DUNE PDELab Tutorial 03
Conforming Finite Elements for a Nonlinear Heat Equation

DUNE/PDELab Team

February 5, 2021

Contents

1 Introduction

In this tutorial we extend the elliptic problem from tutorial 01 to the time dependent case. Following the method of lines a general approach to time-dependent problems is presented.

Depends On

This tutorial depends on tutorial 01.

2 PDE Problem

In this tutorial we consider the following problem:

$$\begin{aligned} \partial_t u - \Delta u + q(u) &= f && \text{in } \Omega \times \Sigma, \\ u &= g && \text{on } \Gamma_D \subseteq \partial\Omega, \\ -\nabla u \cdot \nu &= j && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D, \\ u &= u_0 && \text{at } t = 0. \end{aligned}$$

This problem is a straightforward extension of the stationary problem solved in tutorial 01. The parameter functions f, g, j may now also depend on time and (with some restrictions) the subdivision into Dirichlet and Neumann boundary can be time-dependent as well. The initial condition u_0 is a function of $x \in \Omega$.

Multiplying with a test function and integrating in space results in the following weak formulation [2]: Find $u \in L_2(t_0, t_0 + T; u_g + V(t))$:

$$\frac{d}{dt}\int_\Omega uv\,dx + \int_\Omega \nabla u \cdot \nabla v + q(u)v - fv\,dx + \int_{\Gamma_N} jv\,ds = 0 \qquad \begin{matrix} \forall v \in V(t), \\ t \in \Sigma, \end{matrix} \tag{1}$$

where $V(t) = \{v \in H^1(\Omega) \,:\, v = 0 \text{ on } \Gamma_D(t)\}$ and $H^1(\Omega) \ni u_g(t)|_{\Gamma_D} = g$. This can be written in a more compact way with residual forms as follows:

$$\frac{d}{dt} m^{\mathrm{L2}}(u, v) + r^{\mathrm{NLP}}(u, v) = 0 \quad \forall v \in V(t), t \in \Sigma.$$

where

$$m^{\mathrm{L2}}(u, v) = \int_\Omega uv\,dx$$

is the new temporal residual form, actually the L_2 inner product, and

$$r^{\mathrm{NLP}}(u, v) = \int_\Omega \nabla u \cdot \nabla v + (q(u) - f)v\,dx + \int_{\Gamma_N} jv\,ds,$$

is the spatial residual form known from tutorial 01. Under suitable assumptions it can be shown that problem (1) has a unique solution, see [2] for the linear case.

3 Finite Element Method

In order to arrive at a fully discrete formulation we follow the method of lines paradigm:

1) Choose a finite-dimensional test space $V_h(t) \subset V(t)$. Then (1) results in a system of ordinary differential equations for the coefficients $z_j(t)$ in the expansion of $u_h(t) = \sum_{j=1}^{n} (z_j(t))_j \phi_j$.

2) Choose an appropriate method to integrate the system of ordinary differential equations (ODEs).

The finite-dimensional space we choose here is just the conforming finite element space $V_h^{k,d}(\mathcal{T}_h, t)$ introduced in tutorial 01. It may now depend also on time due to the time-dependent splitting of the boundary in Dirichlet and Neumann part. Also the function $u_{h,g}(t)$ depends on time and we have $u_h(t) \in U_h(t) = u_{h,g}(t) + V_h(t)$.

For the integration of the system of ODEs, subdivide the time interval into not necessarily equidistant subintervals:

$$\overline{\Sigma} = \{t^0\} \cup (t^0, t^1] \cup \ldots \cup (t^{N-1}, t^N]$$

with $t^0 = t_0$, $t^N = t_0 + T$, $t^{k-1} < t^k$ for $1 \leq k \leq N$ and set the time step to $\Delta t^k = t^{k+1} - t^k$.

One of the simplest ODE integrators is the one-step-θ rule which reads:

$$\begin{aligned} \text{Find } u_h^{k+1} \in U_h(t^{k+1}) \text{ s.t.: } & \frac{1}{\Delta t_k}(m_h^{\mathrm{L2}}(u_h^{k+1}, v; t^{k+1}) - m_h^{\mathrm{L2}}(u_h^k, v; t^k)) + \\ & \theta r_h^{\mathrm{NLP}}(u_h^{k+1}, v; t^{k+1}) + (1-\theta) r_h^{\mathrm{NLP}}(u_h^k, v; t^k) = 0 \quad \forall v \in V_h(t^{k+1}). \end{aligned} \tag{2}$$

Reordering terms shows that this method results in the solution of a nonlinear system per time step which has the same structure as before:

$$\text{Find } u_h^{k+1} \in U_h(t^{k+1}) \text{ s.t.: } \quad r_h^{\theta,k}(u_h^{k+1}, v) + s_h^{\theta,k}(v) = 0 \quad \forall v \in V_h(t^{k+1}).$$

where

$$\begin{aligned} r_h^{\theta,k}(u, v) &= m_h^{\mathrm{L2}}(u, v; t^{k+1}) + \Delta t^k \theta r_h^{\mathrm{NLP}}(u, v; t^{k+1}), \\ s_h^{\theta,k}(v) &= -m_h^{\mathrm{L2}}(u_h^k, v; t^k) + \Delta t^k (1-\theta) r_h^{\mathrm{NLP}}(u_h^k, v; t^k). \end{aligned}$$

Implementation-wise the new residual form is comprised of a linear combination of temporal and spatial residual forms.

The one step θ method results in the implicit Euler method for $\theta = 1$, the Crank-Nicolson method for $\theta = 1/2$ and the explicit Euler method for $\theta = 0$. A large number of alternative methods are possible. In particular the class of Runge-Kutta methods is introduced below.

Runge-Kutta Methods in Shu-Osher Form

For the temporal and spatial residual forms m_h and r_h, Runge-Kutta methods can be written in the Shu-Osher form [6, 3]:

1. $u_h^{(0)} = u_h^k$.
2. For $i = 1, \ldots, s \in \mathbb{N}$, find $u_h^{(i)} \in u_{h,g}(t^k + d_i \Delta t^k) + V_h(t^{k+1})$:

$$\sum_{j=0}^{s} \Bigg[a_{ij} m_h \left(u_h^{(j)}, v; t^k + d_j \Delta t^k \right) + b_{ij} \Delta t^k r_h \left(u_h^{(j)}, v; t^k + d_j \Delta t^k \right) \Bigg] = 0 \qquad \forall v \in V_h(t^{k+1}).$$

3. $u_h^{k+1} = u_h^{(s)}$.

Here we assume that the same type of boundary condition holds through the interval $(t^k, t^{k+1}]$. The parameter s denotes the number of stages of the scheme. An s-stage scheme is given by the parameters

$$A = \begin{bmatrix} a_{10} & \ldots & a_{1s} \\ \vdots & & \vdots \\ a_{s0} & \ldots & a_{ss} \end{bmatrix}, \quad B = \begin{bmatrix} b_{10} & \ldots & b_{1s} \\ \vdots & & \vdots \\ b_{s0} & \ldots & b_{ss} \end{bmatrix}, \quad d = (d_0, \ldots, d_s)^T.$$

Explicit schemes are characterized by $a_{ij} = 0$ for $j > i$ and $b_{ij} = 0$ for $j \geq i$. *Diagonally implicit* schemes are characterized by $a_{ij} = b_{ij} = 0$ for $j > i$. Fully implicit schemes where A and B are full are not considered in PDELab. Without loss of generality it can be assumed that $a_{ii} = 1$. Moreover, some diagonally implicit schemes, in particular all those implemented in PDELab right now, satisfy $b_{ii} = b \, \forall i$. This means that in case of a linear problem the matrix is the same in all steps and needs to be assembled only once. PDELab allows you to add new schemes by specifying A, B and d. All explicit Runge-Kutta methods and most implicit Runge-Kutta methods can be brought to the Shu-Osher form. Some examples are:

- One step θ scheme (introduced above):

  $$A = \begin{bmatrix} -1 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1-\theta & \theta \end{bmatrix}, \quad d = (0,1)^T.$$

  Explicit/implicit Euler ($\theta \in \{0, 1\}$), Crank-Nicolson ($\theta = 1/2$).

- Heun's second order explicit method

  $$A = \begin{bmatrix} -1 & 1 & 0 \\ -1/2 & -1/2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}, \quad d = (0,1,1)^T.$$

- Alexander's two-stage second order strongly S-stable method [1]:

  $$A = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & \alpha & 0 \\ 0 & 1-\alpha & \alpha \end{bmatrix}, \quad d = (0, \alpha, 1)^T$$

  with $\alpha = 1 - \sqrt{2}/2$.

- Fractional step θ [4], three stage second order strongly A-stable scheme:

  $$A = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} \theta(1-\alpha) & \theta\alpha & 0 & 0 \\ 0 & \theta'\alpha & \theta'(1-\alpha) & 0 \\ 0 & 0 & \theta(1-\alpha) & \theta\alpha \end{bmatrix}, \quad d = (0, \theta, 1-\theta, 1)^T$$

  with $\theta = 1 - \sqrt{2}/2$, $\alpha = 2\theta$, $\theta' = 1 - 2\theta = 1 - \alpha = \sqrt{2} - 1$. Note also that $\theta\alpha = \theta'(1-\alpha) = 2\theta^2$.

Explicit Time Stepping Schemes

Considering the case of the explicit Euler method ($\theta = 0$) in (2) results in the problem: Find $u_h^{k+1} \in U_h(t^{k+1})$ s.t.:

$$m_h^{\mathrm{L2}}(u_h^{k+1}, v; t) - m_h^{\mathrm{L2}}(u_h^k, v; t) + \Delta t^k r_h^{\mathrm{NLP}}(u_h^k, v; t) = 0 \quad \forall v \in V_h(t^{k+1}).$$

For certain spatial schemes, e.g. finite volume or discontinuous Galerkin, and exploiting that m_h^{L2} is bilinear, the corresponding algebraic system to be solved is (block-) diagonal:

$$Dz^{k+1} = s^k - \Delta t^k q^k. \tag{3}$$

Moreover, a stability condition restricting the time step Δt^k has to be obeyed. The maximum allowable time step can be computed explicitly for the simplest schemes and depends on the mesh $\mathcal{T}_h$. For explicit time-stepping schemes therefore the following algorithm is employed:

i) While traversing the mesh assemble the vectors s^k and q^k separately and compute the maximum time step Δt^k.

ii) Form the right hand side $b^k = s^k - \Delta t^k q^k$ and "solve" the diagonal system $Dz^{k+1} = b^k$ (can be done in one step).

This procedure can be applied also to more general time-stepping schemes such as strong stability preserving Runge-Kutta methods [5].

4 Realization in PDELab

The structure of the code is very similar to that of tutorial 01. It consists of the following files:

1) The ini-file `tutorial03.ini` holds parameters read by various parts of the code which control the execution.

2) The main file `tutorial03.cc` includes the necessary C++, DUNE and PDELab header files and contains the `main` function where the execution starts. The purpose of the `main` function is to instantiate DUNE grid objects and call the `driver` function.

3) File `driver.hh` instantiates the necessary PDELab classes for solving a nonlinear instationary problem and finally solves the problem.

4) File `nonlinearheatfem.hh` contains the local operator classes `NonlinearHeatFEM` and `L2` realizing the spatial and temporal residual forms.

5) File `problem.hh` contains a parameter class which encapsulates the user-definable part of the PDE problem.

4.1 Ini-File

The ini-file contains a few new parameters compared to tutorial 01. The `fem` section contains a new parameter for the order of the time stepping scheme (so 1 would be implicit Euler, 2 a second order scheme like Alexander's 2 stage scheme):

```
[fem]
degree=2
torder=3
dt=0.02
```

and the `problem` section contains a parameter for the final time (the initial time t_0 is always zero):

```
[problem]
eta=5.0
T=2.0
```

4.2 Function `main`

The `main` function is very similar to the one in tutorial 01. In order to simplify things only the structured grids `OneDGrid` and `YaspGrid` are used.

4.3 Parameter Class in `problem.hh`

The parameter class provides all data of the PDE problem: Coefficient functions, boundary and initial conditions. They all may depend on time now in comparison to tutorial 01. In order to pass the time argument there are at least two options:

1) Extend the interface of all methods by an additional time argument.

2) Pass the evaluation time via a separate function and store the time in a private data member for subsequent spatial evaluations.

The second approach is taken in PDELab as it allows the reuse of classes from the stationary case. Interfaces of methods are not changed and only an additional method needs to be provided. This new method on the class `Problem` has the following implementation:

```
  //! Set time in instationary case
  void setTime (Number t_)
  {
    t = t_;
  }
```

The method just stores the given time in an internal data member.

The use of the time is shown by the Dirichlet boundary condition extension method implementing the function $u_g(x,t) = \sin(2\pi t) * \prod_{i=1}^{d-1} \sin^2(\pi x_i) \sin^2(10\pi x_i)$:

```
  //! Dirichlet extension
  template<typename E, typename X>
  Number g (const E& e, const X& x) const
```

```
  {
    auto global = e.geometry().global(x);
    Number s=sin(2.0*M_PI*t);
    for (std::size_t i=1; i<global.size(); i++)
      s*=sin(global[i]*M_PI)*sin(global[i]*M_PI);
    for (std::size_t i=1; i<global.size(); i++)
      s*=sin(10*global[i]*M_PI)*sin(10*global[i]*M_PI);
    return s;
  }
```

Note that there is no extra method for the initial condition. The method `g` is assumed to provide the initial condition for the initial time.

4.4 Function `driver`

We now go through the changes in the function `driver` which are due to instationarity. The first change concerns the initialization of simulation time and set-up of the parameter object:

```
  RF time = 0.0;
  RF eta = ptree.get("problem.eta",(RF)1.0);
  Problem<RF> problem(eta);
  problem.setTime(time);
```

Now a PDELab grid function can be constructed:

```
  auto glambda = [&](const auto& e, const auto& x)
    {return problem.g(e,x);};
  auto g = Dune::PDELab::
    makeInstationaryGridFunctionFromCallable(gv,glambda,problem);
```

A new function `makeInstationaryGridFunctionFromCallable` is provided since the grid function produced by it now also provides a method `setTime` to pass the time argument. Note that the lambda closure and the parameter object need to be provided as the `setTime` method is only on the parameter class and not on the lambda closure.

Setting up the grid function space and interpolating the initial condition is the same as before. Then the grid operator for the spatial part can be set up as in the stationary problem tutorial 01. We just use the new local operator `NonlinearHeatFEM`:

```
  typedef NonlinearHeatFEM<Problem<RF>,FEM> LOP;
  LOP lop(problem);
  typedef Dune::PDELab::ISTL::BCRSMatrixBackend<> MBE;
  int degree = ptree.get("fem.degree",(int)1);
  MBE mbe((int)pow(1+2*degree,dim));
  typedef Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,
                                     RF,RF,RF,CC,CC> GO0;
  GO0 go0(gfs,cc,gfs,cc,lop,mbe);
```

The temporal part is done similarly and just employs another local operator `L2`:

```
  typedef L2<FEM> TLOP;
```

```
TLOP tlop;
typedef Dune::PDELab::GridOperator<GFS,GFS,TLOP,MBE,
                                   RF,RF,RF,CC,CC> GO1;
GO1 go1(gfs,cc,gfs,cc,tlop,mbe);
```

Spatial and temporal part are now combined using the new class `OneStepGridOperator` which can assemble the linear combinations needed in the Runge-Kutta methods:

```
typedef Dune::PDELab::OneStepGridOperator<GO0,GO1> IGO;
IGO igo(go0,go1);
```

Linear solver backend and Newton method can be set up in the same way as before, just using the `OneStepGridOperator`. Then a time-stepping scheme in the form of the matrices A, B and the vector d introduced above needs to be selected. Several classes are already provided and it is very easy to write your own class:

```
Dune::PDELab::OneStepThetaParameter<RF> method1(1.0);
Dune::PDELab::Alexander2Parameter<RF> method2;
Dune::PDELab::Alexander3Parameter<RF> method3;
```

The selection mechanism for the time-stepping scheme is implemented by setting a pointer to an appropriate method:

```
int torder = ptree.get("fem.torder",(int)1);
Dune::PDELab::TimeSteppingParameterInterface<RF>*
  pmethod=&method1;
if (torder==1) pmethod = &method1;
if (torder==2) pmethod = &method2;
if (torder==3) pmethod = &method3;
if (torder<1||torder>3)
  std::cout<<"torder␣not␣in␣[1,3]"<<std::endl;
```

Note that `TimeSteppingParameterInterface` is the interface from which all time stepping parameter classes derive.

Then an object of class `OneStepMethod` can be instantiated. This class is able to compute a single time step of a general Runge-Kutta one step method.

```
typedef Dune::PDELab::OneStepMethod<RF,IGO,PDESOLVER,Z,Z> OSM;
OSM  osm(*pmethod,igo,pdesolver);
osm.setVerbosityLevel(2);
```

Before entering the time loop we set up VTK output for the instationary case. Paraview has several ways to handle sequences of output files. Here we use an advanced method that writes a special `pvd`-file providing the names of the files for each individual step and the associated absolute time. This is particularly useful when non-equidistant time steps are used. All the data files for the time steps are put in a subdirectory in order not to clog your file system. The following code creates a directory (the name is read from the ini-file):

```
    stat = mkdir(filename.c_str(),S_IRWXU|S_IRWXG|S_IRWXO);
    if( stat != 0 && stat != -1)
      std::cout << "Error:␣Cannot␣create␣directory␣"
                << filename << std::endl;
```

```
  }
int subsampling=ptree.get("output.subsampling",(int)1);
typedef Dune::SubsamplingVTKWriter<GV> VTKWRITER;
VTKWRITER vtkwriter(gv,Dune::refinementIntervals(subsampling));
typedef Dune::VTKSequenceWriter<GV> VTKSEQUENCEWRITER;
VTKSEQUENCEWRITER vtkSequenceWriter(
```

Now a `SubsamplingVTKWriter` and a `VTKSequenceWriter` can be instantiated:

```
int subsampling=ptree.get("output.subsampling",(int)1);
typedef Dune::SubsamplingVTKWriter<GV> VTKWRITER;
VTKWRITER vtkwriter(gv,Dune::refinementIntervals(subsampling));
typedef Dune::VTKSequenceWriter<GV> VTKSEQUENCEWRITER;
VTKSEQUENCEWRITER vtkSequenceWriter(
  std::make_shared<VTKWRITER>(vtkwriter),filename,filename,"");
```

filled with data

```
typedef Dune::PDELab::VTKGridFunctionAdapter<ZDGF> VTKF;
vtkSequenceWriter.addVertexData(std::shared_ptr<VTKF>(
                            new VTKF(zdgf,"solution")));
```

and the first file is written

```
vtkSequenceWriter.write(time,Dune::VTK::appendedraw);
```

Now the time loop is under user control:

```
RF T = ptree.get("problem.T",(RF)1.0);
RF dt = ptree.get("fem.dt",(RF)0.1);
while (time<T-1e-8)
  {
    // assemble constraints for new time step
    problem.setTime(time+dt);
    Dune::PDELab::constraints(b,gfs,cc);

    // do time step
    Z znew(z);
    osm.apply(time,dt,z,g,znew);

    // accept time step
    z = znew;
    time+=dt;

    // output to VTK file
    vtkSequenceWriter.write(time,Dune::VTK::appendedraw);
  }
```

First the final time and time step are read from the parameter file. Then, for each time step the constraints are reassembled. Note that the boundary condition type is not supposed to change *within* a time step, so all stages of the Runge-Kutta Method will use the same constraints (but not necessarily the same values at the Dirichlet boundary).

Then, the `apply` method of the one step method is used to advance one step in time. Its arguments are the current `time`, the size of the time step `dt`, the coefficient vector `z` of the old time step, the boundary condition function `g` used to interpolate boundary conditions for the stages and, as last argument, the new coefficient vector `znew` to be computed as a result.

In order to advance to the next time step the coefficient vector is copied and the time variable is advanced. Finally, the output file for the time step is written.

Intentionally, the time loop is under user control to allow other things to be done, e.g. choose adaptive time step size, proceeding to important absolute times, writing only every nth output file and so on.

4.5 Spatial Local Operator

The file `nonlinearheatfem.hh` contains the two local operators for the spatial and the temporal part. The class `NonlinearHeatFEM` implements the spatial part. It is actually very easy since the main part can be reused from the `NonlinearPoissonFEM` local operator implemented in tutorial 01 through public inheritance:

```
template<typename Param, typename FEM>
class NonlinearHeatFEM :
  public NonlinearPoissonFEM<Param,FEM>,
  public Dune::PDELab::
    InstationaryLocalOperatorDefaultMethods<double>
{
  Param& param;
public:
  //! Pass on constructor
  NonlinearHeatFEM (Param& param_, int incrementorder_=0)
    : NonlinearPoissonFEM<Param,FEM>(param_,incrementorder_),
    param(param_)
  {}
  //! set time for subsequent evaluation
  void setTime (double t) {
    param.setTime(t);
  }
};
```

Just a few methods need to be added to the local operator. Default implementations can be inherited from `InstationaryLocalOperatorDefaultMethods`. Here only the method `setTime` needs to be implemented to pass on the time to the parameter class for subsequent evaluations.

The following methods need to be provided by a local operator for instationary computations:

```
void setTime (R t_)
```

to set the time for all subsequent method calls.

```
R getTime () const
```

to get the time set previously.

```
void preStep (RealType time, RealType dt, int stages)
```

This method is called at the beginning of a time step, before all stages.

```
void postStep ()
```

This method is called at the end of a time step, after all stages.

```
void preStage (RealType time, int r)
```

This method is called at the beginning of a Runge-Kutta stage, where r is the number of the stage (starting with $r = 1$).

```
int getStage () const
```

Return the number of the current stage.

```
void postStage ()
```

Called after the intermediate result of a stage has been computed.

```
RealType suggestTimestep (RealType dt) const
```

In an explicit scheme this method can be called at the end of stage 1 to suggest a stable time step for the explicit scheme.

4.6 Temporal Local Operator

The temporal local operator needs to implement the residual form related to the L_2-inner product. It could also be used to compute an L_2 projection of some function.

Like in the spatial local operator we provide a finite element map as template parameter and derive from some helpful classes to provide default implementations of some methods:

```
template<typename FEM>
class L2
  : public Dune::PDELab::FullVolumePattern,
    public Dune::PDELab::LocalOperatorDefaultFlags,
    public Dune::PDELab::
  InstationaryLocalOperatorDefaultMethods<double>
```

The following flag indicates that we provide a `pattern_volume` method specifying the sparsity pattern. Actually, this method is provided from the base class `FullVolumePattern`:

```
  enum { doPatternVolume = true };
```

The next flag indicates that just volume integrals depending on trial and test functions are required:

```
  enum { doAlphaVolume = true };
```

The implementation of `alpha_volume` is very similar to the reaction term in tutorial 01. For efficiency reasons also `jacobian_volume` is provided as well as the `*_apply_*` variants for matrix-free operation (which might be very useful here since the mass matrix is well conditioned).

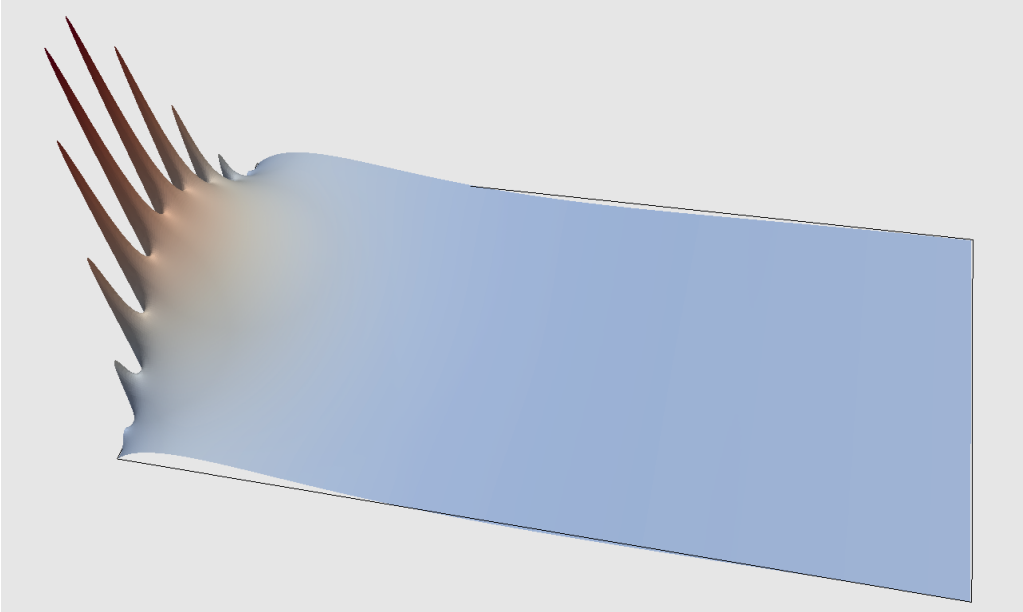

Figure 1: Solution of the nonlinear heat equation.

4.7 Running the Example

The example solves the nonlinear heat equation in 1, 2 and 3 space dimensions. A two-dimensional result is illustrated in Figure 1. On the left boundary $x = 0$ a function depending on time t and y-coordinate is given, all the other boundaries are of Neumann type with zero flux. The solution exhibits the typical smoothing effect of parabolic problems in space and time.

5 Outlook

Several ideas could be pursued from this example:

- Experiment with a rough initial condition, observe smoothing effect in time.
- Explore the monotonicity properties of the scheme by providing discontinuous initial conditions and testing different polynomial degrees and temporal orders.

References

[1] R. Alexander. Diagonally implicit Runge-Kutta methods for stiff O. D. E.'s. *SIAM Journal on Numerical Analysis*, 14(6):1006–1021, 1977.

[2] A. Ern and J.-L. Guermond. *Theory and practice of finite element methods*. Springer, 2004.

[3] S. Gottlieb, D.I. Ketcheson, and C.W. Shu. *Strong Stability Preserving Runge-Kutta and Multistep Time Discretizations*. World Scientific, 2011.

[4] Dominik Meidner and Thomas Richter. A posteriori error estimation for the fractional step theta discretization of the incompressible navier–stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 288:45 – 59, 2015.

[5] C. W. Shu. Total-variation-diminishing time discretizations. *SIAM J. Sci. Stat. Comput.*, 9:1073, 1988.

[6] Chi W. Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *J. Comput. Phys.*, 77:439–471, 1988.