MPCTools Octave/Matlab Cheat Sheet

1 Functions Reference

Here we present some of the most useful functions from MPCTools. These descriptions are not intended to be complete, and you should consult the complete documentation (`documentation.pdf`) for more details.

Obtaining MPCTools. The latest files can be found on <https://bitbucket.org/rawlings-group/octave-mpctools>. You will see a link on the left to download all of the files in a compressed archive. No specific installation is required beyond Octave/Matlab and CasADi, but note that CasADi must be at least Version 3.1.

Getting Started. Functions are stored in a package called `mpctools` using Octave/Matlab's `+package` system. To access these functions, you will need to put the folder `+mpctools` and the file `import_mpctools.m` *inside* a folder that is on your Octave/Matlab path. For example, if `/home/me/octave` is on your Octave/Matlab path, you should put `+mpctools` and `import_mpctools.m` inside `/home/me/octave`. You *cannot* add the `+mpctools` folder directly to your path.

To use the functions within your code, we provide the `import_mpctools` function that returns a "module" struct that contains handles to all of the functions in the public API. For example, your script should start with

```
mpc = import_mpctools();
```

which will allow you to access each of the function names with the syntax `mpc.getCasadiFunc`, `mpc.nmpc`, etc. If instead you want direct access to the functions, you can use

```
import_mpctools('*');
```

which will instead place all the function handles in the current workspace.

Function Usage All of the functions returned by `import_mpctools()` allow arguments to be specified positionally, using keywords via `'keyword', value` pairs, or as a structure whose fields are the keywords and values are values. Consult `documentation.pdf` for more details.

Many functions have optional arguments or default values that aren't listed below. Consult the complete documentation in `documentation` to see what additional keyword options are available.

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

`getCasadiFunc(f,argsizes,argnames)`

Takes a function handle and sizes of arguments to build a CasADi `Function` object. Note that the original function `f` should return a single (column) vector. The input `argnames` is optional, but it should be a cell array 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.

```
mpc = import_mpctools();

% 2 states and 1 control.
function dxdt = ode(x,u)
    dxdt = [x(1)^2 + u(1); x(2) - u(1)];
end

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. These objects are useful to simulate plants with nonlinear ODEs. They can also be used as the plant "model" for particularly challenging ODEs, although they can significantly slow the optimization.

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 struct 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.

- `Vf`: a single CasADi function of x to use as a terminal cost.
- `lb`, `ub`, `guess`: Structs 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 an 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`.
- `verbosity`: an integer to control how detailed the solver output is. Lower numbers give less output. Default is 0 (no output).

This function returns a `ControlSolver` object (see "Repeated Optimization" below for more details).

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 `'rk4', true` 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.

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 columns so that `y(:,t)` gives the value of y at time t.

As in `nmpc`, the input `N` must be a struct 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 $\ell_x(x_0, \overline{x}_0)$.
- `lb`, `ub`, `guess`: Structs to hold bounds and a guess for the decision variables. Same as in `nmpc`.
- `verbosity`: same as in `nmpc`.

The return value is the 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 values

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

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 struct of sizes. This must have entries `x`, `u`, and `y`. Additional arguments are below.

- `phi`, `funcargs`: Objective function for if the solution is non-unique. `phi` must be a CasADi function to define the objective to use in the case of non-unique steady states.
- `lb`, `ub`, `guess`: Structs to hold bounds and a guess for the decision variables. Each entry must be a column vector. Note that if you want to force outputs y to a specific value, you should set equal lower and upper bounds for those entries (e.g., via `ControlSolver.fixvar('y', 1, ysp)`).
- `verbosity`: same as in `nmpc`.

Repeated Optimization. If you plan to be solving the same optimization repeatedly, speed can be improved by using the `ControlSolver` class. These objects are the returned value of `nmpc`, `nmhe`, or `sstarg`. 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., `contvars = [1; 3]; solver.fixvar('y',1,ysp(contvars),contvars)` to only fix the first and third values of y).

`solve()`

Solves the optimization problem. Some stats (including solver success or failure) is stored into the `solver.stats` dictionary, the status string is stored in `solver.status` (although it may or may not be correctly defined for solvers other than IPOPT), 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 isequal(solver.status, 'Solve_Succeeded')
    solver.saveguess();
    solver.fixvar('x',0,solver.var.x(:,1));
end
```

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 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 columns.

Linear MPC Functions. There are no specific functions to handle linear problems. However, you can specify the option `'isQP', true` in each of `nmpc`, `nmhe`, and `sstarg` to let the solver know that the constraints are linear and the objective function quadratic, which can potentially speed up solution.

To linearize nonlinear systems, we provide a useful function.

`getLinearizedModel(f, args, names)`

Evaluates the derivatives of the CasADi function `f` at the point indicated in `args` (which should be a cell array of column vectors) and returns a dictionary. `names` should be a cell array of keys to use in the returned dictionary. Optionally, you can specify a `Delta` keyword argument to discretize the returned matrices.

Choosing the Solver. By default, all optimization problems are solved using the NLP solver IPOPT. To choose a different solver, specify a `'solver', solver` option to `nmpc`, `nmhe`, or `sstarg`. By default, CasADiis bundled with the solvers IPOPT, qpOASES (for quadratic programs), and BONMIN (for mixed-integer problems). For each of these, `solver` should be a lower-case string, e.g., `'ipopt'`, `'qpoases'`, and `'bonmin'`. If installed on your machine, you may also use Gurobi (string `'gurobi'`). Note however that CasADi does not (yet) provide full access to solver options for any solver besides IPOPT, and thus certain options (most notably `'verbosity'` and `'timelimit'`) may not be respected if `solver` is not `'ipopt'`.

Function Arguments Generally, the functions used in MPC-style control problems have standard arguments, for example $f(x, u)$, $\ell(x, u)$ for control, or $f(x, u, w)$, $\ell(w, v)$ for estimation. However, if your instance does not follow these default patterns, you can adjust the number and order of arguments that each function takes by providing the argument names to `getCasadiFunc()`. For example, in `nmpc()`, if your stage cost function is only a function of x and not u, then you would define

```
l = mpctools.getCasadiFunc(@(x) x'*x, [2], {'x'});
```

The function `nmpc()` will then check the argument number and names.

Note that each name must exactly correspond to the name of a variable (or parameter; see next section) in the optimization problem, e.g., `'x'`, `'u'`, etc. If, for some reason, you cannot define the functions with the correct argument names, you can also use a `'funcargs'` argument as follows:

```
l = mpctools.getCasadiFunc(@(x) x'*x, [2], {'not_x'});
funcargs = struct();
funcargs.l = {'x'};
mpctools.nmpc('l', 1, 'funcargs', funcargs, ...
```

Without `funcargs`, this code would otherwise lead to an error.

Problem Parameters Often, your problem may have parameters that you would like to change after each optimization, or even parameters that change within the horizon of the current optimization problem. To support these situations, `nmpc()`, `nmhe()`, and `sstarg()` all take a `'par'` keyword argument in which you should pass a struct of parameter values. These parameters can be scalars, vectors, or matrices. To make the parameters time-varying, time should be running along the second dimension (third in the case of matrix parameters; see `gainscheduling.m` for an example).

To use these parameters in each function, you simply define the function as taking additional named arguments, for example

```
function cost = stagecost(x, u, xsp, usp)
    dx = x - xsp;
    du = u - usp;
    cost = dx'*dx + du'*du;
end%function
l = mpctools.getCasadiFunc(@stagecost, ...
    [2, 2, 3, 3], {'x', 'u', 'xsp', 'usp'});
```

To track a time-varying setpoint, you then define `par.xsp` and `par.usp` as matrices with each column giving the setpoint for that time (see `timevaryingmpc.m` for an example).

Time-varying parameters are accessed modulo length. For example, if your parameters have a period of 10 and your prediction horizon is 25, you only need to provide the 10 unique parameters, and for times 11 and beyond, the appropriate column will be chosen.

2 Common Mistakes

Below we list some common issues that may cause headaches.

- 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 Disclaimer

Note that since CasADi is in active development, MPCTools will need to be updated to reflect changes in CasADi's Octave/Matlab 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 previously, the latest files can always be found on <https://bitbucket.org/rawlings-group/octave-mpctools>. For questions, comments, or bug reports, please open an issue on Bitbucket or contact us by email.

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