Optimal Control of the Initial Condition for the KS Equation With LibPFASST

Sebastian Götschel

October 1, 2020

1 Problem setting

We are considering the 1D Kuramoto-Sivashinsky equation

$$y_t = -yy_x - y_{xx} - y_{xxxx}, \quad x \in \Omega := [0, 32\pi] \tag{1}$$

with periodic boundary conditions on some time interval $t \in [0, T]$. Given some measured data (in the optimal control setting called *desired state*) $y_d(t, x)$, the aim is to identify an initial condition $y(0, x) = u(x)$ such that the solution $y(t, x; u)$ minimizes the deviation from the desired state. We will often refer to u as the *control*. Put into formula, we want to minimize the *objective functional*

$$J(y, u) := \frac{1}{2} \int_0^T \int_\Omega (y - y_d)^2 \mathrm{d}x \; \mathrm{d}t + R(u), \tag{2}$$

where R is some generic regularization term, e.g., $R(u) = \frac{\lambda}{2} \|u\|^2$.

Gradient-based optimization. For the gradient-based methods considered here, the optimization iteration proceeds as

$$u_{k+1} = u_k + \alpha_k d_k \tag{3}$$
$$d_{k+1} = -\nabla j(u_{k+1}) + \beta_k d_k, \tag{4}$$

where k is the iteration number, $\nabla j(u_k)$ the *reduced gradient*, and α_k some step length for the control update. The parameter β_k is used to update the descent direction d_k and determines the type of method (steepest descent for $\beta_k = 0$, various choices for nonlinear conjugate gradient methods).

Computation of the reduced gradient. Adjoint gradient computation consists of three steps:

1. Solve the PDE (1) for a given control u
2. Solve the adjoint PDE

$$-p_t = pp_x - p_{xx} - p_{xxxx} + (y - y_d), \quad x \in \Omega := [0, 32\pi] \tag{5}$$

   with periodic boundary boundary conditions and $p(T, x) = 0$.

3. Set $\nabla j(u) = p(0,x) + R'(u)$.

The source term in eq. 5 comes from differentiating the objective functional 2 with respect to the state y. If, instead of or in addition to, the difference $y - y_d$ in $[0,T]\times\Omega$ we track some desired final state $y_d^T(x) - y(T,x)$ the terminal condition in the adjoint gets modified to $p(T,x) = y_d^T(x) - y(T,x)$ to account for this deviation.

For details on PDE-constrained optimization see, e.g., [1, 4].

2 Implementation

This tutorial example consists of six source files

- hooks.f90 - output residuals during sweeps/PFASST iterations
- level.f90 - interpolation/restriction of solutions
- main.f90 - main program, contains main optimization loop
- pf_optimization_1d.f90 - routines for evaluating objective, gradient, steps size selection
- probin.f90 - define problem parameters
- sweeper.f90 - defines problem equations and optimization quantities

as well as

- Makefile
- imex.**nml** - specify input parameters for test runs
- plot.py - helper file to produce figures from npy output.

In the following we provide a walkthrough through the main parts of the implementation. For all details you will need to go through the actual source code, though.

2.1 Encapsulation, sweeper, and controller for optimal control

The LibPFASST library comes with a special data structure pf_ndarray_oc_t (file pf_ndarray_oc_encap.f90) to store solutions as well as a IMEX sweeper pf_imexQ_oc_t (file pf_imexQ_oc_sweeper.f90) capable of forward- and backward-in-time IMEX-SDC sweeps.

As the optimal control problem requires state and adjoint solution, PFASST's pf_ndarray_t is extended to contain two variables stored in yflatarray and pflatarray. It provides all the functionality of pf_ndarray_t, with the functions taking an optional flag argument specifying on which component to operate on (flag==1 operates on yflatarry, flag==2 on pflatarry, and flag==0 on both).

Similarly, the optimal control sweeper provides the same functionality as the standard IMEX sweeper pf_imex_sweeper_t, again with a flag argument for the function specifying the component and thus forward or backward sweeps (or both).

Finally, for running PFASST routines like pf_predictor_oc, pf_pfasst_block_oc or pf_vcycle_oc are provided in pf_parallel_oc.f90. These routines are similar to the standard PFASST functions in pf_parallel.f90, but adapted to the optimal control setting (e.g., sweeping backwards on the adjoint).

In most cases, the user should not need to edit these library files to solve application problems.

2.2 The optimization loop

Besides setting up PFASST and allocating data structures for various quantities used during the optimization, the main part of solving the optimal control problems happens in the optimization loop in main.f90, see Listing 1.

```
do k=1,max_opt_iter
        if (k .eq. 1) then
                call evaluate_objective(...)
         else
                objective=objectiveNew ! from step size selection/linesearch
        end if

        call mpi_allreduce(objective, globObj, 1, MPI_REAL8, MPI_SUM, pf%comm%comm, ierror)
        if (globObj < tol_obj) exit ! leave loop

        if (k .eq. 1) then ! in later iterations, this is done in linesearch
                call evaluate_gradient(...)
        end if

         ! compute gradient norms
        globL2NormGradSq = sum(gradient**2)*Lx/dble(nvars(pf%nlevels))
        globLinftyNormGrad = maxval(abs(gradient))

        if (sqrt(globL2NormGradSq) < tol_grad) exit

        ! determine new search direction, here: steepest descent
        beta = 0.0_pfdp
        searchDir = -gradient + beta*prevSearchDir

        ! line search for step size selection
        call armijo_step()

        if (stepTooSmall) exit
end do
```

Listing 1: Most important parts of the optimization loop from main.f90

Algorithmic parameters like max_opt_iter or tol_grad are defined in probin.f90 and can be set via command line parameters or an ∗.**nml** file. The main work happens in the functions evaluate_objective and evaluate_gradient from pf_optimization_1d.f90, where state and adjoint equations are solved, the objective functional gets evaluated and the gradient assembled. Only in the first optimization iteration these are called directly; in the following iterations they get called during linesearch (e.g., armijo_step).

Solving the state equation by PFASST is done in evaluate_objective, and is shown in excerpts in Listing 2.

```
subroutine evaluate_objective(...)
        ! ...
        ! set initial condition
        q1%yflatarray = ctrl
        call pf%levels(pf%nlevels)%q0%copy(q1, 1)
        do step = 1, nsteps ! this iterates over all steps this processor computes
                thisstep = (step-1)*pf%comm%nproc + pf%rank
                ! this is the 'true' time step we are computing
                ! i.e., this processors time interval in this step
                ! is [thisstep*dt, (thisstep+1)*dt]

                ! use PFASST to solve on this block of time steps, i.e., every processors computes
                ! their time step in parallel, exchanging data to transmit
                ! initial conditions on the time intervals
                pf%state%pfblock = step
                call pf_pfasst_block_oc(pf, dt, step*pf%comm%nproc, .true., 1, step=thisstep)

                ! evaluate objective functional for all quadrature nodes
                do m = 1, nnodes
                        call objective_function(...)
                end do
                ! integrate objective functional
                objective = ...

                ! copy/broadcase qend to q0 for next step
                if( step < nsteps ) then
                        call pf%levels(pf%nlevels)%qend%pack(...)
                        call pf_broadcast(...)
                        call pf%levels(pf%nlevels)%q0%unpack(...)
                end if
        end do

        ! L2(Omega) control norm
        L2NormCtrlSq = sum(ctrl**2)*Lx/dble(nx)
    objective = 0.5*objective + 0.5*alpha*L2NormCtrlSq
end subroutine evaluate_objective
```

Listing 2: evalute_objective from pf_optimization_1d.f90

The main difference to the other Tutorial examples, where a single PDE is solved only once, is, that we cannot use the usual PFASST controller, but have assign blocks of time steps to processors by hand as well as record the solution values in order to solve the adjoint equation backward in time. There, the processors need to solve the specified time steps for which they have the state equation's solution, thus avoiding costly communication as well as memory issues, as every processor only has to store the values it actually needs.

The evaluate_gradient is similar, and consists of solving the adjoint and constructing the gradient. As the gradient requires the value of the adjoint at $t = 0$, the first processor broadcasts this solution to the other processors, see Listing 3. This broadcast could be left out when it is ensured that only the first processor updates the control (i.e., the initial condition); this modification is left as an exercise for the reader.

```
if(pf%rank == 0) then
        ! gradient is adjoint at t0 (plus regularization term derivative = alpha*ctrl)
        ! pack away on rank 0
        call pf%levels(pf%nlevels)%q0%pack(pf%levels(pf%nlevels)%send, 2)
end if
! broadcast
call pf_broadcast(pf, pf%levels(pf%nlevels)%send, pf%levels(pf%nlevels)%mpibuflen, 0)
! everybody sets gradient (actually this could be skipped, only rank 0 needs initial condition)
gradient = pf%levels(pf%nlevels)%send
gradient = gradient + alpha * ctrl
```

Listing 3: Excerpt of evalute_gradient from pf_optimization_1d.f90

We do not discuss the line search functions armijo_step and strong_wolfe_step here. They are rather straightforward implementations of the standard Armijo- and strong Wolfe conditions (see your favorite textbook on nonlinear optimization, e.g., [3]).

2.3 Specifying the equations

During sweeps, the sweeper calls feval and fcomp routines, which are defined in sweeper.f90. Here the equations are specified. We use an IMEX approach, treating $-y_{xx} - y_{xxxx}$ implicitly.

```
select case (flags) ! are we solving for state or adjoint?
case(1) ! State
        ! Grab the arrays from the encap
        yvec => get_array1d_oc(y,1) ! ( ,1) means get state component
        fvec => get_array1d_oc(f,1)

        ! explicit piece: -y y_x
        tmp = -0.5_pfdp*yvec*yvec
        call fft%conv(tmp, this%ddx, fvec)

case(2) ! Adjoint
        ! Grab the arrays from the encap
        yvec => get_array1d_oc(y,1) ! state sol required for rhs (y-y_desired)
        fvec => get_array1d_oc(f,2)
        pvec => get_array1d_oc(y,2)

        ! explicit piece: +y y_x
        tmp = 0.5_pfdp*pvec*pvec
        call fft%conv(tmp, this%ddx, fvec)

        ! source term from distributed tracking objective
        fvec = fvec + (yvec-this%ydesired(mystep, idx, :))
end select
```

Listing 4: Excerpt of feval from sweeper.f90

Listing 4 shows an excerpt of the feval function, specifying the explicit part of the equation. flags specifies which equation is to be solved, flags==1 is the state equation, flags==2 specifies the adjoint. In the latter, the source term from the distributed tracking type objective enters. The implicit piece is also eval-

uated using FFT, **call** fft%conv(yvec,−this%lap−this%lap∗this%lap,fvec); this does not differ between state and adjoint.

```
if (piece == 2) then
        ! Apply the inverse operator with the FFT convolution
        call fft%conv(rhsvec,1.0_pfdp/(1.0_pfdp − dtq∗(−this%lap∗this%lap−this%lap)),yvec)
        fvec = (yvec − rhsvec) / dtq
end if
```

Listing 5: Excerpt of fcomp from sweeper.f90

The fcomp function in Listing 5 is similar to the other tutorial examples and has no modifications due to the optimization problem.

2.4 Helper and convenience functions

In sweeper.f90 some helper functions specific to the optimization problem can be found. These are, e.g., initialize_ocp to allocate memory for the desired state, or set_ydesired to fill the variable with a desired state coming from measurements or artificially generated data. More importantly, functionality to evaluate the objective functional is provided (objective_function(...)), as well as functions to restrict the desired state (and the computed fine state solution) to coarser levels as required for solving the adjoint.

3 Numerical example

Here we show two numerical results, without giving much detail. Please consider imex.**nml** for input parameters.

3.1 Forward solve

We use the example from [2] and solve the KS equation for $x \in [0, 32\pi]$ on the time interval $[0, 60]$ using the initial condition $y(0, x) = \cos(x/16)(1+\sin(x/16))$. The solution using 600 time steps and two levels with 3/5 Lobatto nodes and 512/1024 spatial degrees of freedom is shown in Fig. 1.

3.2 Identifying the initial condition

Due to the complicated dynamics, identifying the initial condition is demonstrated on the rather short time interval $[0, 1]$, with 100 time steps. The other discretization parameters are as above. We start the optimization with an initial guess $u_0 = 0.75u_{\text{exact}}$. The target state is shown in Fig. 2. Fig. 3 shows the progress of the steepest descent method, while Fig. 4 shows the computed control (i.e., identified initial condition) as well as the difference of the identified and true initial condition.

References

[1] Michael Hinze, Rene Pinnau, Michael Ulbrich, and Stefan Ulbrich. *Optimization with PDE constraints*. Springer, Berlin, 2009.

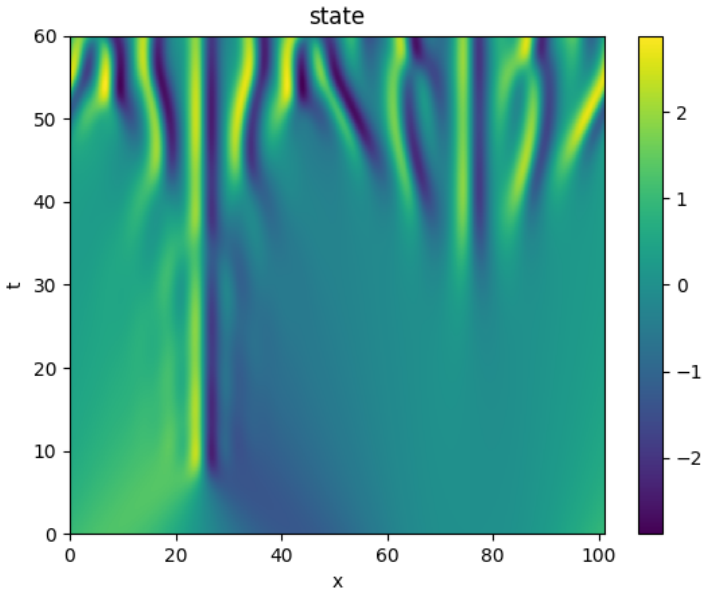

Figure 1: Solution of the KS equation

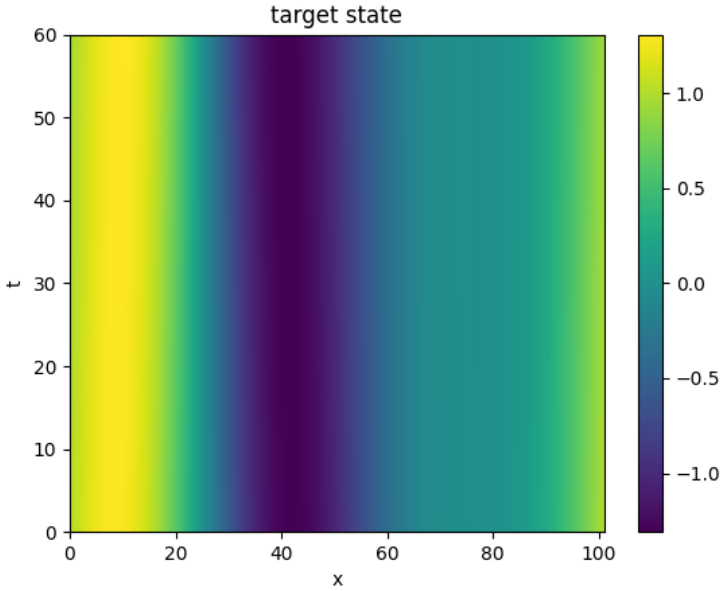

Figure 2: Target state (KS solution with correct initial condition).

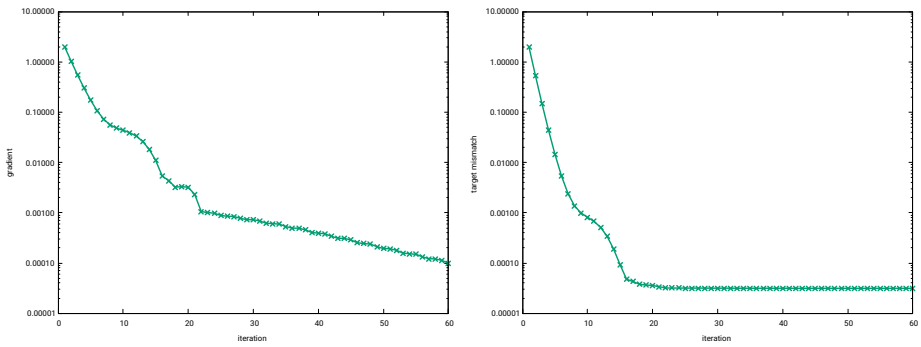

Figure 3: Optimization progress. Left: norm reduced gradient. Right: Target mismatch.

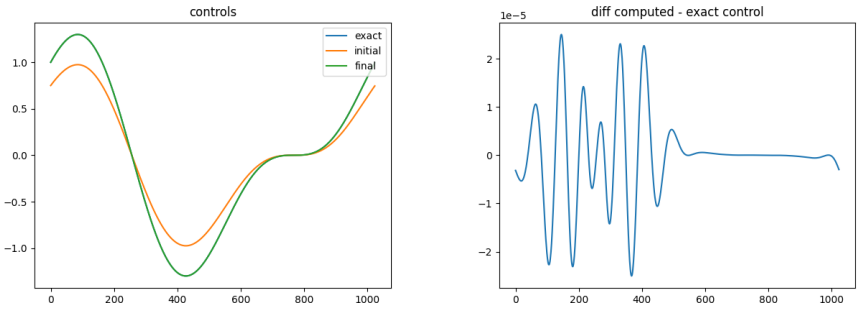

Figure 4: Left: Initial and final control. Right: difference of final to exact control.

[2] Aly-Khan Kassam and Lloyd N Trefethen. Fourth-order time-stepping for stiff pdes. *SIAM J. Sci. Comput.*, 26(4):1214–1233, 2005.

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2006.

[4] Fredi Tröltzsch. *Optimal Control of Partial Differential Equations: Theory, Methods and Applications*. AMS, 2010.