MPC with Casadi/Python

Michael Risbeck

June 10, 2015

What's in mpc-tools-casadi?

A Python package: `mpctools`[1]

- You should put the `mpctools` folder somewhere on your Python path.
- In Python, use `import sys; print sys.path` to see what folders are on your path.

A cheatsheet (in the `doc` folder).

- Should get you started writing your own code.
- Compares plain CasADi vs. CasADi + `mpctools`.

A bunch of example files, e.g.,

- `nmpcexample.py`: Example of linear vs. nonlinear MPC.
- `cstr_startup.py`: startup and a setpoint change (with no disturbances) for the CSTR system from Example 1.11.

[1]See `install.pdf` for detailed installation instructions.

Why did we write this code?

- We plan to solve nonlinear MPC problems.
- CasADi is more robust than our `mpc-tools`
- However, setting up an MPC problem in CasADi takes a fair bit of code
- Everyone copy/pasting their own code is bad.
- A simpler interface means we (and others) can save a lot of time.

From official CasADi Examples

```python
# For all collocation points: eq 10.4 or 10.17 in Biegler's book
# Construct Lagrange polynomials to get the polynomial basis at
# the collocation point
for j in range(deg+1):
    L = 1
    for j2 in range(deg+1):
        if j2 != j:
            L *= (tau-tau_root[j2])/(tau_root[j]-tau_root[j2])

    lfcn = SXFunction([tau],[L])
    lfcn.init()
    # Evaluate the polynomial at the final time to get the
    # coefficients of the continuity equation
    lfcn.setInput(1.0)
    lfcn.evaluate()
    D[j] = lfcn.getOutput()

    # Evaluate the time derivative of the polynomial at all
    #collocation points to get the coefficients of the
    #continuity equation
    tfcn = lfcn.tangent()
    tfcn.init()
    for j2 in range(deg+1):
        tfcn.setInput(tau_root[j2])
        tfcn.evaluate()
        C[j][j2] = tfcn.getOutput()
```

We don't want everyone writing this themselves!

Python Basics

For our purposes Python+Numpy isn't that much different from Octave/MATLAB.

Octave/MATLAB	Python+Numpy
`A = zeros(3,2);`	`A = np.zeros((3,2))`
`x = ones(2,1); %2D.`	`x = np.ones((2,)) #1D.`
`y = A*x;`	`y = A.dot(x)`
`z = y.^2; % Elementwise.`	`z = y**2 # Elementwise.`
`A(1,1) = 2; % One-based.`	`A[0,0] = 2 # Zero-based.`
`A(2,:) = [3,4];`	`A[1,:] = np.array([3,4])`
`s = struct('field',1);`	`s = {"field" : 1}`
`disp(s.field);`	`print s["field"]`

General Tips

- Remember that indexing is 0-based.
- Use NumPy's `array` instead of `matrix`.
  - Despite what Octave/MATLAB says, everything is *not* a matrix of doubles
  - Have to use `A.dot(x)` instead of `A*x`
  - However, indexing is MUCH easier with arrays
- Use `bmat([[A,B],[C,D]]).A` to assemble matrices from blocks.
  - Equivalent to `[A, B; C, D]` in Octave/MATLAB
  - Trailing `.A` casts back to `array` type from `matrix`
- Use `scipy.linalg.solve(A,b)` to compute $A^{-1}b$.

System Model

Start by defining the system model as a Python function.

```python
def ode(x,u,d):
    # Grab the states, controls, and disturbance.
    [c, T, h] = x[:Nx]
    [Tc, F] = u[:Nu]
    [F0] = d[:Nd]

    # Now create the right-hand side function of the ODE.
    rate = k0*c*np.exp(-E/T)

    dxdt = [
        F0*(c0 - c)/(np.pi*r**2*h) - rate,
        F0*(T0 - T)/(np.pi*r**2*h)
        - dH/(rho*Cp)*rate
        + 2*U/(r*rho*Cp)*(Tc - T),
        (F0 - F)/(np.pi*r**2)
    ]

    return np.array(dxdt)
```

System Simulation

The nonlinear system can be simulated using CasADi `integrator` objects with a convenient wrapper.

```
# Turn into casadi function and simulator.
ode_casadi = mpc.getCasadiFunc(ode,
    [Nx,Nu,Nd],["x","u","d"],funcname="ode")
cstr = mpc.DiscreteSimulator(ode, Delta, [Nx,Nu,Nd], ["x","u","d"])

# Simulate with nonlinear model.
x[n+1,:] = cstr.sim(x[n,:] + xs, u[n,:] + us, d[n,:] + ds) - xs
```

Calls to LQR and LQE

The functions `dlqr` and `dlqe` are also provided in `mpc-tools-casadi`.

Octave/MATLAB

```
% Get LQR.
[K, Pi] = dlqr(A, B, Q, R);

% Get Kalman filter.
[L, M, P] = dlqe(Aaug, ...
    eye(naug), Caug, Qw, Rv);
Lx = L(1:n,:);
Ld = L(n+1:end,:);
```

CasADi/Python

```
# Get LQR.
[K, Pi] = mpc.util.dlqr(A,B,Q,R)

# Get Kalman filter.
[L, P] = mpc.util.dlqe(Aaug,
    Caug, Qw, Rv)
Lx = L[:Nx,:]
Ld = L[Nx:,:]
```

Controller Simulation

Octave/Matlab

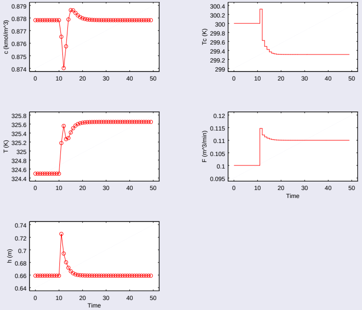
```
for i = 1:ntimes
  % Take plant measurement.
  y(:,i) = C*x(:,i) + v(:,i);

  % Update state estimate with measurement.
  ey = y(:,i) - C*xhatm(:,i) -Cd*dhatm(:,i);
  xhat(:,i) = xhatm(:,i) + Lx*ey;
  dhat(:,i) = dhatm(:,i) + Ld*ey;

  % Steady-state target.
  H  = [1 0 0; 0 0 1];
  G = [eye(n)-A, -B; H*C, zeros(size(H,1), m)];
  qs = G\[Bd*dhat(:,i); ...
      H*(target.yset-Cd*dhat(:,i))];
  xss = qs(1:n);
  uss = qs(n+1:end);

  % Regulator.
  u(:,i) = K*(xhat(:,i) - xss) + uss;
  if (i == ntimes) break; end

  % Simulate with nonlinear model.
  t = [time(i); mean(time(i:i+1)); time(i+1)];
  z0 = x(:,i) + zs;   F0 = p(:,i) + Fs;
  Tc = u(1,i) + Tcs;  F  = u(2,i) + Fs;
  [tout, z] = ode15s(@massenbal, t, z0, opts);
  x(:,i+1) = z(end,:)' - zs;

  % Advance state estimate.
  xhatm(:,i+1) = A*xhat(:,i) ...
      + Bd*dhat(:,i) + B*u(:,i);
  dhatm(:,i+1) = dhat(:,i);
end
```

Python+Numpy

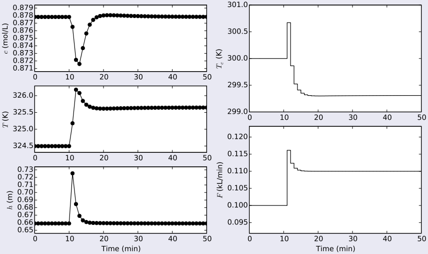
```
for n in range(Nsim + 1):
    # Take plant measurement.
    y[n,:] = C.dot(x[n,:]) + v[n,:]

    # Update state estimate with measurement.
    err[n,:] = (y[n,:] - C.dot(xhatm[n,:])
        - Cd.dot(dhatm[n,:]))
    xhat[n,:] = xhatm[n,:] + Lx.dot(err[n,:])
    dhat[n,:] = dhatm[n,:] + Ld.dot(err[n,:])

    # Make sure we aren't at the last timestep.
    if n == Nsim: break

    # Steady-state target.
    rhs = np.concatenate((Bd.dot(dhat[n,:]),
        H.dot(ysp[n,:] - Cd.dot(dhat[n,:]))))
    qsp = linalg.solve(G,rhs) # i.e. G\rhs.
    xsp = qsp[:Nx]
    usp = qsp[Nx:]

    # Regulator.
    u[n,:] = K.dot(xhat[n,:] - xsp) + usp

    # Simulate with nonlinear model.
    x[n+1,:] = cstr.sim(x[n,:] + xs,
        u[n,:] + us, d[n,:] + ds) - xs

    # Advance state estimate.
    xhatm[n+1,:] = (A.dot(xhat[n,:])
        + Bd.dot(dhat[n,:]) + B.dot(u[n,:]))
    dhatm[n+1,:] = dhat[n,:]
```

Results

Octave/MATLAB



Python+Numpy



What can we do with `mpc-tools-casadi`?

- Discrete-time linear MPC
- Discrete-time nonlinear MPC
  - Explicit models
  - Runge-Kutta discretization
  - Collocation
- Discrete-time nonlinear MHE
  - Explicit models
  - Runge-Kutta discretization
  - Collocation
- Basic plotting function
- Example scripts
  - Linear
  - Solution of linear as nonlinear
  - Periodic linear
  - Example 2-8
  - Simple collocation
  - Example 1-11

Example Script I

Example script for a simple nonlinear MPC problem.

```
# Control of the Van der Pol oscillator.
import mpctools as mpc
import numpy as np

# Define model and get simulator.
Delta = .5
Nsim = 20
Nx = 2
Nu = 1
def ode(x,u):
    dxdt = [(1 - x[1]*x[1])*x[0] - x[1] + u, x[0]]
    return np.array(dxdt)

# Create a simulator.
vdp = mpc.DiscreteSimulator(ode, Delta, [Nx,Nu], ["x","u"])

# Then get nonlinear casadi functions and a linearization.
ode_casadi = mpc.getCasadiFunc(ode, [Nx,Nu], ["x","u"], funcname="f")
lin = mpc.util.getLinearization(ode_casadi,[0,0],[0],Delta=Delta)

# Also discretize using RK4.
ode_rk4_casadi = mpc.getCasadiFunc(ode_rk4, [Nx,Nu], ["x","u"], funcname="F",
                                   rk4=True, Delta=Delta, M=1)
```

Example Script II

```python
# Define stage cost and terminal weight.
def lfunc(x,u): return mpc.mtimes(x.T,x) + mpc.mtimes(u.T,u)
l = mpc.getCasadiFunc(lfunc, [Nx,Nu], ["x","u"], funcname="l")

def Pffunc(x): return 10*mpc.mtimes(x.T,x)
Pf = mpc.getCasadiFunc(Pffunc, [Nx], ["x"], funcname="Pf")

# Create linear discrete-time model for comparison.
def Ffunc(x,u): return (mpc.mtimes(mpc.util.DMatrix(lin["A"]),x) +
    mpc.mtimes(mpc.util.DMatrix(lin["B"]),u))
F = mpc.getCasadiFunc(Ffunc, [Nx,Nu], ["x","u"], funcname="F")

# Make optimizers.
x0 = np.array([0,1])
Nt = 20
commonargs = dict(
    N={"x":Nx, "u":Nu, "t":Nt},
    verbosity=0,
    l=l,
    x0=x0,
    Pf=Pf,
    lb={"u" : -.75*np.ones((Nsim,Nu))},
    ub={"u" : np.ones((Nsim,Nu))},
    runOptimization=False,
)
solvers = {}
```

Example Script III

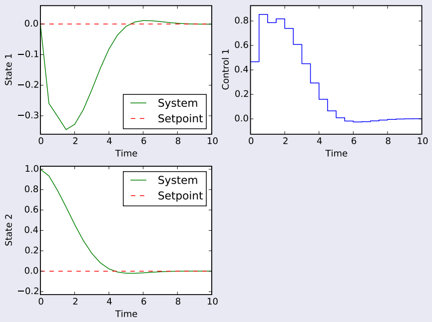
```
solvers["lmpc"] = mpc.nmpc(f=F,**commonargs)
solvers["nmpc"] = mpc.nmpc(f=ode_rk4_casadi,**commonargs)

# Now simulate.
times = Delta*Nsim*np.linspace(0,1,Nsim+1)
x = {}
u = {}
for method in solvers.keys():
    x[method] = np.zeros((Nsim+1,Nx))
    x[method][0,:] = x0
    u[method] = np.zeros((Nsim,Nu))
    for t in range(Nsim):
        solvers[method].fixvar("x",0,x[method][t,:])
        solvers[method].solve()
        print "%5s %d: %s" % (method,t,solvers[method].stats["status"])
        u[method][t,:] = solvers[method].var["u",0,:]
        x[method][t+1,:] = vdp.sim(x[method][t,:],u[method][t,:])
    fig = mpc.plots.mpcplot(x[method],u[method],times,title=method)
    fig.savefig("vdposcillator_%s.pdf" % (method,))
```

Output

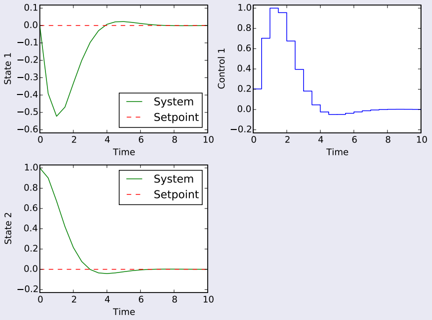
For this problem, nonlinear MPC performs slightly better.

- The computation isn't much more time-consuming because of the power of Casadi.
- The problem isn't difficult to set up because of `mpc-tools-casadi`.

Linear MPC



Nonlinear MPC



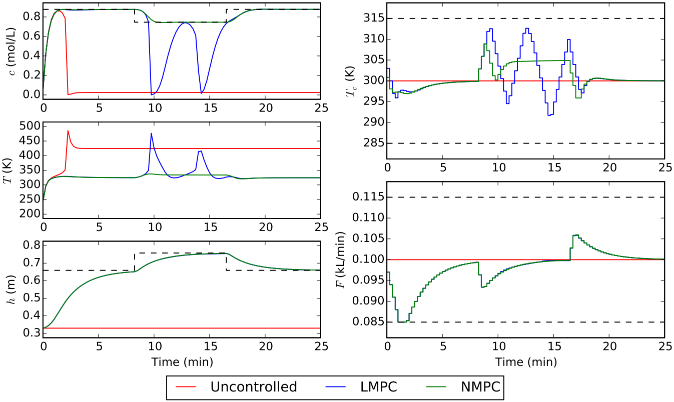
More Complicated Example

Using `mpc-tools-casadi`, we can replace the LQR and KF from Example 1.11 with nonlinear MPC and MHE.

- `cstr_startup.py` shows basic structure and a setpoint change.
- `cstr_nmpc_nmhe.py` shows steady-state target finding and NMHE.
- See the cheatsheet for important functions and syntax.

cstr_startup.py

Here, nonlinear MPC knows to be less aggressive.



What can't we do yet?

- True continuous-time formulation
  - Continuous-time models with explicit time dependence are not supported
  - Quadrature for continuous-time objective function is available via collocation
  - DAE systems are possible in principle
- Quality guess generation
  - Solve sequence of smaller problems
  - Use as initial guess for large problem
  - Must do "by hand"