
  u_.Axpy(k3, delta_t_);
  t = time_;

  help1.Zeros();
  stiff_.VectorMult(u_, &help1);

  help2.Zeros();
  conv_.VectorMult(u_, &help2);

  help3.Zeros();
  help3.CloneFrom(help2);
  help3.Axpy(help1, nu_);

  s.Zeros();
  s.Axpy(source_, std::exp(-std::pow(t, 10.0)));

  rhs_.Zeros();
  rhs_.CloneFrom(s);
  rhs_.Axpy(help3, -1.0);

  if (!dirichlet_dofs_.empty()) {
    rhs_.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
    // correct solution with dirichlet BC
    k4.SetValues(vec2ptr(dirichlet_dofs_), dirichlet_dofs_.size(),
        vec2ptr(dirichlet_values_));
  }

  // Solve linear system for k4.
  solver_.Solve(rhs_, &k4);

  // -----------------------------------------
  // setup rhs for sol
  help4.Zeros();
  help4.Axpy(k1, 1.0);
  help4.Axpy(k2, 2.0);
  help4.Axpy(k3, 2.0);
  help4.Axpy(k4, 1.0);

  rhs_.Zeros();
  rhs_.CloneFrom(sol_);

  rhs_.Axpy(help4, delta_t_ / 6.0);

  sol_.CloneFrom(rhs_);
#endif
}
```

3.4.7 visualize(int time)

The member function visualize(int time) in instationary_convdiff_tutorial.cc writes out data for visualization. Note that HiFlow^3 has no own visualising module, so far.

,

```
void ConvectionDiffusionApp::visualize(int time) {

  int num_intervals = 2;
  ParallelCellVisualization<double> visu(space_, num_intervals, comm_, MASTER_RANK);

  // Generate filename
  std::stringstream input;
  input << num_partitions_<< "_solution_" << refinement_level_ << "_" << time;

  std::vector<double> remote_index(mesh_->num_entities(mesh_->tdim()), 0);
  std::vector<double> sub_domain(mesh_->num_entities(mesh_->tdim()), 0);
  std::vector<double> material_number(mesh_->num_entities(mesh_->tdim()), 0);

  for (mesh::EntityIterator it = mesh_->begin(mesh_->tdim());
      it != mesh_->end(mesh_->tdim());
      ++it)
  {
    int temp1, temp2;
    mesh_->get_attribute_value ( "_remote_index_", mesh_->tdim(),
                                 it->index(),
                                 &temp1);
    mesh_->get_attribute_value ( "_sub_domain_", mesh_->tdim(),
                                 it->index(),
                                 &temp2);
    material_number.at(it->index()) = mesh_->get_material_number(mesh_->tdim(), it->index());
    remote_index.at(it->index()) = temp1;
    sub_domain.at(it->index()) = temp2;
  }

#ifndef RUNGE_KUTTA
  sol_->UpdateCouplings();
  visu.visualize(EvalFeFunction<LAD>(space_, *(sol_)), "u");
#else
  sol_.UpdateCouplings();
  visu.visualize(EvalFeFunction<LAD>(space_, sol_), "u");
#endif
  // Visualize the source and convection terms and all attributes
  visu.visualize(EvalSourceFunction(space_, time_), "s");
  visu.visualize(EvalConvectionFunction(space_, 0),"a_x");
  visu.visualize(EvalConvectionFunction(space_, 1),"a_y");

  visu.visualize_cell_data(material_number, "Material␣Id");
  visu.visualize_cell_data(remote_index, "_remote_index_");
  visu.visualize_cell_data(sub_domain, "_sub_domain_");
  visu.write(input.str());

#ifdef WITH_HDF5
    //XDMF Visualization
    static bool first_time = true;
    XdmfVisualization<double> xdmf_visu(comm_, "xdmf_visualization", !first_time);
    //write grid
    if(first_time) {
      int xdmf_num_intervals = 1;
      std::string grid_name = "mesh";
      xdmf_visu.add_to_view(&space_, 0, "_sol_", "u");
```

```
        xdmf_visu.write_view(xdmf_num_intervals, grid_name);
        first_time = false;
    }

    xdmf_visu.add_timestep(time_);
#ifndef RUNGE_KUTTA
    xdmf_visu.write_solution_vector(sol_, "_sol_");
#else
    xdmf_visu.write_solution_vector(&sol_, "_sol_");
#endif

    xdmf_visu.print_xdmf();
#endif
}
```

4 Program Output

HiFlow^3 can be executed in a parallel or sequential mode which influence the generated output data. Note that the log files can be viewed by any editor.

4.1 Parallel Mode

Executing the program in parallel, for example with four processors by **mpirun -np 4 ./instationary_convdiff_tutorial** generates following output data.

- Solution data:
  - **X_solution_Y_Z.pvtu** Solution of the convection-diffusion problem for refinement level Y with default value = 5 (parallel vtk-format) with X processes. It combines the local solutions owned by the different processes to a global solution. Z is the current time step.
  - **X_solution_Y_Z_W.vtu** Local solution of the convection-diffusion problem for refinement level Y of the degrees of freedoms which belong to cells owned by process W, for W=0, 1, 2 and 3 (vtk-format). X is the number of processors. Z is the current time step.
- Log files:
  - **instationary_convdiff_tutorial_debug_log** Log file listing errors helping to simplify the debugging process. This file is empty if the program runs without errors.
  - **instationary_convdiff_tutorial_info_log** Log file listing parameters and some helpful information to control the program as for example information about the residual of the linear and non-linear solver used.
- Terminal output: Mesh refinement level.

4.2 Sequential Mode

Executing the program sequentially by **./instationary_convdiff_tutorial** following output data is generated.

- Solution data.
  - **1_solution_X_Y.vtu** Solution of the convection-diffusion problem for refinement level X with default value = 5 (vtk-format). Y is the current time step.

- Log files:
  - **instationary_convdiff_tutorial_debug_log** is a list of errors helping to simplify the debugging process. This file keeps empty if the program runs without errors.
  - **instationary_convdiff_tutorial_info_log** is a list of parameters and some helpful informations to control the program as for example information about the residual of the linear solver used.
- Terminal output: Mesh refinement level.

4.3 Visualizing the Solution

HiFlow^3 only generates output data, see section 3.4.7 , but does not visualize. The mesh/geometry data as well as the solution data can be visualized by any external program which can handle the vtk data format as e.g. the program paraview [4].

5 Numerical Example

We choose

$$\begin{aligned}
a &= \sqrt{\left(x-\frac{1}{2}\right)^2+\left(y-\frac{1}{2}\right)^2}\cdot\left(-\left(y-\frac{1}{2}\right),x-\frac{1}{2}\right)^\top, \qquad (8)\\
\nu &= 10^{-5},\\
s &= e^{-t^{10}}\cdot\begin{cases}\cos\left(2\pi\sqrt{(x-\frac14)^2+(y-\frac14)^2}\right), & \text{if}\quad \sqrt{(x-\frac14)^2+(y-\frac14)^2}\le\frac14\\ 0, & \text{else}\end{cases}.
\end{aligned}$$

Figure 1 show the convection velocity a given in (8).

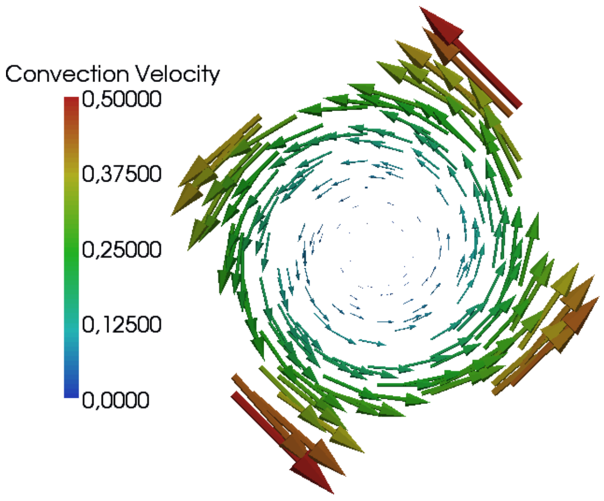

Figure 1: Convection velocity a for the instationary problem

5.1 θ-Family of Time Discretization Methods

For the explicit Euler method we tested with $\Delta t = 1$ on a six times refined grid, see Figure 2.

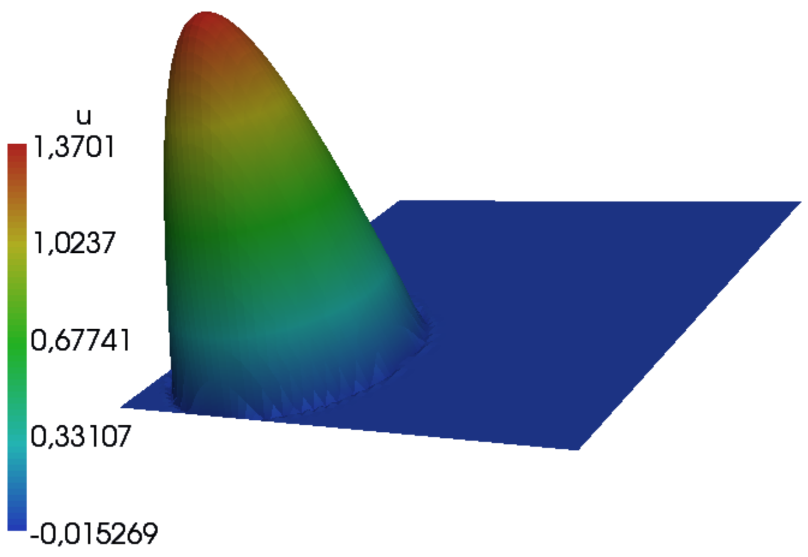

Figure 2: Solution at $T = 100$ computed by explicit Euler for $\Delta t = 1$ on a six times refined grid

As we can see the source is not transported, because the method is of too low order, even though it is stable.
For the Crank-Nicolson method we started with the same setting, see Figure 3. As we can see, the solution is stable and we get the expected helical solution.

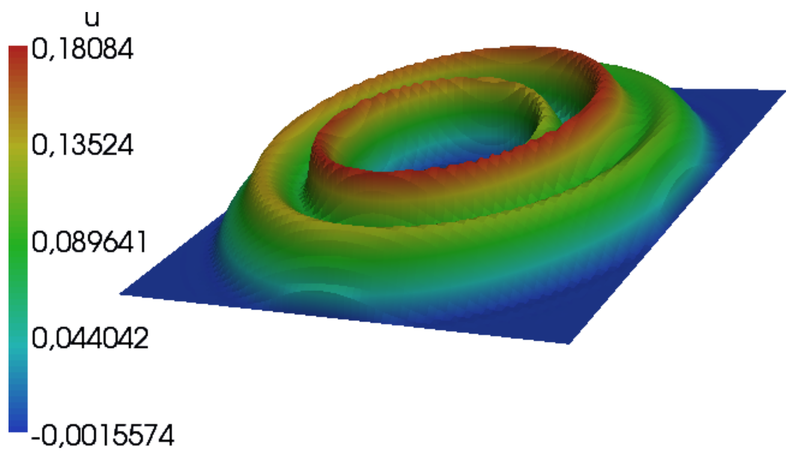

Figure 3: Solution at $T = 100$ computed by Crank-Nicolson for $\Delta t = 1$ on a six times refined grid

But the big time-step size leads to a solution that is too diffusive. Repeating the same test case with different time-step sizes and levels of refinement shows that the "waves" should be more distinctive, see Figures 4 and 5.
For the implicit Euler method we ran the same tests as for the Crank-Nicolson method except for the test on refinement level eight, see Figures 6 and 7.
This method is even more diffusive than the Crank-Nicolson, but also a bit more stable. The advantage of Crank-Nicolson is that it is compared with the explicit and implicit Euler methods of second order instead of first order only.
So the Crank-Nicolson method provides the best mixture of stability and quality of the computed solution.

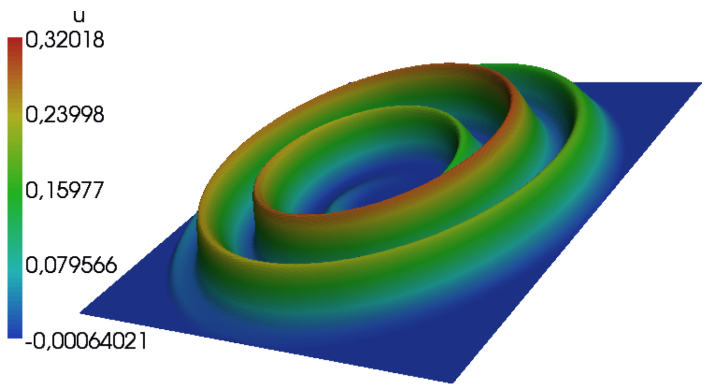

Figure 4: Solution at $T = 100$ computed by Crank-Nicolson for $\Delta t = 0.2$ on a eight times refined grid

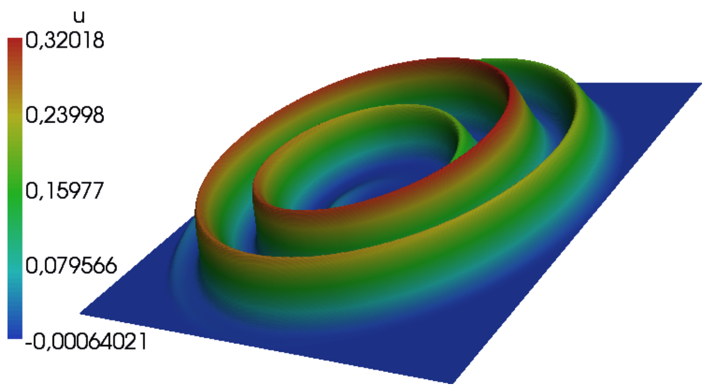

Figure 5: Solution at $T = 100$ computed by Crank-Nicolson for $\Delta t = 0.1$ on a eight times refined grid

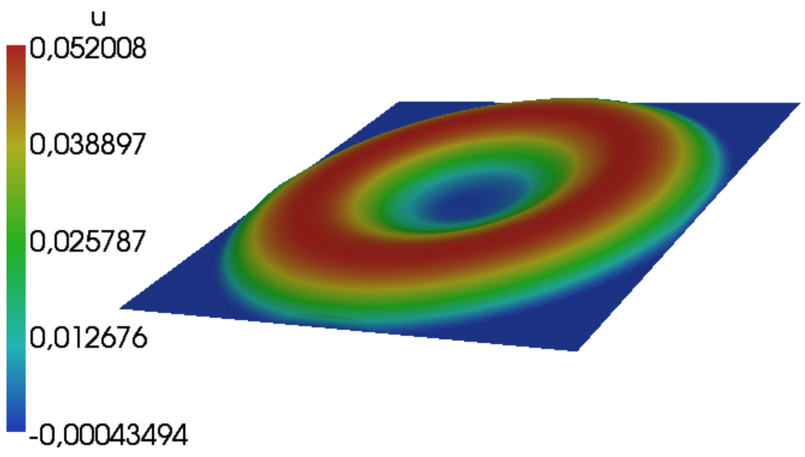

Figure 6: Solution at $T = 100$ computed by implicit Euler for $\Delta t = 1$ on a six times refined grid

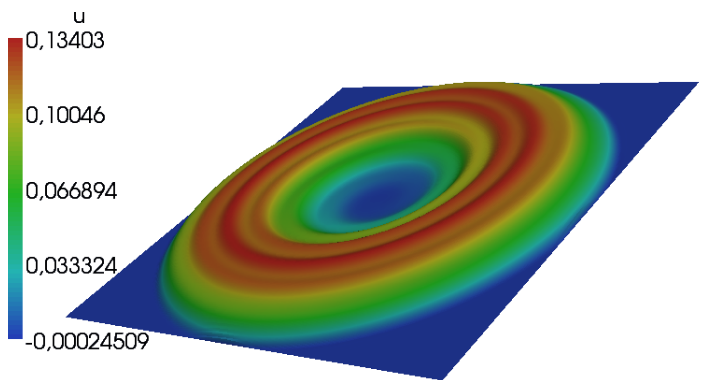

Figure 7: Solution at $T = 100$ computed by implicit Euler for $\Delta t = 0.1$ on a eight times refined grid

5.2 The classical Runge-Kutta method

This method turned out to be stable for time-step sizes smaller than or equal to $\Delta t = 0.015$. As figure 8 shows the scheme is also too diffusive. Although it is of fourth order it does not provide the accuracy of the Crank-Nicholson scheme, see below.

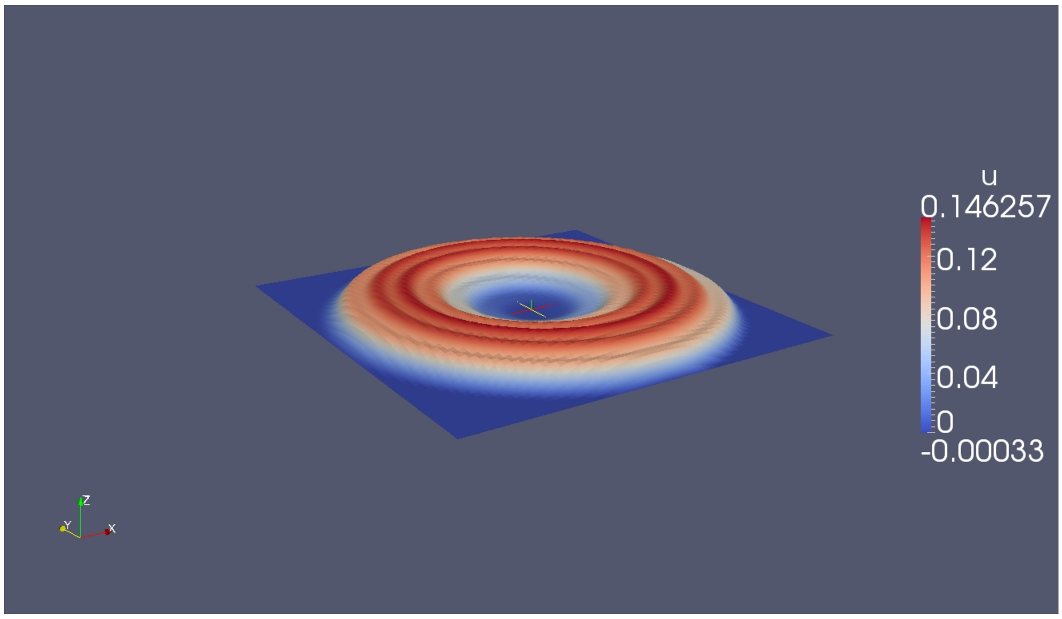

Figure 8: Solution at $T = 100$ computed by the classical Runge-Kutta method for $\Delta t = 0.01$ on a six times refined grid

5.3 Comparison of the two methods

If we compare the θ family of methods with the classical Runge-Kutta method we obtain an ambivalent result.
On the one hand the classical Runge-Kutta method is an explicit method which means that one time-step can be computed very fast. One can also use the CG method to compute the k_i, $i = 1, \ldots, 4$, appearing in the scheme instead of the GMRES solver which is needed for the θ

family. Another advantage of the classical Runge-Kutta method is that it is of fourth order instead of order one (explicit and implicit Euler) or order two (Crank-Nicholson).
On the other hand the classical Runge-Kutta methods needs significantly smaller time-step sizes than e.g. the Crank-Nicholson or implicit Euler scheme. This cirumstance not only leads to longer computation times but in particular the amount of rounding errors is potentially increased. That way the classical Runge-Kutta method looses its fourth-order-advantage compared to the Crank-Nicholson scheme although the latter is of second order only.
In summary we recommend to use the Crank-Nicholson time-stepping scheme because it provides the best compromise between stability and accuracy of the solution.

References

[1] http://www.mcs.anl.gov/research/projects/mpi.

[2] A.Valli A.Quarteroni. *Numerical Approximation of Partial Differential Equations*. Springer-Verlag, 2008.

[3] William D. Gropp. *MPI - Eine Einführung*. Oldenbourg, 2007.

[4] Amy Henderson Squillacote. *The ParaView guide: a parallel visualization application*. Kitware, 2007.