Last updated: December 19, 2020

Numerical Approach

The canonical methods used to solve optimal control problems are the Pontryagin maximum principle and the Hamilton-Jacobi-Bellman equations. Despite the power of these analytic approaches, it remains challenging to apply them to problems with more than a handful of states or multiple types of constraints. Numerical methods to solve optimal control problems include forward-backward sweep method (Lenhart and Workman), and discretize-then-optimize. Here, we use a form of the latter, the **control parameterization** method.

We attempted to solve the optimal control using a forward-backward sweep method (Lenhart and Workman 20XX) but were unable to define the appropriate adjoint equations for the constrained problem. We then applied a form of the 'control parameterization method' described by Lin et al. (2014).

We wrote the function 'control', which computes values of derivatives for state variables in the ODE system, the accumulated penalty for violating constraints, and the accumulated objective function. We solved our ordinary differential equations using the R package **deSolve** (sotaert 2014). Our equations exhibited stiffness so we applied the Adams methods (Sotaert). Although **deSolve**'s default integration method (lsoda) detects stiffness properties, we followed the suggestion in Sotaert et al. and selected the Adams method. We penalized values of the control that violated constraints by raising the absolute value of the difference of control values that lie outside the supported region to the power of 1.25. Originally, we squared this difference but this penalty is shallow for small errors and thus not differentiable. The code snippet below outlines the general structure of the control function.

```
derivs = numeric(3);

control <- function(t,y,parms,f1,f2,f3) {

  ## entries in y (system of ODEs)
  X = y[1];

  ## Control intervals
  u <- f(t);

  ## Apply positivity constraints, penalize if violated
  ut = max(u,0); bad = abs(u-ut)^1.25; u = ut;

  ## Apply upper bound constraint, penalize if violated
  ut = min(u,1); bad = bad + abs(u-ut)^1.25;

  ## Derivative of state variable
```

```
  derivs[1] = u * X;

  ## Cumulative penalty
  derivs[2] = bad;

  ## Cumulative objective function
  derivs[3] = log(X);

  return(list(derivs));
}
```

Lin et al. (2014) provide an overview of the control parameterization method. In this approach, the control is approximated by a "linear combination of basis functions" which are often "piecewise-constant basis functions." In practice, this means that we divide our time horizon into an evenly spaced grid with n points that has $n-1$ intervals. The control function is approximated by $n-1$ control intervals. The value on these intervals is optimized. The grid points remain fixed during optimization. In practice, we divide up our time horizon with grid points and generate a function that performs constant interpolation on the interval between the grid points. The code below demonstrates this procedure in R:

```
## Generate grid points
topt=seq(0,5,length=11);

## Generate random set of control values
par = runif(length(topt),0.01,0.05);

## Generate function for interpolation
f = approxfun(topt,par,rule=2);
```

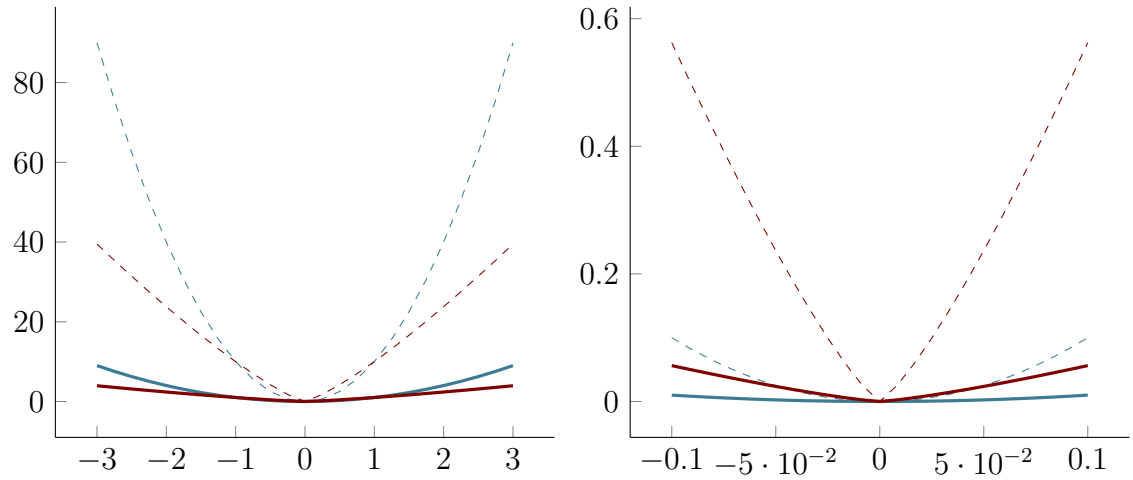
We use the control intervals in our optimization routine. We write a function to be optimized. This function (optimfun) takes values for the control θ as its only argument. The control is used to generate the control intervals. The control intervals are used in combination with the initial conditions and model parameters to solve the differential equation that governs the evolution of the state variables (control). The function containing the differential equation also calculates an integrated constraint violation penalty, which sums across all violations of the state and parameter constraints. The function containing the differential equation also calculates the value of the objective function. The final value maximizes the objective function, and introduces a penalty for constraint violations and solutions with instability (oscillation).

```
optim_fun = function(theta){

  ## Generate function for interpolation
  f = approxfun(topt, theta, rule=2);
```

```
  ## Vectorize initial conditions and parameters
  y0 = c(inits,other)

  ## Solve ODE
  out = ode(y=y0, times=seq(0,5,by=0.1), control, method=odemethod,
      parms=mParms, f=f);

  ## Get final value of constraint penalty
  pen = out[nrow(out),"pen"]; # integrated constraint violation penalty

  ## Get final value of objective function
  obj = out[nrow(out),"obj"];

  ## Calculate penalty for instability
  wiggly = diff(diff(tMat[,1])) + diff(diff(tMat[,2])) + diff(diff(tMat[,3]));

  ## Sum up the value function
  val = obj - pwt*pen - lambda*sum(wiggly^2) ## SPE: sum instead of mean on
      wiggliness
  return(-val)
}
```

We initially considered a quadratic loss function to impose penalties for constraint violations. The quadratic loss function is added on to the objective function for optimization. We weight the penalty to deal with sharp changes in slope of the value function. The quadratic loss function is relatively shallow at small values, so we turned to a modified loss function, $\text{abs}(f(x) - x)^{1.25}$. The function has a greater slope at small values but is smooth. The figure below shows the quadratic and modified loss function over large constraint violations, $(-3, 3)$, and small constraint violations, $(-.1, .1)$. The solid lines are the unweighted loss functions (weight=1; quadratic is blue, modified is red). The dashed lines are the loss functions weighted by a constant of 10.

We started optimization with a smaller penalty for constraint violations (penalty = 1), which is similar to a penalty weight of 10 for a quadratic loss function at small values. After a first round of optimization, we imposed a larger penalty for constraint violation (penalty = 10).



Finally, we initialize our optimization routine. In general, we use a combination of strategies to initialize the optimization routine. For some parameters, we randomly generate values on the support of the control functions (e.g. $0 \leq u(t) \leq 1$). In other cases, we initialize our optimization with values from the analysis of unconstrained versions of our optimal control problem. We then set weights for the penalty we apply for constraint violations and instability. We choose to start with large penalties that decrease with each iteration of optimization; this has the effect of penalizing constraint violations early in the routine when the control is likely to be further from the optimum and reducing the weight of subsequent, smaller violations to the constraints.

```
## Randomly generate initial values for control
par0 = runif(2*length(topt),0.01,0.05);

## Generate initial values for control from analysis of unconstrained problem
# beta2 is maxed-out at the end in the unconstrained problem (from analysis of
    adjoint equations)
# so start with the max first
par0 = c(par0, mParms[2]*seq(0,1,length=length(topt))^2)

## optimize: start with a large lambda, and decrease it with each iteration.
# pwt=1; lambda=0.2; fvals = numeric(5);
# SPE: large penalty weight, and large lambda at first
pwt=10; lambda=1; fvals = numeric(40);
```

Once all of this machinery is in place, we proceed to iteratively solve our control problem. We take the following approach:

- Use the Runge-Kutta-4 method (rk4) for ODEs and Nelder-Mead for optimization (1 iteration).
- Reset the controls to lie within their constraints.

- Use the implicit Adams method (impAdams or impAdamsd) for ODEs and BFGS for optimization (1 iteration).
- Reset the controls to lie within their constraints. - do I do this here as well?
- Enter an optimization loop. Within the loop:

  Use the implicit Adams method (impAdams or impAdamsd) for ODEs and Nelder-Mead for optimization.

  Reset the controls to lie within their constraints.

  Use the implicit Adams method (impAdams or impAdamsd) for ODEs and BFGS for optimization (1 iteration).

  Reduce the weight for lambda by half.

  Reduce the weight for constraint violation by half. (?)

  What's the particular logic of this procedure - need to write a bit about this.

```
fit = optim(par0, fn=optim_fun, method="Nelder-Mead",control =
    list(maxit=5000,trace=4,REPORT=1));
```

References

Lin, Q., R. Loxton, K. Lay Teo, ,Department of Mathematics and Statistics, Curtin University, GPO Box U1987 Perth, Western Australia 6845, and ,Department of Mathematics and Statistics, Curtin University of Technology, GPO Box U 1987, Perth, W.A. 6845. 2014. The control parameterization method for nonlinear optimal control: A survey. Journal of Industrial & Management Optimization, **10**:275–309.