MPCTools Documentation

1 Introduction

MPCTools is a model predictive control (MPC) oriented interface to CasADi for Octave and Matlab. It is intended to be a replacement for the legacy `mpc-tools` developed by the Rawlings group from 2000 to 2010. Here we document the Octave/Matlab interface, which is available as of CasADi v3.1.

Current development sources can be found at `https://bitbucket.org/rawlings-group/octave-mpctools`.

2 General Function Usage

Each of these functions can take arguments positionally and as keyword, value pairs (with keywords given as strings). Just as in Python, there can be a sequence of positional arguments followed by a sequence of (keyword, value) arguments. Note that allowing both types of arguments creates an ambiguity for positional arguments that are also strings. Thus, to pass a string positional argument, you must wrap the string in a cell array. Note that structs can also be used using a keyword of `**` (similar to Python's `**` operator). For arguments that are not supplied, the default value (if any) will be used. Passing an empty matrix `[]` for any argument also specifies that the default is requested (similar to passing `None` in Python).

As an example, consider a function `func` that takes two numeric arguments "x" (no default value) and "y" (with default value 1), as well as a string argument "name" (default value `'func'`). Each of the following function calls is equivalent:

- `func(0)` (Positional arguments with implied defaults)
- `func(0, [], [])` (Positional arguments with explicit defaults)
- `func(0, 1, \{'name'\}, 'func')` (Positional string argument)
- `func(0, 'y', 1, 'name', 'func')` (Mix of positional and keyword arguments)
- `func('x', 0, 'y', 1)` (Keyword arguments with implied defaults)
- `func('x', 0, 'name', 'func')` (Keyword string argument)
- `func(0, '**'struct('y', 1, 'name', 'func'))` (Struct with some keyword arguments)
- `func('**', struct('x', 0, 'y', 1))` (Struct of all keyword arguments)
- `func(0, 'y', 1, '**', struct('name', 'func'))` (All three input types)

For readability, it is best to use all keyword arguments for functions with more than roughly 4 arguments.

Note that as features are added to MPCTools, additional keyword arguments may be added to functions. In general, these arguments will be added at the end of the argument list. However, from time to time, an argument will be added earlier in the list, thus shifting the position of all arguments that follow. Thus, to avoid compatibility issues with future versions of MPCTools, you should only use positional arguments for the arguments that are listed in the function signature (i.e., given in the first line of the documentation string before `...`). For all arguments not listed in the function signature, you should use keyword, value pairs.

Documentation for specific functions is given below.

3 Optimization Variables and Parameters

Within each optimization, symbols x are used for states, u for controls, y for measured outputs, w for estimated state disturbances, and v for estimated measurement noise. These are considered the "standard" variables, as they are present in many control, MHE, and steady-state target problems. By default, the call signature for each function used within the optimization problem is taken from the variable names of the corresponding CasADi function. For example, to use the standard $f(x, u)$, you should define the CasADi function `f` as

```
f = mpctools.getCasadiFunc(@f, [Nx, Nu], {'x', 'u'})
```

with the third argument to `getCasadiFunc()` giving the variable identifiers as strings. If you do not provide these names, or if you need to override them, you can use the `'funcargs'` keyword to `nmpc`, `nmhe`, and `sstarg` (see full function documentation in Section 5).

For control problems, setpoints are denoted x_{sp} and u_{sp} (written `'xsp'` and `'usp'` as string identifiers). To use these setpoints in the stage cost, use arguments `'xsp'` and `'usp'` in its definition. For example, using the standard quadratic stage cost $\ell(x, u, t) = (x - x_{\text{sp}}(t))' Q (x - x_{\text{sp}}(t)) + (u - u_{\text{sp}}(t))' R (u - u_{\text{sp}}(t))$, you should define

```
function cost = stagecost(x, u, xsp, usp)
    Q = 1;
    R = 1;
    dx = x - xsp;
    du = u - usp;
    cost = dx'*Q*dx + du'*R*du;
end
l = mpctools.getCasadiFunc(@stagecost, [Nx, Nu, Nx, Nu], {'x', 'u', 'xsp', 'usp'})
```

Note that the (time-varying) values of x_{sp} and u_{sp} are stored in `ControlSolver.par.xsp` and `ControlSolver.par.usp`. See example script `timevaryingmpc.m` for complete example usage.

Changes in outputs are denoted Δu (string `Du`). These can be used in `nmpc` in one of two ways: first, if there is a `Du` field in any of the `'lb'`, `'ub'`, or `'guess'` arguments; and second, if there is a `'Du'` argument in any of the functions `f`, `l`, or `e`.

To facilitate one-norm or other linear objectives, we also provide the special variables `'absx'`, `'absu'`, and `'absDu'`. These are meant to be surrogates for $|x|$, $|u|$, and $|\Delta u|$ in the sense that they are defined as

$$\begin{aligned} \text{abs}x &\geq x \\ \text{abs}x &\geq -x \end{aligned}$$

with similar expressions for u and Δu. Thus, they will be exactly equal to the corresponding absolute value as long as they are included in a convex positive-definite penalty term in the objective.

For parameters besides x_{sp} and u_{sp}, we use the special identifier p for time-varying vector-valued parameters. Any user-defined parameters can be given arbitrary variable names (provided they do not clash with the reserved names discussed in this section) and passed in the `'par'` argument to `nmpc`, `nmhe`, and `sstarg`. Note that these parameters can be vectors or matrices, but they must be constant-in-time. They can be used as function arguments as with x_{sp} and u_{sp}.

4 Solver Options

To facilitate setting solver options, the `ControlSolver` class takes keyword arguments `verbosity`, `timelimit`, etc., to set common solver options. These values will be translated into the solver-specific option names and applied to the solver object. Note that these values can also be passed as keyword arguments to `nmpc()`, `nmhe()`, and `sstarg()`. See Section 6 for more details.

To set solver-specific options that do not have a common name, you should use `ControlSolver.init()`. This method takes a set of `'key', value` pairs. For example, IPOPT has a `tol` parameter that controls the relative convergence tolerance. To set this value, you would use, e.g.,

```
controller = mpctools.nmpc(..., 'solver', 'ipopt');
controller.init('tol', 1e-10);
```

To see what options are available for your particular solver, use `ControlSolver.getoptions()` (not available for some solvers) or consult the CasADi documentation.

5 Public API Documentation

Below, we document all functions in the MPCTools public API. Function arguments are generally listed as

- `argname` : Argument Type [`default value (if any)`]

This information can also be accessed in Octave/Matlab using the typical `help` function with an `'mpctools'` prefix, e.g., `help('mpctools.getCasadiFunc')`.

5.1 getCasadiFunc

`[fcasadi, qcasadi] = getCasadiFunc(f, varsizes, [varnames], [funcname='f'], ...)`

Returns a Casadi function using the function handle `f`.

Inputs are as follows:

- `f`: Function Handle

  Handle to function that is being transformed.

- `varsizes`: Row Vector

  Gives the number of elements in each input argument (all must be vectors).

- `varnames`: Cell Array

  List of strings for variable names. Defaults to generic names `x_1`, `x_2`, etc.

- `funcname`: String

  Name to use for function.

- `rk4`: Logical [`false`]

  Whether the function should be discretized with an explicit RK4 method. Note that the first argument of `f` is assumed to be the differential state.

- `Delta`: Scalar

  Timestep to use if `rk4` is `true`.

- `M`: Integer

  Number of iterations of RK4 to perform. Note that the total timestep is always equal to `Delta`, so the stepsize for each iteration is `Delta/M`.

- `quad` : Function Handle

  Handle to function that should be integrated over the interval. Note that `quad` must accept the same arguments (in the same order) as `f`. If provided, `rk4` must be `true`.

- `quadname` : String

  Name to use for quadrature function. Default is `['Q', funcname]`.

- `casaditype` : String [`'SX'`]

  String `'SX'` or `'MX'` to decide which type of CasADi symbolic variables to use. In general `'SX'` should be used for primitive algebraic operations (e.g., ODE right-hand sides), and `'MX'` should be used for higher-level constructs. Consult the CasADi User Guide for more details.

  Note that previous versions allowed the use of a logical keyword argument `scalar`, with `scalar=true` indicating `'SX'` and `scalar=false` indicating `'MX'`. This option has been removed and should be replaced by `casaditype`.

The outputs are casadi `Function` objects. Note that `qcasadi` is only defined if a `quad` argument was passed.

5.2 getCasadiIntegrator

`integrator = getCasadiIntegrator(f, Delta, argsizes, argnames, ...)`

Returns a Casadi `Integrator` object.

Arguments are as follows:

- `f` : Function

Function that defines the ODE right-hand side. Note that the first argument of `f` must be the differential variables x.

- `Delta` : Positive Scalar

  Integration timestep.

- `argsizes` : Cell or Vector

  Gives the size of each argument to `f`

- `argnames` : Cell of Strings

  Gives names for each argument.

- `funcname` : String

  Name for the Integrator object. Must be a valid variable identifier.

- `wrap` : Logical [`true`]

  Whether to wrap the Integrator object so that it can be called via $f(x, u, p, ...)$. If `false`, it must be called as `f('x0', x, 'p', vertcat(u, p, ...))`, so you will typically always want `wrap` to be `true`.

- `Nt` : Integer [1]

  Number of time points to include. If `Nt` is larger than 1, all of the arguments of `f` must be vectors. The returned function will then take a vector for x and matrices for all other parameters (with time along the second dimension), and will return a matrix of the next `Nt` states.

  Note that if `wrap` is `false`, this argument has no effect.

- `options` : Struct

  Struct of options to send to cvodes.

- `solver` : String [`'cvodes'`]

  Which solver to use for integration. Typical options are `'cvodes'`, `'idas'`, `'rk'`, and `'collocation'`. Consult the CasADi documentation for more information about these options.

- `casaditype` : String [`'SX'`]

  String `'SX'` or `'MX'` to decide which type of CasADi symbolic variables to use in the ODE expression. Consult the CasADi User Guide for more details.

  Note that previous versions allowed the use of a logical keyword argument `scalar`, with `scalar=true` indicating `'SX'` and `scalar=false` indicating `'MX'`. This option has been removed and should be replaced by `casaditype`.

5.3 getLinearizedModel

`model = getLinearizedModel(f, args, names, [Delta], [deal=false])`

Linearizes the model

$$\frac{dx}{dt} = f(x, u, ...)$$

the point `{xss, uss, ...}` given in `args`. `names` should be a cell array to define the fields into which each matrix should go. For example, to linearize the model

$$f(x, u) \approx Ax + Bu$$

you should use `names = {'A', 'B'}`, and `model` will be a struct with fields “A” and “B”.

`f` should be either a casadi.Function object or a native function handle.

If `Delta` is given, the model is also discretized with that timestep.

By default, the return value model is a struct whose fields are the strings in `names`. However, if `deal` is set to `true`, the function will return the individual matrices as multiple outputs, e.g.

```
    [A, B] = getLineraizedModel(f, {xss, uss}, {'A', 'B'}, 'deal', true())
```

Sometimes, this syntax is more convenient.

5.4 nmpc

`solver = nmpc(f, l, N, x0, lb, ub, guess, ...)`

Returns ControlSolver object for solving MPC control problems.

Inputs are as follows:

- `f`: Casadi Function

  Gives model for evolution of the system. By default, `f` is assumed to be in discrete-time, but continuous-time `f` can be used with collocation (see argument `N`).

- `l`: Casadi Function

  Gives the stage cost for the system. To include (possibly time-varying) setpoints for x and u, define `l` to take additional arguments called "xsp" and "usp", and provide the values for these setpoints in the `par` argument. To penalize rates of change, you may also use "Du" as an argument.

- `N`: Struct of Integers

  Contains fields "x", "u", and "t" to specify the dimension of x and u, as well as say how many time points to use. Optionally, it can contain a "c" entry to say how many (interior) collocation points to use (for a continuous-time model `f`).

- `x0`: Vector

  Gives the initial condition for the system. If not provided, no initial condition is used.

- `lb`: Struct
- `ub`: Struct
- `guess`: Struct

  Give the bounds and initial guess for the system variables. Each entry of these structs should have the time dimension last (e.g., field "x" should be of size `[N.x, N.t + 1]`, and "u" should be `[N.u, N.t]`). Any fields that aren't given default to $+\infty$, $-\infty$, and 0 respectively.

  These structs can also contain "Du" and "Dx" fields to specify rate-of-change bounds for the system.

  When using collocation, if these structs contain "x" entries and not "xc" entries, then values for "xc" will be inferred using linear interpolation. If you do not want this behavior, you need to explicitly provide the "xc" entry.

- `Vf`: Casadi Function

  Gives the terminal cost for the system (zero if not given).

- `Delta` : Scalar

  Defines the timestep. Must be provided to use collocation.

- `par` : Struct

  Defines values of fixed parameters. Special entries "xsp" and "usp" define time-varying setpoint values (and must be sizes `[N.x, N.t + 1]` and `[N.u, N.t]` respectively.

  Parameters can be vectors (including scalars) or matrices, and the values can be time-varying constants. For time-varying vector (or scalar) parameters, pass values as a matrix with each slice `p(:,t)` (i.e., each column) giving the value at the corresponding time point. For time-varying matrices, use a 3D array, with each slice p(:,:,t) giving the value at each time. Note that access is modulo length, so if a parameter p is passed as a 1 by 3 matrix, then its value will repeat every three time points regardless of the horizon N.t.

- funcargs : Struct

  Contains cell arrays for each function that define the sequence of arguments for the given function.

  By default, function argument names are read directly from the corresponding Casadi function. Thus, you only need to use funcargs if you created the Casadi function without names, or if you used names that are different from the names in the optimization problem.

- periodic : Logical [false]

  Determines whether or not to add a periodicity constraint to the problem.

- e : Casadi Function

  Function that defines path constraints $e(x, u) \leq 0$. Note that the arguments can be modified using funcargs.e. One constraint is written for each time point.

  To soften these constraints, include an "s" entry in N to give the number of slacks you need. Note that if funcargs.e does not contain "s", then the constraints are written as $e(x, u) \leq s$, which means N.s must correspond to the number of components of e. Otherwise, you may use "s" as an explicit argument to e (e.g., if you want to only soften certain constraints).

- ef : Casadi Function

  Defines the terminal constraint $e_f(x(N)) \leq 0$.

- udiscrete : Logical Vector

  Vector of true and false to say whether components of u should be discretely (i.e., integer) valued. Note that to actually enforce this restriction in the optimization problem, you will need to choose a solver that supports discrete variables (e.g., bonmin).

- uprev : Column Vector

  Gives the previous value of u to calculate the first rate of change.

- discretel : Logical [true]

  Indicates whether the objective function should be a discrete sum of stage costs (true) or a collocation-based quadrature (false). Note that the latter requires N.c > 0.

- casaditype : String ['SX']

  Chooses which Casadi type to use for the NLP. If the functions are all matrix-based or more complicated (e.g., Casadi integrators), or if the NLP is very large then 'MX' is usually best. If the functions are fairly simple (e.g., a sequence of scalar operations) and if the NLP is small to medium size, then 'SX' is typically better. Default is 'SX'.

- xf : Column Vector

  Gives the terminal value for x. If provided, a terminal equality constraint is added. You may consider using a large terminal penalty instead if you encounter solver issues.

- singleshooting : Logical [false]

  Specifies whether to use single shooting to remove the system model from the NLP, which trades problem size for sparsity. Note that single shooting cannot be used with collocation.

- h : Casadi Function

  If supplied, y variables are added to the formulation with the constraint y = h(x). y can then be used like any other variable (e.g., supplying bounds, using it in l, etc).

  Note that if h is specified, you also must specify N.y in N.

- finaly : Logical [true]

Decides whether to include `y` at the final time point or not. Default is `true`, which means it is included and can be used in a terminal constraint. Note that if `h` takes `u` as an argument, then the first `u` will be used, and so unexpected behavior may result.

- `g` : Casadi Function

  If supplied, indicates that the model is a DAE system of the form `x^+ = f(x,z,u), g(x,z) = 0` with differential states `x` and algebraic states `z`.

  As with any other function, you can define `g` to take any set of arguments. Note that if `g` takes `u` as an argument, then the constraint at time `N.t` is written using `u(N.t - 1)`.

- `customvar` : Cell array of strings

  List of custom variables to add to the problem. Note that each variable must have been used in one of the functions so that sizes can be inferred. Custom variables are also time-invariant, i.e., there is only one copy that is used everywhere.

  Bounds on custom variables can be included in `lb` and `ub`. Otherwise, variables are unbounded.

The output is a ControlSolver object.

5.5 nmhe

`solver = nmhe(f, h, u, y, l, N, lx, x0bar, lb, ub, guess, ...)`

Returns ControlSolver object for solving MPC control problems.

Inputs are as follows:

- `f`: Casadi Function

  Gives model for evolution of the system. By default, `f` is assumed to be in discrete-time, but continuous-time `f` can be used with collocation (see argument `N`).

- `h` : Casadi Function

  Gives measurement function.

- `u` : Matrix
- `y` : Matrix

  Give the known values for u and y. Should be sized `[N.u, N.t]` and `[N.y, N.t + 1]` respectively. Both can also be given in the `par` struct, but values given as arguments override those in `par`.

  Note that `u` is only needed if the model has inputs, while `y` must always be given (either as an argument or in `par`).

- `l`: Casadi Function

  Gives the stage cost for the system error.

- `N`: Struct of Integers

  Contains fields "x", "u", "y" and "t" to specify the dimension of x, u, and y, as well as say how many time points to use. Optionally, it can contain a "c" entry to say how many (interior) collocation points to use (for a continuous-time model `f`). It can also contain a "d" entry to give the number of disturbance model states, which allows `d` to be used as an argument in `f`, `h`, and `l` (the change in `d`, `Dd` can also be used in `l`).

- `lx`: Casadi Function

  Gives the arrival cost for x0 (zero if not given). Note that if priorupdate (see below) is anything other than 'none', then this argument cannot be specified, as the default quadratic prior is used.

- `x0bar`: Vector

  Gives the prior value to use for $\bar{x}_0$ in `lx`. Can also be given in `par`.

- `lb`: Struct
- `ub`: Struct
- `guess`: Struct

  Give the bounds and intial guess for the system variables. Each entry of these structs should have the time dimension last (e.g., field "x" should be of size `[N.x, N.t + 1]`, and "u" should be `[N.u, N.t]`). Any fields that aren't given default to $+\infty$, $-\infty$, and 0 respectively.

  When using collocation, if these structs contain "x" entries and not "xc" entries, then values for "xc" will be inferred using linear interpolation. If you do not want this behavior, you need to explicitly provide the "xc" entry.

- `Delta` : Scalar

  Defines the timestep. Must be provided to use collocation.

- `par` : Struct

  Defines values of fixed parameters. Special entries "xsp" and "usp" define time-varying setpoint values (and must be sizes `[N.x, N.t + 1]` and `[N.u, N.t]` respectively.

  Parameters can be vectors (including scalars) or matrices, and the values can be time-varying constants. For time-varying vector (or scalar) parameters, pass values as a matrix with each slice `p(:,t)` (i.e., each column) giving the value at the corresponding time point. For time-varying matrices, use a 3D array, with each slice `p(:,:,t)` giving the value at each time. Note that access is modulo length, so if a parameter `p` is passed as a 1 by 3 matrix, then its value will repeat every three time points regardless of the horizon `N.t`.

- `funcargs` : Struct

  Contains cell arrays for each function that define the sequence of arguments for the given function.

  By default, function argument names are read directly from the corresponding Casadi function. Thus, you only need to use `funcargs` if you created the Casadi function without names, or if you used names that are different from the names in the optimization problem.

- `wadditive` : Logical [`false`]

  Decides whether w is an additive disturbance, i.e., $x^+ = f(x,u) + w$ or is explicitly included in the model, i.e., $x^+ = f(x,u,w)$.

  When collocation is used, this choice means that w is the difference between the state at the right collocation *endpoint* and the state at the next time point. In terms of the problem variables, interval `k`'s collocation points are given by

  ```
  [x(:,k), xc(:,:,k), x(:,k + 1) - w(:,k)]
  ```

  and the value of w is the instantaneous jump at the very end of the interval.

- `penalizevN` : Logical [`true`]

  Decides whether the final measurement error $v(N)$ should be penalized. If true, an extra $l(0, v(N))$ term is added to the objective function (i.e., $l(w,v)$ with $w = 0$. Note that if you have custom arguments for l, only w is set to zero.

- `casaditype` : String [`'SX'`]

  Chooses which Casadi type to use for the NLP. If the functions are all matrix-based or more complicated (e.g., Casadi integrators), or if the NLP is very large then `'MX'` is usually best. If the functions are fairly simple (e.g., a sequence of scalar operations) and if the NLP is small to medium size, then `'SX'` is typically better. Default is `'SX'`.

- `g` : Casadi Function

  If supplied, indicates that the model is a DAE system of the form $x^+ = f(x,z,u)$, $g(x,z) = 0$ with differential states x and algebraic states z.

As with any other function, you can define `g` to take any set of arguments. Note that if `g` takes `u` as an argument, you should supply an extra value of `u` (i.e., give `N.t + 1` values of `u`) to use for the final constraint. Otherwise, the constraint at time `N.t` is written using `u(N.t - 1)`.

- `e` : Casadi Function

  Function that defines path constraints $e(x, u) \leq 0$. Note that the arguments can be modified using `funcargs.e`. One constraint is written for each time point.

  To soften these constraints, include an "s" entry in `N` to give the number of slacks you need. Note that if `funcargs.e` does not contain "s", then the constraints are written as $e(x, u) \leq s$, which means `N.s` must correspond to the number of components of `e`. Otherwise, you may use "s" as an explicit argument to `e` (e.g., if you want to only soften certain constraints).

- `singleshooting` : Logical [`false`]

  Specifies whether to use single shooting to remove the system model from the NLP, which trades problem size for sparsity. Note that single shooting cannot be used with collocation.

- `priorupdate` : String [`'none'`]

  Specifies the prior update to use. Available options are as follows (with k referring to the initial time of the MHE problem):

  - `'none'` : No automatic updates to prior parameters.
  - `'filtering'` : Updates prior parameters using an EKF step applied to $\hat{x}(k - N|k - N)$.
  - `'smoothing'` : Updates prior parameters using $\hat{x}(k-N|k-1)$. Note that a correction is added to prevent "double-counting" of the data $y(k - N)$ through $y(k)$.
  - `'hybrid'` : Identical to 'filtering' except that $\hat{x}(k - N|k - 1)$ is used instead of $\hat{x}(k - N|k - N)$.

  For any choice besides `'none'`, the arrival cost is of the form $\ell_x(x) = (x - \bar{x}_0)P^{-1}(x - \bar{x}_0)$ with parameter `Pinv` giving the quadratic weight P^{-1}, and parameter `x0bar` giving the the minimum value $\bar{x}_0$. This function is supplied automatically, which means the `lx` argument should not be specified. Initial values for `x0bar` and `Pinv` must both be specified in the `par` struct. These parameters are then updated when `MHESolver.saveestimate()` is called.

  Note that prior updates are not supported for DAE models.

  For a linear system with uncorrelated w and v, the `'filtering'` and `'smoothing'` updates are equivalent to the (time-varying) Kalman Filter. `'hybrid'` is not equivalent to the Kalman Filter but is faster and can give better results for nonlinear systems.

- `customvar` : Cell array of strings

  List of custom variables to add to the problem. Note that each variable must have been used in one of the functions so that sizes can be inferred. Custom variables are also time-invariant, i.e., there is only one copy that is used everywhere.

  Bounds on custom variables can be included in `lb` and `ub`. Otherwise, variables are unbounded.

The output is an MHESolver object (subclass of ControlSolver with some extra methods specific to MHE problems).

5.6 sstarg

```
solver = sstarg(f, h, l, N, lb, ub, guess, ...)
```

Returns ControlSolver object for solving MPC control problems.

Inputs are as follows:

- `f`: Casadi Function

  Gives model for evolution of the system. By default, `f` is assumed to be in discrete-time, but continuous-time `f` can be used with collocation (see argument `N`).

- `h` : Casadi Function

Gives measurement function. If not given, the variable y is not included in the optimization problem.

- `l`: Casadi Function

  Gives the objective function for the system. If included, `l` must have been defined with names, or funcargs.l must also be provided.

  If not given, a dummy objective is used so that any feasible solution is also optimal.

- `N`: Struct of Integers

  Contains fields "x", "u", "y" and "t" to specify the dimension of x, u, and y, as well as say how many time points to use. Optionally, it can contain a "c" entry to say how many (interior) collocation points to use (for a continuous-time model `f`).

- `lb`: Struct
- `ub`: Struct
- `guess`: Struct

  Give the bounds and initial guess for the system variables. Each entry of these structs should have the time dimension last (e.g., field "x" should be of size `[N.x, N.t + 1]`, and "u" should be `[N.u, N.t]`). Any fields that aren't given default to $+\infty$, $-\infty$, and 0 respectively.

  These structs can also contain "Du" fields to specify rate-of-change bounds for the system.

- `par` : Struct

  Defines values of fixed parameters. Special entries "xsp" and "usp" define time-varying setpoint values (and must be sizes `[N.x, N.t + 1]` and `[N.u, N.t]` respectively.

  Parameters can be vectors (including scalars) or matrices, and the values can be time-varying constants. For time-varying vector (or scalar) parameters, pass values as a matrix with each slice `p(:,t)` (i.e., each column) giving the value at the corresponding time point. For time-varying matrices, use a 3D array, with each slice `p(:,:,t)` giving the value at each time. Note that access is modulo length, so if a parameter `p` is passed as a 1 by 3 matrix, then its value will repeat every three time points regardless of the horizon `N.t`.

- `funcargs` : Struct

  Contains cell arrays for each function that define the sequence of arguments for the given function.

  By default, function argument names are read directly from the corresponding Casadi function. Thus, you only need to use `funcargs` if you created the Casadi function without names, or if you used names that are different from the names in the optimization problem.

- `e` : Casadi Function

  Function that defines constraints for the system. If given, either `e` must have been defined with names, or `funcargs.e` must be provided.

  To soften these constraints, include an "s" entry in `N` to give the number of slacks you need. Note that if `funcargs.e` does not contain "s", then the constraints are written as $e(x, u) \leq s$, which means `N.s` must correspond to the number of components of `e`. Otherwise, you may use "s" as an explicit argument to `e` (e.g., if you want to only soften certain constraints).

- `discretef` : Logical [`true`]

  If `true`, the model `f` is assumed to be in discrete time, and the constraint is written as $f(x) = x$. If `false`, `f` is assumed to be continuous time, and the constraint is $f(x) = 0$.

- `udiscrete` : Logical Vector

  Vector of `true` and `false` to say whether components of u should be discretely (i.e., integer) valued. Note that to actually enforce this restriction in the optimization problem, you will need to choose a solver that supports discrete variables (e.g., bonmin).

- `casaditype` : String [`'SX'`]

Chooses which Casadi type to use for the NLP. Since `sstarg` problems are typically quite small, `'SX'` is almost always the best choice. However, if the model `f` includes any nonscalar operations (e.g., Casadi Integrator calls), then you will need to us `'MX'`.

- `g` : Casadi Function

  If supplied, indicates that the model is a DAE system of the form `x^+ = f(x,z,u), g(x,z) = 0` with differential states `x` and algebraic states `z`.

- `customvar` : Cell array of strings

  List of custom variables to add to the problem. Note that each variable must have been used in one of the functions so that sizes can be inferred. Custom variables are also time-invariant, i.e., there is only one copy that is used everywhere.

  Bounds on custom variables can be included in `lb` and `ub`. Otherwise, variables are unbounded.

The output is a ControlSolver object.

5.7 mpcplot

`[vals, xax, uax] = mpcplot(x, u, [t], [xsp], [xc], [tc], ...)`

Makes a plot of the given MPC solution.

Arguments are as follows:

- `x` : Array

  Array of size `[Nx, Nt + 1]` that gives the values of the states at the discrete time points. Plotted normally.

- `u` : Array

  Array of size `[Nu, Nt]` that gives the values of the controls during the discrete time windows. Plotted using zero-order hold (stairstep). If omitted, only x is plotted.

  Alternatively, can be size `[Nu, Nt + 1]`, which will lead to a first- order hold plot.

- `t` : Vector [`0:Nt`]

  Vector giving the time values at each of the discrete time points.

- `xsp` : Array

  If given, defines the setpoints for the state. If size `[Nx, Nt]`, it is plotted using a zero-order hold (stairstep). Otherwise, it must be size `[Nx, Nt + 1]`, and it is plotted using a first-order hold (normal).

- `xc` : Array

  Array of states at (interior) collocation points. Must be size `[Nx, Nc, Nt]`. Used to interpolate the plot of x.

- `tc` : Array

  Array of time points for the (interior) collocation points. Must be size `[Nc, Nt]`. If not given, it is assumed that the collocation times are roots of Legendre polynomials.

- `usp` : Array

  Similar to `xsp`, but for `u`.

- `plot` : Logical [`true`]

  Whether or not to actually perform the plotting. If `false`, this function will simply return a struct with all the (properly sized) values that would have been plotted.

- `marker` : String [`''`]

- `xmarker` : String

- `umarker` : String

  Plot markers to use. The value of `marker` sets the marker to use for both x and u. To set them individually, provide `xmarker` and/or `umarker`, which overrides `marker` for each variable.

  Pass the empty string (`''`) to not use any marker.

- `collocmarker` : String [`''`]

  Marker to use for collocation points (if provided). Default is to use no marker.

- `fig` : Figure Handle

  Figure handle to use for plotting. Default is to make a new figure. This is useful if you wish to plot multiple datasets on the same figure.

- `title` : String [`''`]
- `timelabel` : String [`'Time'`]

  Strings to use for the window title and bottom x-axis (time-axis) label.

- `legend` : String [`''`]

  String to use as a legend entry. If nonempty, will add a legend to the figure.

- `legendloc` : String [`'North'`]

  String that specifies where legend should be. Should be in the format of Octave/Matlab legend locations (e.g., `'West'`, `'NorthEast'`, etc.).

- `color` : String or RGB array
- `spcolor` : String or RGB array

  Colors to use for data (i.e., x and u) and setpoint respectively. Both default to black.

- `xnames` : Function or Cell Array of Strings
- `unames` : Function or Cell Array of Strings

  Defines labels for the states and controls respectively. Used as y-axis labels on each subplot.

  If a function, for each label, the function is called with a single argument of the integer index of the variable. If a cell array of strings, the strings are used directly.

- `labelrot` : Integer

  Sets the rotation for y-axis labels (in degrees). By default, 0 rotation if all labels are short (five characters or fewer), otherwise 90.

- `linestyle` : String [`'-'`]
- `splinestyle` : String [`':'`]

  Chooses which type of line to use for the plots. (`splinestyle` is line style for setpoint `xsp` if given).

The output `vals` is a struct with fields “x”, “u”, “t”, etc. that have all data properly resized to have `Nt + 1` time points. It is useful if you want to make your own plots but don’t want to have to reshape everything yourself.

Other outputs `xax` and `uax` are vectors of axes handles for `x` and `u` respectively. Note that they will be empty if `plot` is `false`.

5.8 spdinv

```
[Ainv, Achol] = spdinv(A)
```

Computes inverse of symmetric positive-definite matrix `A` via (upper-triangular) Cholesky factorization. `A` must be given as a single positional argument.

Also returns the Cholesky factor `Achol` (with `Achol'*Achol == A`).

Note that `A` can be a 3D array, at which point a 3D array of inverses is returned assuming each each `A(:,:,i)` slice is a matrix.

5.9 rk4

`x = rk4(f, x0, [par={}], [Delta=1], [M=1])`

`[q, x] = rk4(..., 'quad', Q)`

Does `M` steps of explicit rk4 integration with a total time of `Delta`.

- `f` : Function

  ODE right-hand side function.

- `x0` : Vector

  Initial condition.

- `par` : Cell Array

  Parameters (extra arguments) to `f` (called as `f(x0, par{:})`).

- `Delta` : Positive Scalar

  Timestep to use for integration.

- `M` : Positive Integer

  Number of steps to take. Note that the total time is always `Delta`, and so the duration of each step is Δ/M.

- `quad` : Function

  Quadrature function. If provided, must take the same arguments as f. Note that this makes the first return value equal to the quadrature rather than the state at the next time point.

Returned values are `x`, the approximate value of `x` after `Delta` time units; and `q`, the approximate integral of `Q(x, ...)` over the timestep.

5.10 ekf

`[Pm, xhatm, P, xhat] = ekf(f, h, x, u, w, y, Pm, Q, R, [projectionfunc])`

Updates the prior distribution P^- using the Extended Kalman filter.

`f` and `h` should be Casadi functions. `f` must be discrete-time. P, Q, and R are the prior, state disturbance, and measurement noise covariances (in particular, the input `Pm` refers to $P(k|k-1)$).

Note that `f` must be $f(x,u,w)$ and `h` must be $h(x)$.

The value of `x` that should be fed is $\hat{x}(k|k-1)$, and the value of `P` should be P(k | k-1). xhat will be updated to xhat(k | k) and then advanced to $\hat{x}(k+1|k)$, while `P` will be updated to $P(k|k)$ and then advanced to $P(k+1|k)$. The return values are a list as follows

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

Depending on your specific application, you will only be interested in some of these values.

The optimal argument `projectionfunc` is a function handle that is called to project x back into the feasible space after the correction step. It should take x as its only argument and return the projected version of x. For example, if x is nonnegative, using

```
projectionfunc = @(x) max(x, 0)
```

would project x back into the nonnegative orthant before advancing.

5.11 c2d

```
[...] = c2d(Delta, a, b, [q, r], [m], [g, h], [quad], [Nquad], [return])
```

Discretization with continuous objective.

Converts from continuous-time objective

$$l(x,u) = \int_0^\Delta x'qx + 2x'mu + u'ru + g'x + h'u \, dt \qquad dx/dt = ax + bu$$

to the equivalent (assuming `u` is constant on $[0, \Delta]$).

$$L(x,u) = x'Qx + 2x'Mu + u'Qu + G'x + H'u \qquad x^+ = Ax + Bu$$

in discrete time.

Note that `q` can be given if and only if `r` is given (similarly `g` and `h`).

Optional argument `quad` decides whether to use approximate quadrature to compute the Q, R, and M matrices (default `false`). `Nquad` is the number of steps to use in the quadrature (default 100). Quadrature may be necessary when `Delta*a` has eigenvalues with real part less than -25 or greater than 25.

Argument `return` decides what is returned. The default value is 'struct', which returns a struct with fields "A" and "B", as well as fields "Q", "R", and "M" if arguments `q` and `r` were given. `return` can also be a string consisting of "ABQRMGH", at which point, the individual matrices given in the string will be returned.

Reference: C. Van Loan, 1978, "Computing integrals involving the matrix exponential".

5.12 nlfilter

```
[xhat, status] = nlfilter(h, xhatm, y, Rinv, Pinv, [xlb], [xub])
```

Solves the nonlinear filtering problem

$$\min_x (x - \hat{x}^-)'P^{-1}(x - \hat{x}^-) + (y - h(x))'R^{-1}(y - h(x))$$

This is essentially a zero-step MHE problem with quadratic prior and cost.

Arguments are as follows:

- `h` : Casadi Function

  Measurement function giving $y = h(x)$.

- `xhatm` : Column Vector

  Numerical estimate for $\hat{x}^-$, i.e., $\hat{x}(k|k-1)$.

- `y` : Column Vector

  Numerical value for the current measurement $y(k)$.

- `Rinv` : Matrix
- `Pinv` : Matrix

  Penalty matrices R^{-1} and P^{-1}. Note that they should already be inverted when they are passed as arguments.

- `xlb` : Column Vector
- `xub` : Column Vector

  Bounds to enforce on x in the optimization problem. Useful if h is undefined for certain values of x and you need to restrict the domain.

The two outputs are the column vector `xhat`, which is the optimal estimate $\hat{x}(k|k)$ and a string `status` that gives the status of the optimization problem.

5.13 collocweights

`[r, a, b, q] = collocweights(N, ['left'], ['right'])`

Returns collocation weights based on roots of Legendre polynomials.

The first argument `N` gives the number of interior collocation points. The second and third arguments can be the strings 'left' or 'right' to add additional collocation points on the boundary.

Note that all arguments must be passed as positional arguments.

In Octave, simply calls builtin colloc. In Matlab, uses m-file implementation.

Reference: J. Villadsen, M. L. Michelsen, "Solution of Differential Equation Models by Polynomial Approximation".

5.14 getCasadiDAE

`[dae, argorder] = getCasadiDAE(Delta, f, [g], ...)`

Arguments are as follows:

- `Delta` : Positive Scalar

  Integration timestep.

- `f` : Casadi Function

  Function that defines the ODE right-hand side.

- `g` : Casadi Function

  Function that defines the algebraic constraints for the system. Note that when there is overlap between arguments of `f` and `g`, the sizes must be consistent.

  If `g` is not given, then the integrator is just a normal ODE.

- `funcname` : String [`'dae'`]

  Name for the Integrator object. Must be a valid variable identifier.

- `diffstate` : Integer or String [1]
- `algstate` : Integer or String [2]

  Integer giving the position of or string giving the name of the variables that define the differential variables x and algebraic variables z.

- `options` : Struct

  Struct of options to send to the solver.

- `solver` : String [`'idas'`]

  Which solver to use for integration. Typical options are `'idas'` and `'collocation'`. Consult the CasADi documentation for more information about these options. Note that not all integrators support DAEs.

The returned value `dae` is a CasADi Integrator object that returns the values of `x` and `z` after `Delta` time units. Arguments are passed in the order given in `argorder`, which is arguments of `f` in order followed by arguments unique to `g`.

5.15 version

`version()`

Returns the version of MPCTools as a string.

The second output argument returns the HG changeset ID (as a hexidecimal string), which is more granular than the version number.

6 ControlSolver Documentation

Below, we document all public attributes and member functions to the `ControlSolver` class. Note that, in contrast to the functions in the previous section, these member functions accept only positional arguments.

6.1 ControlSolver

Class for holding an NLP solver for optimal control problems. Note that the user should not create instances of this class directly (i.e., using the constructor), but should instead use the problem-specific interfaces `nmpc`, `nmhe`, and `sstarg` (all of which return `ControlSolver` objects).

Public attributes are as follows:

- `par`
- `lb`
- `ub`
- `guess`
- `conlb`
- `conub`
- `discreteness`

  Structs that hold current values of parameters, bounds, etc. All time-varying entries have time along the final dimension.

  Since values can be changed directly by users, care should be taken to avoid changing the size of any fields, or else there will be odd error messages on the next call to `solve()`.

  Entries in `discreteness` should all either be true or false to say whether the particular variable is discrete-valued or not. Note that not all solvers support discrete variables.

- `var`

  Read-only struct that holds the most recent optimal solution. Before `solve()` is called, this field is empty.

- `stats`

  A struct that contains information about the most recent call to `solve()`. Note that for solvers other than IPOPT, this struct may be empty (due to the Casadi interface not populating these values).

- `status`

  A string containing the solver's most recent return status. When using IPOPT, success is indicated by `'Solve_Succeeded'`. For other solvers, it may not have a meaningful value.

- `obj`

  Objective value of most recent optimization.

- `verbosity` : Integer Between 0 and 12 [0]
- `timelimit` : Positive Scalar (in seconds) [60]
- `maxiter` : Positive Integer [∞]
- `isQP` : Logical [`false`]

  Solver-generic options that can be changed using `set_*()` functions. These are intended so that the user does not need to remember the solver- specific names for common options. However, not all solvers may provide access to all of these options. For example, due to CasADi limitations, when using the solver Bonmin, setting `verbosity=0` does not hide all output.

Note that all of these values can be passed as keyword arguments to `nmpc`, `nmhe`, and `sstarg` to avoid having to call the `set_*()` methods after the object has been built.

To change any other solver options, you will need to use the `init()` function and the solver-specific option names (see also `getoptions()`). Note that CasADi does not provide access to all solver-specific options.

6.2 ControlSolver.copy

`solver = self.copy()`

Returns a copy of the ControlSolver object in its current state.

6.3 ControlSolver.init

`self.init(options) self.init('key1', value1, ['key2', value2], ...)`

Initializes solver object using the provided `options` struct or using multiple `'key', value` pairs. See Casadi user guide for more information about available options.

Note that calling `init()` saves the current state of the object as the default state (i.e., the state that gets restored by calling `reset()`).

6.4 ControlSolver.solve

`self.solve()`

Solves the current optimization problem.

6.5 ControlSolver.import_solution

`self.import_solution(var, [stats], [sol])`

Imports a solution into the current object. This is useful, e.g., if you want to solve the optimization problem using a solver outside of CasADi.

`var` should be either the struct solution (i.e., in the format of `self.var`) or a single long vector of variables.

`stats` should be struct giving solution statistics. In particular, it should contain a "return_status" field. `sol` should be a struct giving extra solution information, e.g., dual multipliers, etc., in the form of `self.sol`. Note that both of these two arguments are optional.

6.6 ControlSolver.saveguess

`self.saveguess([newguess], [toffset])`

Stores a guess from a given struct or from the current solution.

If `newguess` is provided, all of its fields are copied to the current guess (`toffset` defaults to 0). If not, the current optimal solution is used (`toffset` defaults to 1). Note that any extra fields in the given guess are silently ignored. Also, any entries of `newguess` that are NaN will not be used.

`toffset` can be manually specified to explicitly say give the time offset to use. Note that only the overlapping time points are actually changed.

6.7 ControlSolver.fixvar

```
self.fixvar(var, t, val, [inds])
```

Sets guess, lb, and ub for variable `var` at time `t` to `val`.

If the variable is a vector, you can use `inds` to set only some of the components.

6.8 ControlSolver.truncatehorizon

```
self.truncatehorizon(newhorizon)
```

Truncates the horizon of the optimization problem by removing constraints and fixing variables to zero.

Note that if there are terminal costs or constraints, these are not shifted. Thus, if you really need the true shorter horizon problem, you will need to call the original function again with the appropriate horizon from the beginning.

6.9 ControlSolver.getQP

```
problem = getQP(self, [solver='gurobi'], [point])
```

Returns a struct of standard-form parameters to solve a quadratic approximation of the current NLP.

The `solver` argument chooses the output format. The default is `'gurobi'`, which returns a struct with Gurobi-compatible fieldnames. Also available are `'quadprog'`, which returns a struct for use with Matlab's `quadprog`, and `qp`, which returns a *cell array* of arguments that can be given to Octave's `qp` function. Note that only Gurobi supports discrete decision variables.

If given, `point` is a struct that defines the point to use for linearization. If not supplied, `self.guess` is used.

Note that the objective function is the nominal objective function, not the Lagrangian.

6.10 ControlSolver.getoptions

```
self.getoptions()
```

Returns the options available for the current solver as a string.

6.11 ControlSolver.xvec2struct

```
s = self.xvec2struct(v)
```

Reshapes the long vector `v` into a struct of named variables `s`.

6.12 ControlSolver.gvec2struct

```
s = self.gvectostruct(v)
```

Reshapes the long vector `v` into a struct of named constraints `s`. Useful for multipliers on constraints.

6.13 ControlSolver.set_solver

```
self.set_solver([solvername])
```

Sets the solver to use for optimization.

Note that setting the solver will reset any solver-specific options that have been set via `init()`.

6.14 ControlSolver.reset

Resets the object back to its default state.

6.15 ControlSolver.cyclepar

`self.cyclepar('par1', new1, ['par2, 'new2', ...])`

Cycles one or more time-varying parameters. For each parameter given, the oldest value is removed, everything is shifted forward by one spot, and the new value is inserted.

E.g., for parameter xsp, the call

```
   self.cyclepar('xsp', xspnew)
```

is equivalent to

```
   self.par.xsp = [self.par.xsp(:,2:end), xspnew]
```

This function works for time-varying scalars, vectors, and matrices. Note that it assumes that the horizon is full (i.e., `self.horizon == self.maxhorizon`).

7 MHESolver Documentation

Below, we document the public attributes and member functions of the `MHESolver` class. Note that `MHESolver` is a subclass of `ControlSolver`, and thus includes all of its methods as well. These functions take only positional arguments.

7.1 MHESolver

Class for holding an NLP solver for MHE problems. Note that the user should not create instances of this class directly (i.e., using the constructor), but should instead use the `nmhe` function.

In addition to all the properties from the `ControlSolver` class, this object also contains the following extra fields:

- `history`

  Array struct that holds a history of estimates for x, w, and v, as well as values of y and u. `self.history(t)` gives the estimate from `t` time periods ago. The current solution is stored to the history whenever `self.saveestimate()` is called.

  Keyword argument ‘Nhistory’ controls the number of past estimates to include in `history`. The default value is one more than the horizon of the MHE problem.

- `priorupdate` : String

  String indicating which type of prior update (if any) is being used. See documentation of `nmhe()` for more details.

- `fix_truncated_x` : Logical [`false`]

  Whether to fix the x variables that are outside of the current horizon. Usually this is unnecessary, but it can help with `"Restoration_Failed"` issues when the horizon is truncated. Note that you may need to provide a feasible guess for x if `fix_truncated_x` is true.

7.2 MHESolver.newmeasurement

`self.newmeasurement(y, [u], [x0bar])`

Shifts `par.y` to include the new measurement `y` at the current time. Also shifts `par.u` if `u` is given. If `x0bar` is given, its value is simply overwritten.

In contrast to `cyclepar()`, this function accounts for a shortened horizon, and it only removes old data if the horizon is full.

7.3 MHESolver.saveestimate

`self.saveestimate([updateguess=true])`

Saves the current estimates to the history struct. Also updates the prior weight if `self.priorupdate` has been specified.

If `updateguess` is `true` and the horizon is full, then the solver's guess is also updated.

7.4 MHESolver.reset

`self.reset()`

Resets the object to its initial state.

8 KalmanFilter Documentation

Below, we document all public attributes and member functions to the `KalmanFilter` class. The class constructor accepts both positional and keyword arguments, while the other member functions take only positional arguments.

8.1 KalmanFilter

Class for (offset-free) Kalman Filter.

Public attributes are as follows:

- `N`

  struct of system sizes. Fields are `x`, `u`, `y`, `d`, `w`, and `v`.

- `A`
- `B`
- `C`
- `D`
- `Ad`
- `Bd`
- `Cd`
- `Gx`
- `Gd`
- `Qw`
- `Rv`

  Matrices that define the system and disturbance model.

- `xhat`
- `dhat`

  Current estimates of state and integrating disturbances after the current measurement has been considered. These are updated via calls to `filter()`.

- `xhatm`
- `dhatm`

  Current estimates of state and integrating disturbances before the current measurement has been considered. These are updated via calls to `predict()`.

- `Lx`
- `Ld`

  Kalman Filter gains for the states and integrating disturbances.

- `Txd`

- Txy
- Tud
- Tuy

  Matrices for the steady-state target calculation. Multiply either d or ysp and return either xtarg or utarg.

8.2 KalmanFilter.filter

```
[xhat, dhat] = self.filter(y, [xhatm], [dhatm], [u])
```

Performs the Kalman Filter update using the current measurement y. New values are stored to `self.xhat` and `self.dhat`.

If not given, `xhatm` and `dhatm` are taken from the current values of the object. `u` is handled similarly, but it only matters if `self.D` is nonzero.

8.3 KalmanFilter.predict

```
[xhatm, dhatm] = self.predict(u, [xhat], [dhat])
```

Performs the Kalman Filter prediction using the current control action `u`. New values are stored to `self.xhatm` and `self.dhatm`.

If not given, `xhat` and `dhat` are taken from the current values of the object.

8.4 KalmanFilter.target

```
[xtarg, utarg] = self.target(ysp, [dhat], [rsp])
```

Calculates the steady-state target values of x and u using the given value of `ysp` or `rsp`.

If `rsp` is given, it is used directly. Otherwise, it is calculated as `rsp = self.H*(ysp - self.ys)`.