“PDE optimization of the heat equation with time-varying boundary conditions”

December 4, 2017; rev. January 28, 2018

Simon Pirkelmann

1 Setting

Consider the following equation

$$y_t - \alpha\Delta y + w\nabla y = 0 \text{ on } Q := \Omega \times [0,T] \tag{1}$$

where $y : Q \to \mathbb{R}$ is the temperature, $\alpha \in \mathbb{R}$ is the diffusion coefficient and $w : [0,T] \to \Omega$ is the velocity field. We use the shorthand $y_t = \frac{\partial y}{\partial t}$ to denote the time derivative.

Let Ω be the domain. The boundary is partitioned in an boundary Γ_{out} where some outside temperature is prescribed and a control boundary Γ_c. On one part of the boundary a controllable t is applied which is described by a Dirichlet condition

$$y = u \text{ on } \Gamma_c. \tag{2}$$

On the other part we have

$$y = y_{out} \text{ on } \Gamma_{out} \tag{3}$$

where $\frac{\partial y}{\partial n}$ is the derivative of y in normal direction. We approximate the Dirichlet boundary conditions by using the following Robin type boundary condition instead

$$-\beta\frac{\partial y}{\partial n} = \gamma(y - z) \text{ on } \Gamma. \tag{4}$$

By choosing $\gamma := 10^3$, $\beta := 1$ and $z := \begin{cases} y_{out} & \text{on } \Gamma_{out} \\ u & \text{on } \Gamma_c \end{cases}$ we can approximate the Dirichlet boundary conditions. This will also allow us to extend the setting more easily in the future.

2 Numerical simulation of the heat equation

We simulate the equation using the finite element method.

2.1 Weak form

For the weak form we first discretize in time by picking a sampling rate $\Delta t > 0$. We define $y_k := y(\cdot, t_0 + \Delta tk)$, $y_{out,k} := y_{out}(\cdot, t_0 + \Delta tk)$, $u_k := u(t_0 + \Delta tk)$, $z_k := z(t_0 + \Delta tk)$ for $k \in \{0, 1, \dots, N\}$.

The initial value y_0 is given. To compute the next state y_{k+1} for each $k \in \{0, 1, \dots, N-1\}$ we replace y_t in equation (**??**) by $\frac{y_{k+1} - y_k}{\Delta t}$ and y by y_{k+1} using backward Euler. Multiplying with a test function v and integrating over the domain Ω yields

$$\int_\Omega \frac{y_{k+1} - y_k}{\Delta t} v \, dx - \alpha \int_\Omega \Delta y_{k+1} v \, dx + \int_\Omega w_k \nabla y_{k+1} v \, dx = 0 \text{ for } k \in \{0, 1, \dots, N-1\}. \tag{5}$$

Using partial integration on the second integral we get

$$\int_\Omega \frac{y_{k+1}-y_k}{\Delta t} v\,dx - \alpha \int_\Gamma \frac{\partial y_{k+1}}{\partial n} v\,ds + \alpha \int_\Omega \nabla y_{k+1}\cdot \nabla v\,dx + \int_\Omega w_k \nabla y_{k+1} v\,dx = 0 \tag{6}$$

Substituting the boundary condition (**??**) we obtain

$$\int_\Omega \frac{y_{k+1}-y_k}{\Delta t} v\,dx + \alpha \int_\Gamma \frac{\gamma}{\beta}(y_{k+1}-z_k) v\,ds + \alpha \int_\Omega \nabla y_{k+1}\cdot \nabla v\,dx + \int_\Omega w_k \nabla y_{k+1} v\,dx = 0 \tag{7}$$

TODO: Strictly speaking, for backward Euler we may actually have to use z_{k+1} and w_{k+1} here instead. Think about this!
Substituting the definition of z_k on the different parts Γ_{out} and Γ_c of the boundary we get

$$\begin{aligned}\int_\Omega \frac{y_{k+1}-y_k}{\Delta t} v\,dx + \alpha \int_{\Gamma_c} \frac{\gamma}{\beta}(y_{k+1}-u_k) v\,ds + \alpha \int_{\Gamma_{out}} \frac{\gamma}{\beta}(y_{k+1}-y_{out,k}) v\,ds \\ +\alpha \int_\Omega \nabla y_{k+1}\cdot \nabla v\,dx + \int_\Omega w_k \nabla y_{k+1} v\,dx = 0\end{aligned} \tag{8}$$

2.2 Implementation in FEniCS

The above variational equation is implemented in FEniCS to obtain the system matrices for use in the optimization.

Reordering the terms in (**??**) by terms depending on the state y_{k+1} and all external terms we get:

$$\begin{aligned}\underbrace{\int_\Omega \frac{y_{k+1}}{\Delta t} v + \alpha \nabla y_{k+1}\cdot\nabla v\,dx + \alpha\int_\Gamma \frac{\gamma}{\beta} y_{k+1} v\,ds}_{a} + \underbrace{\int_\Omega w_k \nabla y_{k+1} v\,dx}_{f_w} \\ = \underbrace{\int_\Omega \frac{y_k}{\Delta t} v\,dx}_{f_y} + \underbrace{\alpha \int_{\Gamma_c} \frac{\gamma}{\beta} u_k v\,ds}_{f_u} + \underbrace{\alpha \int_{\Gamma_{out}} \frac{\gamma}{\beta} y_{out,k} v\,ds}_{f_{y,out}}\end{aligned} \tag{9}$$

This is a linear system of equations which we can explicitely assemble in FEniCS using the `assemble()` function. We obtain the matrices and vectors

$$A \in \mathbb{R}^{n_y\times n_y}, B_w \in \mathbb{R}^{n_y\times n_y} B_y \in \mathbb{R}^{n_y\times n_y}, b_u \in \mathbb{R}^{n_y}, b_{y,out} \in \mathbb{R}^{n_y},$$

where n_y is the number of finite elements used in the spatial discretization. Equation (**??**) then corresponds to the linear system

$$A y_{h,k+1} + w_k B_w y_{h,k+1} = B_y y_{h,k} + b_u u_k + b_{y,out} y_{out,k}, \tag{10}$$

making use of the fact that u_k and $y_{out,k}$ are constant on the boundary. The system matrices and vectors are generated in FEniCS and then saved to a file for subsequent use in MATLAB.

```
# Prepare a mesh
mesh = UnitIntervalMesh(100)

# specify subdomains for the boundary
left = CompiledSubDomain("near(x[0], 0.)")
```

```
right = CompiledSubDomain("near(x[0], 1.)")
boundary_parts = MeshFunction("size_t", mesh, mesh.topology().dim() - 1
                                    )
left.mark(boundary_parts, 0)    # boundary part where control is
                                    applied
right.mark(boundary_parts, 1)   # boundary part for outside temperature
ds = Measure("ds", subdomain_data=boundary_parts)

# Expression for velocity field
class VelocityFieldExpression(Expression):
    def eval(self, value, x):
        value[0] = -1.0

    def value_shape(self):
        return (1,)

# Define function space
U = FunctionSpace(mesh, "Lagrange", 1)
W = VectorFunctionSpace(mesh, 'P', 1, dim=1)

# Define test and trial functions
v = TestFunction(U)
y = TrialFunction(U)
y0 = TrialFunction(U)
w = Function(W)
e = VelocityFieldExpression(domain=mesh, degree=1)

# Define constants
alpha = Constant(0.75)
beta = Constant(1.0)
gamma = Constant(1.0e3)
u = Constant(1.0)
y_out = Constant(1.0)
w = interpolate(e, W)

# Define variational formulation
a = (y / k * v + alpha * inner(grad(y), grad(v))) * dx + alpha * gamma/
                                    beta * y * v * ds
f_w = dot(w, grad(y)) * v * dx
f_y = y0 / k * v * dx
f_u = alpha * gamma/beta * u * v * ds(0)
f_y_out = alpha * gamma/beta * y_out * v * ds(1)

# Assemble system matrices
A = assemble(a)
B_w = assemble(f_w)
B_y = assemble(f_y)
b_u = assemble(f_u)
b_y_out = assemble(f_y_out)

# output matrices for use in matlab optimization
scipy.io.savemat('sys.mat', {'A': A.array(), 'B_y': B_y.array(), 'B_w':
                                     B_w.array(), 'b_u': b_u.array(), '
                                    b_y_out': b_y_out.array(), 'u': us,
                                     'y_out': y_outs})
```

3 Optimal Control Problem

Our goal is to solve the following problem:

$$
\begin{aligned}
\min_{y,u,w} J(y,u,w) =& \frac{\varepsilon}{2}\int_\Omega (y(x,T)-y_\Omega(T,x))^2\,dx + \frac{\varepsilon}{2}\int_0^T\int_\Omega (y(x,t)-y_\Omega(t,x))^2\,dx\,dt \\
&+\frac{1}{2}\int_0^T\int_{\Gamma_c}(u(x,t)-u_ref(x,t))^2\,ds\,dt + \frac{1}{2}\int_0^T w(t)^2\,dt \\
&\text{s.t.}(\mathbf{??}),(\mathbf{??}) \\
&\quad \underline{u}(x,t)\le u(x,t)\le \overline{u}(x,t) \\
&\quad \underline{y}(x,t)\le y(x,t)\le \overline{y}(x,t) \text{ on } \Omega_y
\end{aligned}
$$

where $\Omega_y \subset \Omega$.
The problem is solved by the First-Discretize-Then-Optimize-Approach. We replace y by its finite element approximations $y_{h,k}$ for each time point $t = k\Delta t, k \in \{0,\dots,N\}$. Let $y_h = (y_{h,0},\dots,y_{h,N})$, $u = (u_0,\dots,u_{N-1})$, $w = (w_0,\dots,w_{N-1})$ and $y_{out} = (y_{out,0},\dots,y_{out,N-1})$. We end up with a finite dimensional optimal control problem which reads:

$$
\begin{aligned}
&\min_{y_h,u,w} J(y_h,u,w) = \sum_{k=0}^{N-1}\Bigg(\frac{\varepsilon}{2}(y_{h,k}-y_{\Omega,k})^T Q(y_{h,k}-y_{\Omega,k}) + \frac{1}{2}(u_k-u_{ref,k})^T R(u_k-u_{ref,k}) \\
&+\frac{1}{2}w_k^T W w_k\Bigg) + \frac{\varepsilon}{2}(y_{h,N}-y_{\Omega,N})^T Q(y_{h,N}-y_{\Omega,N}) \\
&\text{s.t. } A y_{h,k+1} + w_k B_w y_{h,k+1} = B_y y_{h,k} + b_u u_k + b_{y,out} y_{out,k} \text{ for } k\in\{0,\dots,N-1\} \\
&\quad\ \underline{y}_{h,k,i} \le y_{h,k,i} \le \overline{y}_{h,k,i} \text{ for } k \in \{0,\dots,N\}, i\in\mathcal{I}_{\Omega_y} \\
&\quad\ \underline{u}_k \le u_k \le \overline{u}_k \text{ for } k\in\{0,\dots,N-1\}
\end{aligned}
$$

where $\mathcal{I}_{\Omega_y}$ is the set of all indices with finite element nodes within the set Ω_y. When using Lagrange finite elements the degrees of freedom of the FE approximation y_h of the state y correspond to the value of y_h at the finite element nodes. This means we can enforce the state constraint pointwise at the finite element nodes, simply by constraining the corresponding elements of y_h.

In the following it is described how the open-loop optimal control problems are solved in MATLAB with its `fmincon` solver and in C++ with Ipopt, respectively. The implementations are slightly different. In MATLAB linear and nonlinear constraints can be treated seperately. For easier implementation we introduced an additional variable for the nonlinear part of the state equation.
In Ipopt this is not necessary, because linear constraints are not treated any different than nonlinear constraints in the problem formulation required by the solver.

3.1 Implementation in MATLAB

Introducing $z = (y_h,u,w,\nu) \in \mathbb{R}^{n_z}$, where $n_z = (N+1)n_y + Nn_u + Nn_w + Nn_y$ the equality constraint of the optimization problem can be written as a single linear equation

$$A_{eq}z = b_{eq} \tag{11}$$

where

$$A_{eq} = \underbrace{\begin{pmatrix} -B_y & A & & & & -b_u & & & & 0 & & & & I & & \\ & -B_y & A & & & & -b_u & & & & 0 & & & & I & \\ & & \ddots & & & & & \ddots & & & & \ddots & & & & \ddots \\ & & & -B_y & A & & & & -b_u & & & & 0 & & & I \end{pmatrix}}_{\in \mathbb{R}^{Nny \times n_z}} \tag{12}$$

$$b_{eq} = \underbrace{\begin{pmatrix} b_{y,out}y_{out,1} & & & \\ & b_{y,out}y_{out,2} & & \\ & & \ddots & \\ & & & b_{y,out}y_{out,N} \end{pmatrix}}_{\in \mathbb{R}^{Nny \times N}} \in \mathbb{R}^{Nn_y} \tag{13}$$

where n_y, n_u and n_w denote the dimensions of $y_{h,k}$, u_k and w_k, respectively. Note that this is a sparse linear system and should be treated accordingly (i.e. only allocate memory for nonzero matrix entries).

In addition we have the nonlinear equality constraints for which we introduce an additional variable $\nu_k = w_k B_w y_{h,k+1}$ for all $k \in \{0, \dots, N-1\}$. Then the nonlinear constraints can be written in the form $c_{eq}(z) = 0$ with a nonlinear function

$$c_{eq}(z) = \begin{pmatrix} \nu_0 - w_0 B_w y_{h,1} \\ \vdots \\ \nu_{N-1} - w_{N-1} B_w y_{h,N} \end{pmatrix} \in \mathbb{R}^{Nn_y}. \tag{14}$$

NOTE: Using the additional variable ν_k almost doubles the size of the optimization variable z. This has been done because it was a quick and simple way to implement the problem and since the MATLAB implementation is only used for testing purposes anyway, but it is not a very efficient solution. In the C++ implementation described in the next subsection this is done differently.

The optimization algorithm also needs to be provided with the gradient and (optionally) the hessian of the cost functional J. Those quantities are given by

$$\nabla J(z) = \begin{pmatrix} \varepsilon Q(y_{h,0} - y_{\Omega,0}) \\ \vdots \\ \varepsilon Q(y_{h,N} - y_{\Omega,N}) \\ R(u_0 - u_{ref,0}) \\ \vdots \\ R(u_{N-1} - u_{ref,N-1}) \\ Ww_0 \\ \vdots \\ Ww_{N-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^{n_z}, \tag{15}$$

$$\nabla^2 J(z) = \begin{pmatrix} \varepsilon Q & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & \varepsilon Q & & & & & & & \\ & & & R & & & & & & \\ & & & & \ddots & & & & & \\ & & & & & R & & & & \\ & & & & & & W & & & \\ & & & & & & & \ddots & & \\ & & & & & & & & W & \\ & & & & & & & & & O_{Nn_y \times Nn_y} \end{pmatrix} \in \mathbb{R}^{n_z \times n_z}, \tag{16}$$

where $O_{Nn_y \times Nn_y}$ is a zero matrix. Note that the hessian is also a sparse matrix.

Finally we also have to provide the Jacobian of the nonlinear constraint function c_{eq}. This is given by

$$J_{c_{eq}}(z) = \underbrace{\begin{pmatrix} 0 & -w_0 B_w & & & -B_w y_1 & & I & \\ & & \ddots & O_{Nn_y \times Nn_u} & & \ddots & & \ddots \\ & & & -w_{N-1} B_w & & -B_w y_N & & I \end{pmatrix}}_{\in \mathbb{R}^{Nn_y \times n_z}} \tag{17}$$

For implementation in MATLAB this is all that is necessary for the optimization to run.

NOTE: Only the Hessian of the objective function is used in the implementation. I tried to use the Hessian of the Lagrange Function, but could not observe a speedup which is why I refrain from going into any more detail here. The Hessian of the objective function seems to be a close enough approximation that the speedup the exact Hessian could provide is offset by the effort it takes to compute it. However, I did not try to optimize the way the Hessian of the Lagrange function is computed, so there may still be some way for improvement here. I chose not to do this, because the MATLAB implementation will hardly be able to compete with the C++ implementation anyway.

3.2 Implementation in C++

In the Ipopt implementation the optimization variable is $z = (y_h, u, w) \in \mathbb{R}^{n_z}$, where $n_z = (N+1)n_y + Nn_u + Nn_w$. Here we implement the nonlinear system dynamics directly, without introducing an additional variable as we did in the MATLAB implementation. The function for the nonlinear constraints is given by

$$g(z) = \begin{pmatrix} Ay_{h,1} + w_0 B_w y_{h,1} - B_y y_{h,0} - b_u u_0 - b_{y,out} y_{out,0} \\ \vdots \\ Ay_{h,N} + w_{N-1} B_w y_{h,N} - B_y y_{h,N-1} - b_u u_{N-1} - b_{y,out} y_{out,N-1} \end{pmatrix} \in \mathbb{R}^{Nn_y}. \tag{18}$$

The Jacobian of g is given by

$$\nabla g(z) = \underbrace{\begin{pmatrix} -B_y & A+w_0B_w & & & & & -b_u & & & & B_w y_{h,1} & & & \\ & -B_y & A+w_1B_w & & & & & -b_u & & & & B_w y_{h,2} & & \\ & & & \ddots & & & & & \ddots & & & & \ddots & \\ & & & & -B_y & A+w_{N-1}B_w & & & & -b_u & & & & B_w y_{h,N} \end{pmatrix}}_{\in \mathbb{R}^{Nn_y \times n_z}} \tag{19}$$

The gradient of the objective function is given by

$$\nabla J(z) = \begin{pmatrix} \varepsilon Q(y_{h,0} - y_{\Omega,0}) \\ \vdots \\ \varepsilon Q(y_{h,N} - y_{\Omega,N}) \\ R(u_0 - u_{ref,0}) \\ \vdots \\ R(u_{N-1} - u_{ref,N-1}) \\ Ww_0 \\ \vdots \\ Ww_{N-1} \end{pmatrix} \in \mathbb{R}^{n_z}, \tag{20}$$

and the Hessian of the objective function by

$$\nabla^2 J(z) = \begin{pmatrix} \varepsilon Q & & & & & & & & \\ & \ddots & & & & & & & \\ & & \varepsilon Q & & & & & & \\ & & & R & & & & & \\ & & & & \ddots & & & & \\ & & & & & R & & & \\ & & & & & & W & & \\ & & & & & & & \ddots & \\ & & & & & & & & W \end{pmatrix} \in \mathbb{R}^{n_z \times n_z}. \tag{21}$$

For the Hessian of the Lagrange function we additionally need to specify the matrices $\nabla^2 g_i(z)$. Those are of the form

$$\nabla^2 g_k(z) = \begin{array}{c|ccccccccccccc} & y_0 & \cdots & y_{i+1} & \cdots & y_N & u_0 & \cdots & u_{N-1} & w_0 & \cdots & w_i & \cdots & w_{N-1} \\ \hline y_0 & & & & & & & & & & & & & \\ \vdots & & & & & & & & & & & & & \\ y_{i+1} & & & & & & & & & & & B_w(j,:)^T & & \\ \vdots & & & & & & & & & & & & & \\ y_N & & & & & & & & & & & & & \\ u_0 & & & & & & & & & & & & & \\ \vdots & & & & & & & & & & & & & \\ u_{N-1} & & & & & & & & & & & & & \\ w_0 & & & & & & & & & & & & & \\ \vdots & & & & & & & & & & & & & \\ w_i & & & B_w(j,:) & & & & & & & & & & \\ \vdots & & & & & & & & & & & & & \\ w_{N-1} & & & & & & & & & & & & & \end{array} \tag{22}$$

for $k \in \{1, \ldots, Nn_y\}$ where $B_w(j,:)$ denotes the $j-$th row of the matrix B_w, i.e. we have

$$\nabla^2 g_k(z)_{(N+1)n_y+Nn_U+i+1,(i+1)n_y+l} = B_w(j,l) \tag{23}$$

if $k = in_y + j$ with $i \in \{0, \ldots, N-1\}$ and $j, l \in \{1, \ldots, n_y\}$. The Hessian of the Lagrange functional is then given by

$$\sigma_f \nabla^2 J(z) + \sum_{k=1}^{Nn_y} \lambda_k \nabla^2 g_k(z) \tag{24}$$

see also `https://www.coin-or.org/Ipopt/documentation/node22.html`.

NOTE: When the convection term is present, the Jacobian and Hessian matrices are no longer constant!

NOTE: Before implementing the exact Hessian of the Lagrangian we should try using the Hessian of the objective function as an approximation and the approximation by a quasi-Newton method like BFGS (set the option `hessian_approximation` to `limited-memory` for this, see `https://www.coin-or.org/Ipopt/documentation/node31.html`).

4 Example

Consider the following example. Let Ω be the unit interval $[0,1]$ and let $\Omega_y := [\frac{1}{4}, \frac{3}{4}]$ and let Γ_{out} be the left boundary (for $x = 0$) and Γ_c be the right boundary (for $x = 1$). The domains and subdomains are illustrated in Figure **??**.

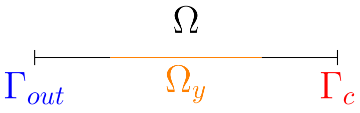

Figure 1: Illustration of the domain.

The mesh is discretized by $n_y = 100$ finite elements. The dimension of the controls is $n_u = 1$, $n_w = 1$. We pick a sampling rate $\Delta t = 10^{-2}$. At the boundary Γ_{out} we have the time-varying boundary value function

$$y_{out}(t) = \frac{1}{2} + \frac{1}{3}\sin(10t). \tag{25}$$

For discrete time points $t = k\Delta t$ this corresponds to

$$y_{out,k} = \frac{1}{2} + \frac{1}{3}\sin(\frac{k}{10}). \tag{26}$$

We choose the parameters of the PDE system

$$\begin{aligned} \alpha &= 0.5 \\ \beta &= 1 \\ \gamma &= 10^6 \end{aligned}$$

and parameters for the optimal control problem

$$
\begin{aligned}
\varepsilon &= 10^{-3}\\
Q &= I_{n_y \times n_y}\\
R &= I_{n_u \times n_u} = 1\\
W &= I_{n_w \times n_w} = 1\\
N &= 10\\
y_{\Omega,k} &= 0.5 \text{ for } k \in \{0,\dots,N\}\\
u_{ref,k} &= 0.5 \text{ for } k \in \{0,\dots,N-1\}.
\end{aligned}
$$

The constraints for control and state are given by

$$
\begin{aligned}
\underline{y}_{h,k,i} &= 0.35 \text{ for } k \in \{0,\dots,N\}, i \in \mathcal{I}_{\Omega_y}\\
\overline{y}_{h,k,i} &= 0.65 \text{ for } k \in \{0,\dots,N\}, i \in \mathcal{I}_{\Omega_y}\\
\underline{u}_k &= 0.25 \text{ for } k \in \{0,\dots,N-1\}\\
\overline{u}_k &= 0.75 \text{ for } k \in \{0,\dots,N-1\}.
\end{aligned}
$$

In the case that we use Langrange finite elements we have that $\mathcal{I}_{\Omega_y} = \{\frac{n_y}{4},\dots,\frac{3n_y}{4}\}$.