mpc-tools-casadi Cheat Sheet

1 Functions Reference

Here we present some of the most useful functions from `mpc-tools-casadi`. These descriptions are not intended to be complete, and you should consult the documentation within the Python module for more details.

Obtaining `mpc-tools-casadi`. The latest files can be found on <https://hg.cae.wisc.edu/hg/mpc-tools-casadi>. You will see a link on the left to download all of the files in a compressed archive. No specific installation is required beyond Python and CasADi.

Getting Started. Functions are arranged in a package called `mpctools`. Typically, everything you need can be found in the main level, e.g.,

```
import mpctools as mpc
```

To access older versions of functions, you can use

```
import mpctools.legacy as mpc_old
```

The legacy functions are used in some of the example files, but they are being phased out because they are slower and less general.

Many functions have optional arguments or default values that aren't listed below. Consult the docstrings throughout `mpc-tools-casadi` to see what options are available.

Simulating Nonlinear Systems. To facilitate nonlinear simulations, we provide the `OneStepSimulator` class, which is a wrapper a CasADi `Integrator` object. To initialize, the syntax is

```
model = OneStepSimulator(ode,Delta,Nx,Nu)
```

where `ode` is a Python function that takes two keyword arguments `x` and `u` of lengths `Nx` and `Nu`. Optional arguments `Nd` or `Nw` can be set to positive integers to add additional arguments.

Once the object has been built, one timestep can be simulated using

```
xnext = model.sim(x,u)
```

Building CasADi Functions. To simplify creation of CasADi functions, there are a few convenience wrappers.

`getCasadiFunc(f,Nx,Nu,Nd)`

Takes a Python function and sizes of x, u, and/or d to build a CasADi `MXFunction` object. Note that the original function `f` should return a list of values (or at least something that can be passed as an argument to CasADi's `vertcat`).

`getCasadiFuncGeneralArgs(f,varsizes)`

For functions with arguments other than x, u, or d, use this version instead. `varsizes` is a list of integers giving the sizes of the inputs to `f` in positional order.

`getCasadiIntegrator(f,Delta,Nx,Nu,Nd)`

Returns an `Integrator` object to integrate the Python function `f` from time 0 to `Delta`.

`getRungeKutta4(f,Delta,M=1)`

Returns an explicit Runge-Kutta 4th order discretization with `M` steps of size `Delta`/`M`. Note that for this function, `f` must already be an `MXFunction`, i.e., you should use it *after* calling `getCasadiFunc`.

Solving MPC Problems. For discrete-time regulation problems, the function `nmpc` should be used.

`nmpc(F,l,x0,N)`

`F` and `l` should be Python *lists* of CasADi discrete-time functions to describe state evolution and stage costs. These lists are accessed modulo length with respect to time, so T-periodic systems should have T entries, time-invariant systems should have only one entry, etc. `x0` is the starting state, and `N` is the number of stages to consider in the horizon. Additional optional arguments are given below.

- `Pf`: a single CasADi function of x to use as a terminal cost.
- `bounds`: A dictionary with entries `"uub"`, `"ulb"`, `"xub"`, and/or `"xlb"` to define box constraints on u or x. Each entry should be a list of vectors that define the bounds throughout the time horizon. These lists are accessed modulo length, so they need only have one element if the bounds are time-invariant.
- `verbosity`: an integer to control how detailed the solver output is. Lower numbers give less output.

Returns a dictionary of optimal variables and other information. Entries include `"x"` and `"u"` with optimal trajectories for x and u. These are both arrays with each column corresponding to values at different time points. Also given are `"obj"` with the optimal objective function value and `"status"` as reported by the optimizer.

For continuous-time problems, there are a few options. To use Runge-Kutta methods, you can either convert your function ahead of time with `getRungeKutta4`, or you can specify optional arguments `timemodel="rk4"` and `Delta=dt` to have the continuous-time functions converted automatically. To use collocation, you can use `timemodel="colloc"`. This also requires specifying the sample time. Note that for both options, an argument `M` controls how many interpolation points are used on each time interval.

Currently, there is no support for a continuous-time objective function (i.e., continuous-time integral of a cost function). We plan to add support for this feature in the future, but in principle you could augment your model with an integrator state to calculate the objective function.

Time-Invariant Problems. If your system is time-invariant and you plan to be solving the problem repeatedly, speed can be improved by using the `TimeInvariantSolver` class.

The easiest way to build one of these objects is by setting the optional argument `returnTimeInvariantSolver` to `True` in `nmpc`. Below we list the useful methods for this class.

`fixvar(var,t,val)`

Fixes the variable named `var` to take on the value `val` at time `t`. This is most useful for changing the initial conditions, e.g., with

```
solver.fixvar("x",0,x0)
```

which allows for easy re-optimization.

`solve()`

Returns a dictionary similar to the output of `nmpc`. Note, however, that the entries `"x"` and `"u"` have time running along the *rows* of the array and *not* the columns.

`saveguess(guess)`

Takes a solution dictionary and stores the values as a guess to the optimizer. By default, time values are offset by 1. This is done so that

```
sol = solver.solve()
if sol["status"] == "Solve_Succeeded":
    solver.saveguess(sol)
    solver.fixvar("x",0,sol["x"][1,:]
```

prepares the solver for re-optimization at the next time point by using the final $N-1$ values of the previous trajectory as a guess for the first $N-1$ time periods in the next optimization.

State Estimation. For nonlinear state estimation, we provide a moving-horizon estimation function and an Extended Kalman Filter function.

`nmhe(f,h,u,y,l,N)`

Solves a nonlinear MHE problem. As with `nmpc`, arguments `f`, `h`, and `l` should be Python *lists* of CasADi functions. `f` must be $f(x,u,w)$, `h` must be $h(x)$, and `l` must be $\ell(w,v)$. `u` and `y` must be arrays of past control inputs and measurements. These arrays must have time running along rows so that `y[t,:]` gives the value of y at time t.

Different from `nmpc`, the input `N` must be a dictionary of sizes. This must have entries `"t"`, `"x"`, `"u"`, and `"y"`. It may also have a `"w"` entry, but this is set equal to `N["x"]` if not supplied. Note that for feasibility reasons, `N["v"]` is always set to `N["y"]` regardless of user input. Additional optional arguments are given below.

- `lx`, `x0bar`: arrival cost for initial state. `lx` should be a CasADi function of only x. It is included in the objective function as $\ell_x(x_0-\overline{x}_0)$, i.e., penalizing the difference between the value of the variable x_0 and the prior mean $\overline{x}_0$.
- `lb`, `ub`, `guess`: Dictionaries to hold bounds and a guess for the decision variables. Each argument should only have entries for the variables with explicit bounds or for which you have a guess, and within each argument, each dictionary entry should be a 2D array with time running along the rows.
- `verbosity`: same as in `nmpc`.

Note that because these optimization problems inherently change data and horizon lengths, there is no equivalent to `TimeInvariantSolver` for the NMPC problems. This functionality may be added later, but for now all problems have to be built from scratch using the `nmhe` function.

`ekf(f,h,x,u,w,y,P,Q,R)`

Advances one step using the Extended Kalman Filter. `f` and `h` must be CasADi functions. `x`, `u`, `w`, and `y` should be the state estimate $\hat{x}(k|k-1)$, the controller move, the state noise (only its shape is important), and the current measurement. `P` should be the prior covariance $P(k|k-1)$. `Q` and `R` should be the covariances for the state noise and measurement noise. Returns a list of

$$[P(k+1|k),\ \hat{x}(k+1|k),\ P(k|k),\ \hat{x}(k|k)].$$

Plotting. For quick plotting, we have the `mpcplot` function. Required arguments are `x` and `u`, both 2D arrays with each column giving the value of x or u at a given time point, and a vector `t` of time points. Note that `t` should have as many entries as `x` has columns, while `u` should have one fewer column.

Functions from Octave/MATLAB. For convenience, we have included a few simple control-related functions from Octave/MATLAB.

`dlqr(A,B,Q,R)`, `dlqe(A,C,Q,R)`

Discrete-time linear-quadratic regulator and estimator. Note that cross-penalties are not supported.

`c2d(A,B,Delta)`

Converts continuous-time model (A,B) to discrete time with sample time `Delta`.

2 Common Mistakes

Below we list some common issues that may cause headaches.

- NumPy arrays versus matrices.

  As the `matrix` data type plays second fiddle in NumPy, all of the functions have been written expecting arrays and it is suggested that you do the same. Any matrix multiplications within `mpc_tools_casadi.py` are written as `A.dot(b)` instead of `A*b` as would be common in Octave/MATLAB.

For quadratic stage costs, we provide `mtimes` (itself, just a wrapper of CasADi's `mul`), which multiplies an arbitrary number of arguments. Unfortunately this isn't compatible with `arrays`, and so you will want to cast to CasADi's `DMatrix` type before multiplying.

If you encounter errors such as "`cannot cast shape (n,1) to shape (n,)`" or something of that nature, be careful about whether you are working with 1D `arrays`, vectors stored as `matrix` objects, etc. This may mean adding `np.newaxis` to your assignment statements or using constructs like `np.array(x).flatten()` to force your data to have the right shape.

- Dimensions for data arrays.

  During initial development, MATLAB's "everything is a matrix of doubles" philosophy was still embraced. This meant that when storing time-series data for a variable x, it made sense to think of each time point as a column vector, and thus data is structured as `x[i,t]` with i the components of x and t the time series. This is how things are arranged in `nmpc` and `mpcplot`.

  However, as development proceeded, we realized that there is no reason why each x has to be a vector. For instance, you may want to arrange your states as a 3D array. This meant that `x[i_1,t,i_2,i_3]` would get confusing, and so the logical choice would be to put time along the first dimension and let an arbitrary number of dimensions follow. This is the convention used in `nmpc` and `TimeInvariantSolver`. Eventually, we plan to rewrite everything to use this "time first" paradigm, but for now we are stuck with this fragmentation, and care is needed.

- Not passing lists of functions to `nmpc` or `nmhe`.

  These arguments are lists to support systems where the evolution equations may change drastically in time. For time-invariant systems, there is only one such function, and so you may forget to enclose it in a list. Any errors that say something "cannot be indexed" are likely due to this omission.

  A related issue is that CasADi `MXFunctions` always return lists, and so you will need to index the returned value to get what you want, e.g.,

```
z = f([x,y])[0]
```

- Not using iterables in user-defined functions.

  Whenever you pass a Python function to `getCasadiFunction`, CasADi's `vertcat` function is called on its output. This means that

```
def myfunc(x):
    return [x[0]**2, 1/x[1]]
```

  would give the expected results, i.e., a CasADi function that returns a vector of 2 elements. However, if your functions are defined using matrix operations, e.g.,

```
def myfunc(x):          # This is a bad definition.
    return mtimes(A,x) # Output can't be vertcated.
```

  then this causes an error because you cannot call `vertcat` on a single vector not enclosed in a list. The simple resolution is to make the output be a one-element list, i.e.,

```
def myfunc(x)
    return [mtimes(A,x)]
```

- Poor initial guesses to solvers.

  By default, all variables are given guesses of 0. For models in deviation variables, this makes sense, but for general models, these values can cause problems, e.g., if there are divisions or logarithms any where. Make sure you supply an initial guess if the optimal variables are expected to be nowhere near 0, and it helps if the guess is consistent with lower and upper bounds. For difficult problems, it may help to solve a series of small problems to get a feasible starting guess for the large overall problem.

- Tight state constraints.

  Although the solvers allow constraints on all decision variables, tight constraints on the state variables (e.g., that the system terminate at the origin) can be troublesome for the solver. Consider using a penalty function first to get a decent guess and then re-solving with hard constraints from there.

3 Example File

Below, we present an example file to show how much code is saved by using `mpc-tools-casadi`. On the left side, we show the the script written using the pure `casadi` module, while on the right, we show the script rewritten to use `mpc-tools-casadi`.

```
# Control of the Van der Pol
# oscillator using pure casadi.
import casadi
import casadi.tools as ctools
import numpy as np
import matplotlib.pyplot as plt

# Define model and get simulator.
Delta = .5
Nt = 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)

# Define symbolic variables.
x = casadi.SX.sym("x",Nx)
u = casadi.SX.sym("u",Nu)

# Make integrator object.
ode_integrator = casadi.SXFunction(
    casadi.daeIn(x=x,p=u),
    casadi.daeOut(ode=casadi.vertcat(ode(x,u))))
vdp = casadi.Integrator("cvodes",ode_integrator)
vdp.setOption("abstol",1e-8)
vdp.setOption("reltol",1e-8)
vdp.setOption("tf",Delta)
vdp.init()

# Then get nonlinear casadi functions
# and rk4 discretization.
ode_casadi = casadi.SXFunction(
    [x,u],[casadi.vertcat(ode(x,u))])
ode_casadi.init()

[k1] = ode_casadi([x,u])
[k2] = ode_casadi([x + Delta/2*k1,u])
[k3] = ode_casadi([x + Delta/2*k2,u])
[k4] = ode_casadi([x + Delta*k3,u])
xrk4 = x + Delta/6*(k1 + 2*k2 + 2*k3 + k4)
ode_rk4_casadi = casadi.SXFunction([x,u],[xrk4])
ode_rk4_casadi.init()

# Define stage cost and terminal weight.
lfunc = (casadi.mul([x.T,x])
    + casadi.mul([u.T,u]))
l = casadi.SXFunction([x,u],[lfunc])
l.init()

Pffunc = casadi.mul([x.T,x])
Pf = casadi.SXFunction([x],[Pffunc])
Pf.init()
```

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

# Define model and get simulator.
Delta = .5
Nt = 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.OneStepSimulator(ode,
    Delta, Nx, Nu, vector=True)

# Then get nonlinear casadi functions
# and rk4 discretization.
def ode_rk4(x,u):
    return mpc.util.rk4(ode, x, [u], Delta=Delta)
ode_rk4_casadi = mpc.getCasadiFunc(ode_rk4,
    [Nx,Nu], ["x","u"], funcname="F")

# 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")
```

```
# Bounds on u.
uub = 1
ulb = -.75

# Make optimizers.
x0 = np.array([0,1])

# Create variables struct.
var = ctools.struct_symSX([(
    ctools.entry("x",shape=(Nx,),repeat=Nt+1),
    ctools.entry("u",shape=(Nu,),repeat=Nt),
)])
varlb = var(-np.inf)
varub = var(np.inf)
varguess = var(0)

# Adjust the relevant constraints.
for t in range(Nt):
    varlb["u",t,:] = ulb
    varub["u",t,:] = uub

# Now build up constraints and objective.
obj = casadi.SX(0)
con = []
for t in range(Nt):
    con.append(ode_rk4_casadi([var["x",t],
        var["u",t]])[0] - var["x",t+1])
    obj += l([var["x",t],var["u",t]])[0]
obj += Pf([var["x",Nt]])[0]

# Build solver object.
con = casadi.vertcat(con)
conlb = np.zeros((Nx*Nt,))
conub = np.zeros((Nx*Nt,))

nlp = casadi.SXFunction(casadi.nlpIn(x=var),
    casadi.nlpOut(f=obj,g=con))
solver = casadi.NlpSolver("ipopt",nlp)
solver.setOption("print_level",0)
solver.setOption("print_time",False)
solver.setOption("max_cpu_time",60)
solver.init()

solver.setInput(conlb,"lbg")
solver.setInput(conub,"ubg")

# Now simulate.
Nsim = 20
times = Delta*Nsim*np.linspace(0,1,Nsim+1)
x = np.zeros((Nsim+1,Nx))
x[0,:] = x0
u = np.zeros((Nsim,Nu))
for t in range(Nsim):
    # Fix initial state.
    varlb["x",0,:] = x[t,:]
    varub["x",0,:] = x[t,:]
    varguess["x",0,:] = x[t,:]
```

```
# Bounds on u.
lb = {"u" : -.75*np.ones((Nt,Nu))}
ub = {"u" : np.ones((Nt,Nu))}
bounds = dict(uub=[1],ulb=[-.75])

# Make optimizers.
x0 = np.array([0,1])
N = {"x":Nx, "u":Nu, "t":Nt}
solver = mpc.nmpc(f=ode_rk4_casadi,N=N,
    verbosity=0,l=l,x0=x0,Pf=Pf,
    lb=lb,ub=ub,runOptimization=False)

# Now simulate.
Nsim = 20
times = Delta*Nsim*np.linspace(0,1,Nsim+1)
x = np.zeros((Nsim+1,Nx))
x[0,:] = x0
u = np.zeros((Nsim,Nu))
for t in range(Nsim):
    # Fix initial state.
    solver.fixvar("x",0,x[t,:])
```

```
    solver.setInput(varguess,"x0")
    solver.setInput(varlb,"lbx")
    solver.setInput(varub,"ubx")

    # Solve nlp.
    solver.evaluate()
    status = solver.getStat("return_status")
    optvar = var(solver.getOutput("x"))

    # Print stats.
    print "%d: %s" % (t,status)
    u[t,:] = optvar["u",0,:]

    # Simulate.
    vdp.setInput(x[t,:],"x0")
    vdp.setInput(u[t,:],"p")
    vdp.evaluate()
    x[t+1,:] = np.array(
        vdp.getOutput("xf")).flatten()
    vdp.reset()
```

```
    # Solve nlp.
    solver.solve()

    # Print stats.
    print "%d: %s" % (t,solver.stats["status"])
    u[t,:] = solver.var["u",0,:]

    # Simulate.
    x[t+1,:] = vdp.sim(x[t,:],u[t,:])
```

```
# Plots.
fig = plt.figure()
numrows = max(Nx,Nu)
numcols = 2

# u plots. Need to repeat last element
# for stairstep plot.
u = np.concatenate((u,u[-1:,:]))
for i in range(Nu):
    ax = fig.add_subplot(numrows,
        numcols,numcols*(i+1))
    ax.step(times,u[:,i],"-k")
    ax.set_xlabel("Time")
    ax.set_ylabel("Control %d" % (i + 1))

# x plots.
for i in range(Nx):
    ax = fig.add_subplot(numrows,
        numcols,numcols*(i+1) - 1)
    ax.plot(times,x[:,i],"-k",label="System")
    ax.set_xlabel("Time")
    ax.set_ylabel("State %d" % (i + 1))

fig.tight_layout(pad=.5)
```

```
# Plots.
mpc.plots.mpcplot(x.T,u.T,times)
```

Even for this simple example, `mpc-tools-casadi` can save a significant amount of coding, and it makes script files much shorter and more readable while still taking advantage of the computational power provided by CasADi.

4 Disclaimer

Note that since CasADi is in active development, `mpc-tools-casadi` will need to be updated to reflect changes in CasADi's Python API. Additionally, function internals may change significantly as we identify better or more useful ways to wrap the relevant CasADi functions. This means function call syntax may change, although we will strive to maintain compatibility wherever possible.

As mentioned previoiusly, the latest files can always be found on <https://hg.cae.wisc.edu/hg/mpc-tools-casadi>. For questions, comments, or bug reports, please contact us by email.

Michael J. Risbeck <risbeck@wisc.edu>
Nishith R. Patel <nrpatel@wisc.edu>
James B. Rawlings <james.rawlings@wisc.edu>
University of Wisconsin–Madison