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, but note that CasADi must be at least Version 2.4.

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
```

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 `DiscreteSimulator` class, which is a wrapper a CasADi `Integrator` object. To initialize, the syntax is

```
model = DiscreteSimulator(ode,Delta,argsizes)
```

where `ode` is a Python function that takes a fixed number of arguments whose lengths are given (in order) in the list `argsizes`.

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

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

Note that the number of arguments will vary based on how many entries you supplied in argsizes.

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

`getCasadiFunc(f,argsizes,argnames)`

Takes a Python function and sizes of arguments to build a CasADi `SXFunction` object. Note that the original function `f` should return a single numpy vector (e.g., by calling `np.array` before returning). The input `argnames` is optional, but it should be a list of strings that give variable names. This helps make things self-documenting.

Optional arguments are available to return a Runge-Kutta discretization. For this, you must specify `rk4=True` and also provide arguments `Delta` with the timestep and `M` with the number of steps to take in each interval. Example usage is shown below.

```
import mpctools as mpc

# 2 states and 1 control.
def ode(x,u):
    dxdt = [x[0]**2 + u[0], x[1] - u[0]]
    return np.array(dxdt)

ode = mpc.getCasadiFunc(ode, [2,1], ["x","u"])

Delta = 0.5 # Set timestep.
ode_rk4 = mpc.getCasadiFunc(ode, [2,1], ["x","u"],
    rk4=True, Delta=Delta, M=1)
```

`getCasadiIntegrator(f,Delta,argsizes,argnames)`

Returns an `Integrator` object to integrate the Python function `f` from time 0 to `Delta`. `argsizes` and `argnames` are the same as in `getCasadiFunc`, but the differential variables (i.e., x in $dx/dt = f(x, y, z)$) must come first.

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

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

`f` and `l` should be individual CasADi functions to describe state evolution and stage costs. `N` is a dictionary that holds all of the relevant sizes. It must have entries `"x"`, `"u"`, and `"t"`, all of which are integers. `x0` is the starting state. Additional optional arguments are given below.

- `Pf`: a single CasADi function of x to use as a terminal cost.
- `lb`, `ub`, `guess`: Dictionaries with entries `"x"` and/or `"u"`, to define box constraints or an initial guess for the optimal values of x and u. Entries for x should be a numpy array of size `N["t"]+1` by `N["x"]`, and for u, entries should be `N["t"]` by `N["u"]`. Note that the time dimensions can be omitted if the bounds are not time-varying.
- `uprev`: Value of the previous control input. If provided, variables Δu will be added to the control problem. Bounds for Δu can be specified as `"Du"` entries in `lb` and `ub`.
- `largs`: List of strings specifying the arguments of `l` in order. `"Du"` can be included in this list if you wish to use rate-of-change penalties for u.
- `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 convert your function ahead of time (e.g., with `util.rk4` as above). To use collocation, you can add an entry `"c"` to the argument `N` to specify the number of collocation points on each time interval. This also requires specifying the sample time `Delta`. Note that if you want a continuous-time objective function (i.e., integral of $\ell(x(t), u(t))$ instead of a sum), then you can specify `discretel=False` as an argument. Note that this is only supported with collocation.

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.

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 individual 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"`. Note that `N["t"]` gives the number of time *intervals*, which means `u` should have `N["t"]` data points, while `y` should have `N["t"] + 1` data points. 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. Same as in `nmpc`.
- `verbosity`: same as in `nmpc`.

`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

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

Steady-State Targets. For steady-state target selection, we provide a function `sstarg` as described below.

`sstarg(f,h,N)`

Solves a nonlinear steady-state target problem. `f` must be $f(x, u)$ and `h` must be $h(x)$ As with the other functions, the input `N` must be a dictionary of sizes. This must have entries `"x"`, `"u"`, and `"y"`. Additional arguments are below.

- `phi`, `phiargs`: Objective function for if the solution is non-unique. `phi` must be a CasADi function with the arguments as given in `phiargs`.
- `lb`, `ub`, `guess`: Dictionaries to hold bounds and a guess for the decision variables. Each entry must be a 1 by n array, i.e., with a dummy "time" dimension first to match `nmpc` and `nmhe`. Note that if you want to force outputs y to a specific value, you should set equal lower and upper bounds for those entries.
- `verbosity`: same as in `nmpc`.

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 `ControlSolver` class.

The easiest way to build one of these objects is by setting the optional argument `runOptimization` to `False` in `nmpc`, `nmhe`, or `sstarg`. This returns a `ControlSolver` object instead of immediately optimizing and returning the solution. 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. You can also specify a fourth argument `inds`, if you only want to set a subset of indices for that variable (e.g., `solver.fixvar("y",0,ysp[contVars],contVars)` to only fix the values of y for controlled variables).

`solve()`

Solves the optimization problem. Some stats (including solver success or failure) is stored into the `solver.stats` dictionary, and the optimal values of the variables are in the `solver.var` struct (e.g., `solver.var["x",t]` gives the optimal value of x at time t).

`saveguess()`

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

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

Plotting. For quick plotting, we have the `mpcplot` function. Required arguments are `x` and `u`, both 2D arrays with each row 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 rows, while `u` should have one fewer rows.

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

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

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

`util.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.

- Forgetting CasADi functions return lists.

  CasADi `SXFunctions` (e.g., the output of `getCasadiFunc`) 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]
```

- 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
```

```
# 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
```

```python
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(
    "ode",
    casadi.daeIn(x=x,p=u),
    casadi.daeOut(ode=ode(x,u)))
intoptions = {
    "abstol" : 1e-8,
    "reltol" : 1e-8,
    "tf" : Delta,
}
vdp = casadi.Integrator("int_ode",
    "cvodes", ode_integrator, intoptions)

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

[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(
    "ode_rk4", [x,u], [xrk4])

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

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

# 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)
```

```python
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 casadi function for rk4
# discretization.
ode_rk4_casadi = mpc.getCasadiFunc(ode,
    [Nx,Nu], ["x","u"], funcname="F",
    rk4=True, Delta=Delta, M=1)

# 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.
lb = {"u" : -.75*np.ones((Nu,))}
ub = {"u" : np.ones((Nu,))}

# 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)
```

```
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(
    "nlp",
    casadi.nlpIn(x=var),
    casadi.nlpOut(f=obj,g=con))
nlpoptions = {
    "print_level" : 0,
    "print_time" : False,
    "max_cpu_time" : 60,
}
solver = casadi.NlpSolver("solver",
    "ipopt", nlp, nlpoptions)

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,:]
    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)
```

```
# 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,:])

    # Solve nlp.
    solver.solve()

    # Print stats.
    print "%d: %s" % (t,solver.stats["status"])
```

```python
    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()

# 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)
import mpctools.plots # Need to grab one function to show plot.
mpctools.plots.showandsave(fig,"comparison_casadi.pdf")
```

```python
    u[t,:] = solver.var["u",0,:]

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

# Plots.
fig = mpc.plots.mpcplot(x,u,times)
mpc.plots.showandsave(fig,"comparison_mtc.pdf")
```

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