
# User Guide

M. Rieck, M. Bittner, B. Grüter, J. Diepolder, P. Piprek
Institute of Flight System Dynamics
Technical University of Munich
www.falcon-m.com – falcon-m@tum.de

February 19, 2020

# Contents

# 1 Welcome to FALCON.m

FALCON.m is the *FSD optimAL CONtrol tool for MATLAB* that has been developed at the Institute of *Flight System Dynamics* of *Technische Universität München*. FALCON.m uses direct discretization methods in combination with gradient based numerical optimization and automatic analytic differentiation to solve mathematical optimal control problems. It is mainly tailored to solving complicated "real life" problems using full discretization methods, additionally offering support for shooting techniques. If the technical details of these methods are unfamiliar to you, you might want to check section 4 of this document. If you are more interested in directly starting to solve problems, section 2 will guide you through the installation of FALCON.m and the implementation of your very first optimal control problem using the tool. Feel free to contact the developers at any time in case you experience any difficulties in using FALCON.m.

The following paragraphs give a very short overview of problems solved with FALCON.m.

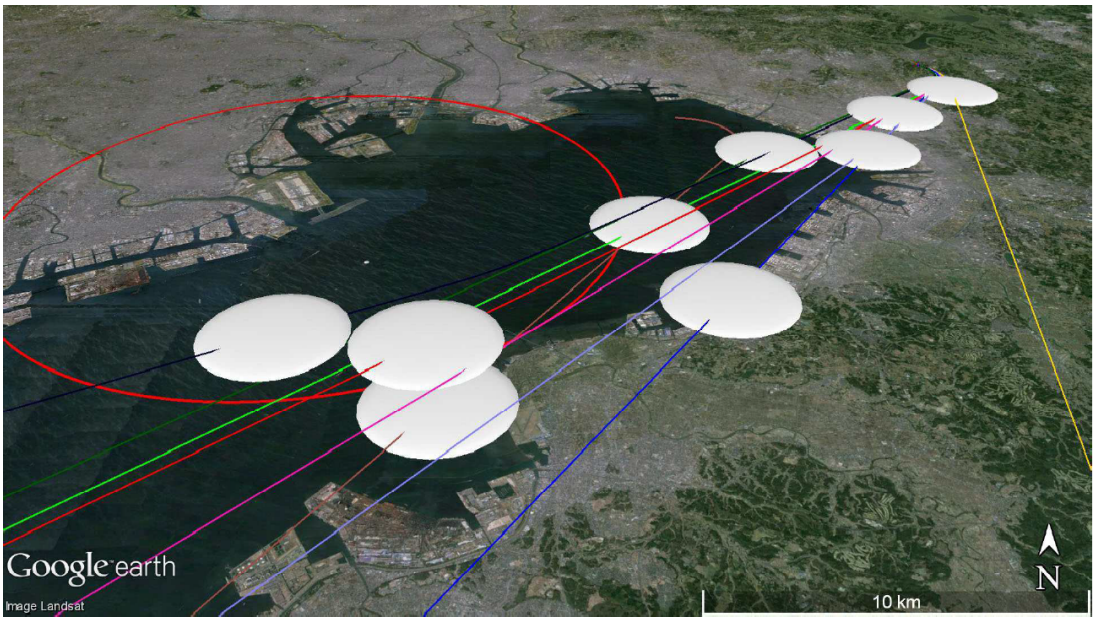


Figure 1: Sample optimal control problem solved with FALCON.m. Nine aircraft are approaching Tokyo International Airport in parallel. The white circles represent the separation limits to be kept between the different aircraft.

Figure 1 shows a visualization of a scenario in which nine aircraft are approaching Tokyo International Airport in parallel. The trajectories of all of these aircraft have been optimized for a given cost index, resulting in an optimal trade-off between arrival time and fuel burn. The aircraft behavior was modeled based on point mass equations of motion in three dimensional space, in the example including a variety of constraints that ensure a realistic behavior of the aircraft.

The round shapes in the figure represent the separation limits between the aircraft that needed to be fulfilled along the overall flight time for each individual pair of air-

craft. The overall problem contained a dynamic system consisting of nonlinear dynamics in 63 states and 27 controls. Anyway, FALCON.m allowed the solution of the problem due to its automatic sparsity generation and its automatic differentiation.

## 1.1 Basic version

Generally, the basic version of FALCON.m is provided to you free-of-charge. With this version, you can solve most of your real-life optimal control problems as it provides the following features:

- Solution of optimal control problems for autonomous, explicit, nonlinear, first order ordinary differential equations
- Analytic calculation of the Jacobian for cost-, constraint-, and dynamic model-functions
- Interfaces to different highly efficient nonlinear programming (NLP) solvers
- Implementation of different dynamic models, path constraints, and point constraints
- Essentially arbitrary number of states, controls, parameters, and constraints in the problem formulation
- Multi-phase optimal control formulations
- Trapezoidal Collocation and Backward Euler Full Discretization
- Post-processing and debugging features (Jacobian check, Simulation, Visualization (GUI), Scaling analysis)

## 1.2 Additional features and Add-ons

The trajectory optimization research group is actively working on extending the basic version of FALCON.m. The following add-ons have been or are currently developed and used for various research activities:

**Analytic Hessian** The analytic Hessian add-on contains all required functionalities to efficiently provide the second derivative information of the objective function and constraints to the NLP solver. This can significantly improve the convergence properties of the problem compared to only providing the Jacobian included in the basic version (especially close to the optimal point, e.g. for near-optimal initial guesses). Furthermore, the Hessian can be utilized to apply post-optimal sensitivity analyses to the solution of the problem.

**Post-optimal Sensitivities** Post-optimal sensitivity analysis provides an efficient way to approximate the optimal solution on a perturbed optimal control problem. This is helpful e.g. if a problem has to be solved for a nominal parameter value, as well as slightly changed values. Applying the implicit function theorem on the KKT-conditions of the NLP, the first derivative of the solution (optimal trajectory, controls) with respect to the disturbed parameter(s) can be determined analytically. For the objective function, even the second derivatives is available.

**Uncertainty Quantification** This add-on offers several approaches, including standard ones like Monte-Carlo analysis as well as sophisticated methods like generalized polynomial chaos, to obtain the distributions of optimal trajectories for given distributions of uncertain parameters. Furthermore, the output distributions can be accounted for in the optimization, e.g. by adding properties to the objective function, such as minimizing a standard deviation value, or to the constraints, as in: the optimal trajectory should respect this constraint with a confidence of two standard deviations. Furthermore, uncertainty quantification in the optimal control problem offers the possibility to evaluate “chance constraints”, i.e. constraints that must be fulfilled to a certain probability level. By this, safe and robust optimal trajectories can be calculated.

**System Identification** Based on the core framework of FALCON.m, a system identification tool was implemented which allows for estimating model parameters for experiment data of system outputs. The extension offers a set of common objective functions, such as least squares and maximum likelihood. Furthermore, the tool offers functionality to design the optimal input for system identification, yielding a persistent excitation with maximum information content on the system.

**Discrete Controls** While traditional optimal control software can find the optimal history of continuous controls, such as a steering angle or an elevator deflection, this extension treats discrete controls, such as a gear shift or a flap setting. An Outer Convexification is applied as relaxation to allow for an efficient solution process where tailored penalty approaches are implemented to restrict the control to the permissible discrete values.

**Trim Tool** The internal version of FALCON.m additionally features a software package for the computation of trim points for dynamic systems. These trim points can be used for example for defining stationary boundary conditions in trajectory optimization or parameter estimation problems. The trim tool exploits the efficient derivative evaluation in FALCON.m and enables the computation of trim states and controls for grids based on a generic trim template including essentially arbitrary constraints.

**Bi-Level/Distributed Optimization** This extension solves multi-optimization problems, i.e. an (upper level) optimization problem is treated which is dependent on the solution of one or more (lower level) optimization/optimal control problems. This is useful e.g. in the context of games or very large problems that can be deconstructed in

the sense of a primal decomposition (e.g. multiple aircraft approaching an airport). In particular, for the case of very large optimal control problems, e.g. when considering uncertainties, it is often possible to find unconnected subproblems which can be solved individually. By distributing these problems together, with suitable adaptations to the NLP solver parameters , the convergence time can be reduced significantly.

If you are interested in using these additional features in research projects with us, just contact us under falcon-m@tum.de!

# 2 Installation of FALCON.m

To install FALCON.m on your computer, just unzip the contents of the zip-archive you downloaded from `http://www.falcon-m.com` to a location where you want it to be stored and where you have read and write access to.

Figure 2 shows the namespace folder `+falcon`, containing the required FALCON.m files on a Windows machine. In the `vendor` folder additional external libraries required by FALCON.m can be found. The folder `examples` contains some example problems that help getting started. Depending on the version of FALCON.m you are using, additional files may exist in this folder.

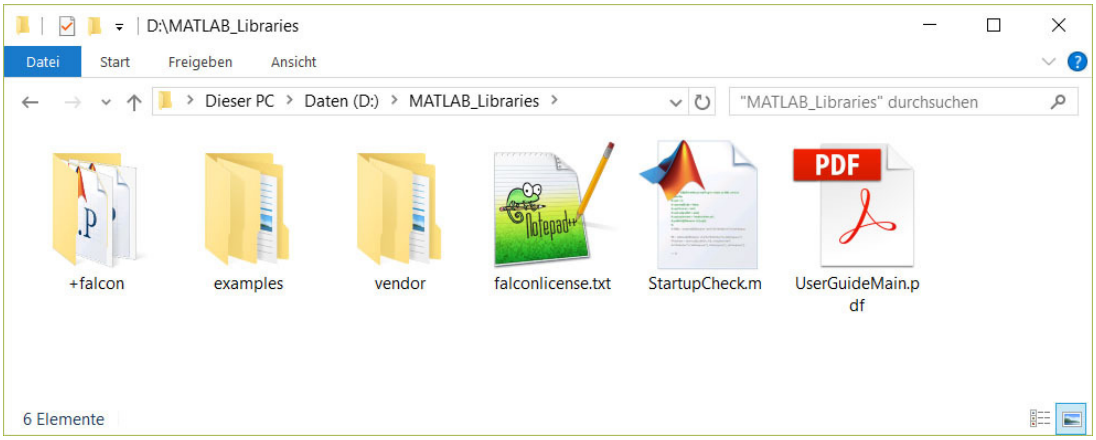


Figure 2: Content of the folder containing FALCON.m

Before solving the first optimal control problem, run `StartupCheck.m` to check if you are using a compatible version of MATLAB and all required external tools are available. Additionally, IPOPT will be automatically downloaded (in case you have a working internet connection) as the default solver for solving your optimal control problem.

In order to be able to use FALCON.m without running the startup check again every time, just add the folder containing the namespace folder `+falcon` to the MATLAB path. In the example above, the path to be added would be `D:\MATLAB_Libraries`. Do not add the folder with its subfolders but only the parent folder itself.

## 2.1 How to: Usage of IPOPT

IPOPT is an interior point NLP (nonlinear program) solver that is the common solver for FALCON.m. It is released under an open source EPL and is automatically downloaded (if desired by the user), when running the `StartUpCheck.m`.[1] In case you have your own IPOPT distribution (coded binaries), you can also just place them in the folder `..\vendor\ipopt`.

## 2.2 How to: Usage of SNOPT

SNOPT is a commercial NLP solver distributed by Stanford Business Software Inc. that can be used to solve NLP problems. The SNOPT license must be bought and the MATLAB interface must be obtained.[2]

In cases where users want to use SNOPT for solving their NLPs, FALCON.m provides an interface. In no way, FALCON.m distributes SNOPT: This means that the user must obtain the license for SNOPT on his/her own.[3]

When SNOPT should be used, please follow the instructions in the folder `..\vendor\snopt\UsageofSNOPT.txt`: This folder contains three already implemented m-files (`fun_generic.m`, `fun_generic2.m`, and `snopt_hdl.m`). These functions are necessary for the interfacing of SNOPT with FALCON.m. Thus, do not alter, rename, or change them in any way as the functionality of SNOPT with FALCON.m is no longer guaranteed.

Please put your obtained SNOPT MATLAB binaries in the folder `..\vendor\snopt\SNOPT` that you have to create. This folder must contain all mex-files (e.g., `snoptcmex.mexext`) as well as all SNOPT m-files (e.g., `snget.m`). In cases where the SNOPT interface is not working, please check that you have added all files appropriately and check the SNOPT error message.

After putting the binaries in the folder, FALCON.m will automatically detect the SNOPT binaries during its next run and you can use SNOPT as an alternative solver for your NLP. Therefore, you must only change the call to your solver to be:

```
solver = falcon.solver.snopt(problem)
```

FALCON.m is then using SNOPT for the optimization. Please note that the output structure for SNOPT differs from the one for IPOPT and therefore you might need to adapt your `solver.Solve` command if you experience errors.

**Remark**: FALCON.m was tested with the SNOPT mex interface from 18-June-2007. Compatibility is therefore only assured for this version as newer updates of the interface are not available to the FALCON.m team. You may find these in one of the initial commits of snopt-matlab github repository[4].

[1] https://github.com/coin-or/Ipopt
[2] http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm
[3] http://www.sbsi-sol-optimize.com/asp/sol_snopt.htm
[4] https://github.com/snopt/snopt-matlab/commits/master?after=b2222596b0d02347f9c3708ac7e6a8f727bc35bc+69

## 2.3 How to: Usage of FMINCON

If you have MATLAB's Optimization Toolbox installed and licensed, you are all set to use FMINCON[5]. You just need to change your solver setup to:

```
solver = falcon.solver.fmincon_algo(problem)
```

Once more, the output structure of the `solver.Solve` command differs and adaptations might be required. Additionally, it should be noted that FMINCON is inefficient compared to IPOPT and SNOPT and should only be used for very small problems.

# 3 Quick Start Guide

This section should help getting started with FALCON.m. It firstly introduces the problem class that can be solved using the tool. Afterwards, the most important basic ideas of the tool are listed, before example problems of increasing complexity are presented.

## 3.1 Optimal Control Problem Formulation

FALCON.m is able to solve optimal control problems of the following form:
Minimize the cost function

$$\min J(\mathbf{x}(t), \mathbf{u}(t), \mathbf{p}) \tag{1}$$

subject to a set of constraints, formed by the differential algebraic equation

$$\begin{bmatrix} \dot{\mathbf{x}}(t) \\ \mathbf{y}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \mathbf{p}) \\ \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t), \mathbf{p}) \end{bmatrix} \tag{2}$$

where $\mathbf{x}(t)$ specifies the states, $\dot{\mathbf{x}}(t)$ the state derivatives and $\mathbf{y}(t)$ additional model outputs. The states $\mathbf{x}(t)$, the controls $\mathbf{u}(t)$ and the parameters $\mathbf{p}$ are limited by a lower and an upper bound:

$$\mathbf{x}_{lb} \leq \mathbf{x}(t) \leq \mathbf{x}_{ub} \tag{3}$$
$$\mathbf{u}_{lb} \leq \mathbf{u}(t) \leq \mathbf{u}_{ub} \tag{4}$$
$$\mathbf{p}_{lb} \leq \mathbf{p} \leq \mathbf{p}_{ub} \tag{5}$$

The problem is considered on the time interval $[t_0, t_f]$ with each of the two either being fixed or free. In the formulation presented here, $t_0$ and $t_f$ are seen to be part of the parameter vector $\mathbf{p}$. Additionally, an arbitrary number of nonlinear constraints of the form

$$\mathbf{g}_{lb} \leq \mathbf{g}(\mathbf{y}, \mathbf{x}, \mathbf{u}, \mathbf{p}) \leq \mathbf{g}_{ub} \tag{6}$$

may be imposed. A special type of constraints appearing in many problems are initial and final boundary conditions specifying a start and an end state condition of the form

$$\mathbf{x}_{0,lb} \leq \mathbf{x}(t_0) \leq \mathbf{x}_{0,ub} \tag{7}$$
$$\mathbf{x}_{f,lb} \leq \mathbf{x}(t_f) \leq \mathbf{x}_{f,ub} \tag{8}$$

[5] https://mathworks.com/help/optim/ug/fmincon.html

For all constraints, equality conditions can be achieved by simply setting the upper and the lower limits to the same values.

$$\square_{lb} = \square_{ub} \tag{9}$$

*Remark:* A maximization of the cost function $\bar{J}$ can be achieved by simply choosing

$$J = -\bar{J}. \tag{10}$$

## 3.2 Important Basic Ideas of FALCON.m

- FALCON.m uses direct discretization methods in order to solve optimal control problems. The free variable is considered to be *time* throughout the implementation but may be chosen however suitable.

- For each value appearing in an optimal control problem, FALCON.m uses value definition objects, specifying the names, bounds and scaling of the values as appropriate. If required, these values are extended to grids over time inside FALCON.m. Examples for value definition objects are `falcon.State`, `falcon.Control`, `falcon.Constraint`,
`falcon.Cost`, `falcon.Parameter` and some more.

- FALCON.m allows the solution of multi phase optimal control problems, where each problem has to hold at least on `falcon.core.Phase`. Each phase holds a stategrid, one or more controlgrids, and a model. Phases may or may not be linked together.

- FALCON.m performs the optimization on a normalized time grid $\tau \in [0, 1]$ for every phase that is mapped to the real time grid by a linear transformation. Problems with variable final and/or initial time can be solved by choosing the initial or final time to be a free parameter.

- FALCON.m uses autonomous dynamics ("time-invariant") as default (the dynamic equations may not directly depend on the free variable, time). Anyway, non-autonomous dynamics can be tackled by introducing a new state $t$ with the dynamics

  $$\dot{t} = 1 \tag{11}$$

  that is added to the state vector of the problem. All other steps in creating the model, e.g., adding a final time parameter to the phase, remain the same. The collocation integrator should achieve that the final time in the state and the parameter are the same. If this is not the case an additional constraint may be added to the final point to assure that the times match.

- In order to achieve better numerical properties of a problem, FALCON.m internally scales all appearing values by a fixed scaling factor. The following relationship is used for scaling

  $$\square_{\text{scaled}} = \square_{\text{original}} \cdot M_{\text{scaling}} \tag{12}$$

It is recommended to scale all values to an order of magnitude of one, meaning that e.g. the scaling factor of a value expected to be about $10^5$ in the problem should be scaled by a scaling factor $M_{\text{scaling}} = 10^{-5}$.

- As FALCON.m uses gradient based optimization algorithms, initial guess values for everything to be optimized need to be available. In case no initial guess values are specified by the user, FALCON.m tries to create them itself.
- In order to solve an optimal control problem in FALCON.m four main steps are required:
  1. Define the model, constraint equations and problem structure in FALCON.m.
  2. Create the analytic derivatives of all appearing functions and create MATLAB executables (.mex files) from these functions.
  3. Prepare the problem itself for solution.
  4. Solve the problem using third party numerical optimization algorithms.

## 3.3 Introductory Example: Time Optimal Car Trajectory

In this subSection the trajectory from a given initial point to a given final point for a car model is optimized for minimum time. Consequently, the cost function is the final time:

$$\min J = t_f \tag{13}$$

The dynamic model of the car comprises the four states and the two controls listed in Table 1.

Table 1: States and controls of the car example problem.

| State | Description |
|---|---|
| $x$ | Position in $x$-direction |
| $y$ | Position in $y$-direction |
| $V$ | Absolute velocity of the car |
| $\chi$ | Course angle / direction of the velocity |

| Control | Description |
|---|---|
| $\dot{V}_{cmd}$ | The absolute acceleration of the car / gas pedal input |
| $\dot{\chi}_{cmd}$ | The angular velocity of the car / Steering wheel input |

The dynamics are given by:

$$\dot{x} = V \cdot \cos\chi \tag{14}$$
$$\dot{y} = V \cdot \sin\chi \tag{15}$$
$$\dot{V} = \dot{V}_{cmd} \tag{16}$$
$$\dot{\chi} = \dot{\chi}_{cmd} \tag{17}$$

Consequently, the combined state and control vectors are:

$$\mathbf{x} = \begin{bmatrix} x & y & V & \chi \end{bmatrix}^\intercal, \qquad \mathbf{u} = \begin{bmatrix} \dot{V}_{cmd} & \dot{\chi}_{cmd} \end{bmatrix}^\intercal \tag{18}$$

with the limits given by

$$\begin{aligned} 0 \leq \quad & x \leq 100 && (19)\\ 0 \leq \quad & y \leq 100 && (20)\\ 0 \leq \quad & V \leq 5 && (21)\\ -2 \cdot \pi \leq \quad & \chi \leq 2 \cdot \pi && (22)\\ -0.1 \leq & \dot{V}_{cmd} \leq 0.1 && (23)\\ -pi/8 \leq & \dot{\chi}_{cmd} \leq pi/8 && (24) \end{aligned}$$

The initial and final conditions for the trajectory are given as equality constraints by

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 5 \\ 0 \end{bmatrix}, \qquad \mathbf{x}_f = \begin{bmatrix} 100 \\ 100 \\ 5 \\ 0 \end{bmatrix} \tag{25}$$

Finally, there are also outputs added:

$$\begin{aligned} \dot{y} &= V \cdot \sin\chi && (26)\\ V_{p1} &= V + 1 && (27)\\ & && (28) \end{aligned}$$

These outputs are only used for debugging and not specifically constrained in the optimization problem. Thus, they merely show how outputs can be defined.

### 3.3.1 Implementation of Basic Problem in FALCON.m

In order to solve this optimal control problem in FALCON.m, first create a new MATLAB script (.m-file, in the following simply referred to as “the script”) to implement the required code in.

Before setting up the optimal control problem itself, it is recommended to setup the value definitions for the states, the controls and the required parameters appearing in the problem. The states are created using the constructor of the state class:
`falcon.State(Name,LowerBound,UpperBound,Scaling).`
Similarly controls and parameters can be created by:
`falcon.Control(Name,LowerBound,UpperBound,Scaling)`
`falcon.Parameter(Name,Value,LowerBound,UpperBound,Scaling)`
As states and controls will always appear on grids, no values can be specified whereas parameters are always considered to be scalar, directly holding one value.

For the considered example, the following code results:

```
%% Define states, controls and parameter
x_vec = [...
    falcon.State('x',       0,    100, 0.01);...
    falcon.State('y',       0,    100, 0.01);...
    falcon.State('V',       0,      5,    1);...
    falcon.State('chi', -2*pi,   2*pi,    1)];

u_vec = [...
    falcon.Control('Vdot'  , -0.1,  0.1, 1);...
    falcon.Control('chidot',-pi/8,+pi/8, 1)];

tf = falcon.Parameter('FinalTime', 20, 0, 40, 0.1);
```

The parameter used for the final simulation time is subject to optimization and uses a value of 20 as the initial guess for the optimization.

Next, the optimal control problem itself needs to be constructed, where the first step is to create a new `falcon.Problem` instance. This class is the main class of all FALCON.m optimal control problems and holds all relevant information. It is created by

```
%% Define optimal control problem
% Create new problem instance
problem = falcon.Problem('Car');
```

where `'Car'` specifies the name of the problem. The state dynamics of the problem will be discretized in time on a grid with 101 discretization points defined by `tau`:

```
% Specify discretization
tau = linspace(0,1,101);
```

Each problem defined in FALCON.m needs to have at least one phase. The following code creates a new phase and directly adds it to the problem:

```
% Add a new phase
phase = problem.addNewPhase(@source_car, x_vec, tau, 0, tf);
```

The input arguments to the `addNewPhase` command are the following:

1. `@source_car`: A MATLAB `function_handle` to the (not yet existing) model dynamics function
2. `x_vec`: The state vector of this phase
3. `tau`: The normalized time discretization for the states
4. `0`: The start time of the phase. In this case the start time is fixed to zero.
5. `tf`: The final time of the phase. In this case the `falcon.Parameter` `tf` created before is used. This way, the final time of the problem is subject to optimization within the bounds specified before.

Next, a control grid is added to the phase using the same discretization as for the states. Note that each phase in the problem may contain multiple control grids with different discretization grids.

```
% Add the control grid
phase.addNewControlGrid(u_vec, tau);
```

`u_vec` contains the definition of the control vector from before.

Afterward, the model outputs must be defined as well:

```
% Define model output
phase.Model.setModelOutputs([falcon.Output('yDOT');
    falcon.Output('Vp1')]);
```

The initial and final boundary conditions for the states are set using the two commands:

```
% Set the boundary conditions
% one column vector is short for equality bounds, i.e., lower bound ==
    upper bound
phase.setInitialBoundaries([0;0;5;0]);
phase.setFinalBoundaries([100;100;5;0]);
```

when only one vector of values is given, FALCON.m uses the numbers as lower and upper bounds, consequently leading to equality constraints. In case two vectors are given, they are used as lower (first column vector input) and upper (second column vector input) bounds, allowing to specify inequality constraints.

The final time `tf` shall be minimized in this example. This can easily be achieved by using the `addNewParameterCost` function of the problem:

```
% Add the cost function
problem.addNewParameterCost(tf);
```

Then, the problem must be prepared for the solution by invoking the following command:

```
% Prepare problem for solving
problem.Bake();
```

Now, the problem may be solved by adding the following command:

```
% Solve the problem
solver = falcon.solver.ipopt(problem);      % solver object
solver.Options.MajorIterLimit = 500;        % maximum number of
    iterations
solver.Options.MajorFeasTol   = 1e-5;       % feasibility tolerance
solver.Options.MajorOptTol    = 1e-5;       % optimality tolerance

solver.Solve();                             % solve command
```

In order to solve the problem, make sure that the FALCON.m folder has been added to your MATLAB path as described in Section 2 and run the script. Obviously, the dynamic model for the car has not yet been implemented and the optimization cannot be run. Anyway, FALCON.m will create the interface for the function holding the dynamic model after asking if it should do so. Answer the question with `y` in the MATLAB console and hit enter. Our script will throw some errors as the problem could not be solved, but FALCON.m created a MATLAB file named `source_car` in the current directory. The file should contain the following function interface:

```
function [states_dot, y] = source_car(states, controls)
% model interface created by falcon.m

% Extract states
```

```
x      = states(1);
y      = states(2);
V      = states(3);
chi    = states(4);

% Extract controls
Vdot   = controls(1);
chidot = controls(2);

% ------------------------ %
% implement the model here %
% ------------------------ %

% implement state derivatives here
x_dot   = ;
y_dot   = ;
V_dot   = ;
chi_dot = ;
states_dot = [x_dot; y_dot; V_dot; chi_dot];

% specify outputs
y =  [yDOT; Vp1];

end
```

Insert the model equations into the function by overwriting the lines:

```
% implement state derivatives here
x_dot   = V*cos(chi);
y_dot   = V*sin(chi);
V_dot   = Vdot;
chi_dot = chidot;
```

Furthermore, insert the output equations:

```
% specify outputs
y =  [V*sin(chi); V+1];
```

Afterward, rerun the previously created script containing the problem definition. While executing the first time, FALCON.m needs some time to create the analytic derivatives of the dynamic model before numerically solving the problem.

After the problem was solved, you can plot the results calling `problem.PlotGUI` (experimental) or by extracting the data from the problem in a custom plot function. In order to create a plot showing the states, the following code may be added to the problem definition script below `problem.Solve()`:

```
%% Plot
figure
for numState=1:4
    subplot(2,2,numState); grid on; hold on;
    xlabel('time');
    ylabel(phase.StateGrid.DataTypes(numState).Name);

    plot(phase.RealTime, phase.StateGrid.Values(numState,:), 'x-');
end
```

Figure 3 shows the results of the problem using slightly different plot commands.

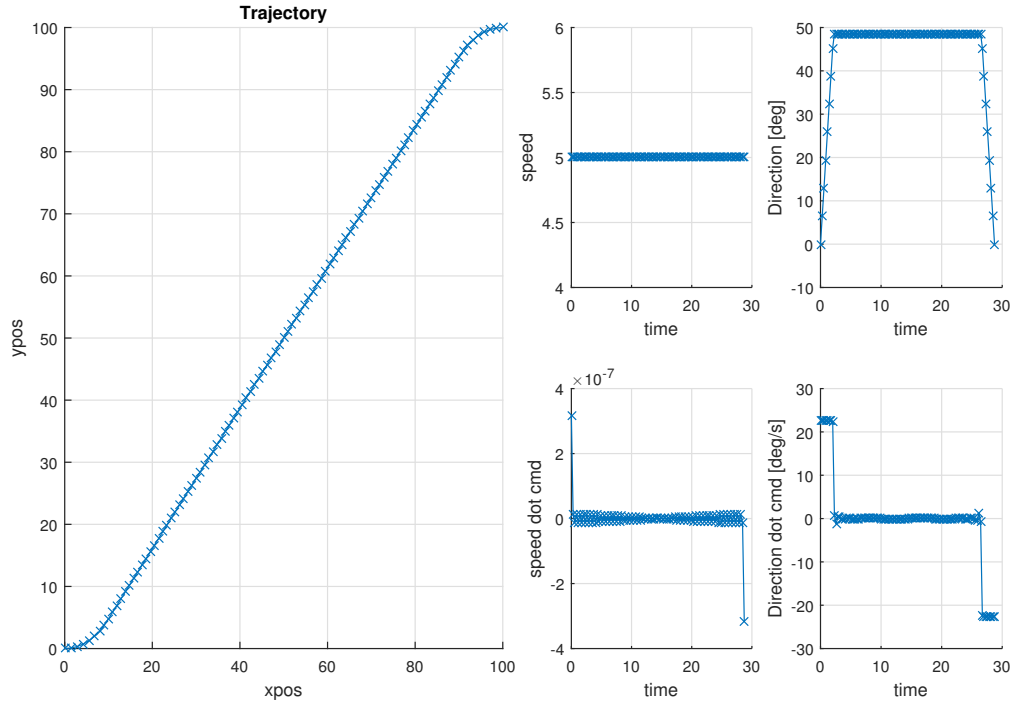


Figure 3: Results for the time minimal car trajectory

### 3.3.2 Adding a Post-Processing Step

Currently, the model outputs are calculated directly inside the model, although they are actually not required in the optimization procedure (which means they are not constrained). This means that significant computational overhead is introduced as not only the time history of the values but also their derivatives are calculated in each iteration of the optimization problem.

To reduce the computational effort required for calculating such “debugging” variables, FALCON.m overs a feature called “post-processing”. By this feature, debugging calculations can be conducted automatically after the actual solution of the optimization problem. To use it the following lines may be added before preparing the problem for the solution:

```
30 % apply post-processing to each phase
31 % problem.addPostProcessingStep(function_handle, state/control/output
32 % objects, debug value object(s))
33 problem.addPostProcessingStep(@(x) x+1, {x_vec(3)},
       falcon.Value('Vp1'));
34 problem.addPostProcessingStep(@postProcessFcn, {x_vec(4),x_vec(3)},
       [falcon.Value('yDOT'),falcon.Value('V_square')]);
35
36 % Prepare problem for solving
37 problem.Bake();
```

Here, `postProcessFcn` is a simple function calculating two output/debug variables:

```
function [y_dot,V_square] = postProcessFcn(chi,V)
% ------------------------ %
% implement the post-processing step here %
% ------------------------ %

% implement post-processing value
y_dot    = V.*sin(chi);
V_square = V.^2;

% EoF
end
```

It is important to note that each post-processing function must be capable of element-wise

This will automatically calculate the desired debugging outputs after the solution and thus, without computational overhead during the optimization. Still, the values are saved in a grid and can be accessed both through the `problem.PlotGUI` as well as the phase object.

It should be noted that the introduced commands add a post-processing evaluation to each phase of the problem automatically. If it is only desired to have post-processing in specific phases of the problem, the command `phase.addPostProcessingStep` can be used. It is called with the same input structure like for the problem but only for specific phases.

It should be noted that older FALCON.m versions allow to add post-processing steps after the problem solution. While this is generally still possible in the current version as well, the behavior is deprecated and will be removed in a future release. Thus, the version to add the steps before the problem solution should be used.

### 3.3.3 Implementation of Path Constraints

Next, the rate of turn of the car should be limited depending on the velocity of the car:

$$-\frac{1}{2 \cdot V} \leq \dot{\chi}_{cmd} \leq \frac{1}{2 \cdot V} \tag{29}$$

This constraint should be active along the whole path of the car and is therefore called *path constraint*. This constraint will be implemented using the two inequality constraints:

$$c_{lb} = -\frac{1}{2 \cdot V} - \dot{\chi}_{cmd} \leq 0 \tag{30}$$

$$c_{ub} = \dot{\chi}_{cmd} - \frac{1}{2 \cdot V} \leq 0 \tag{31}$$

Add the following code in the script somewhere before `problem.Solve`.

```
% Path Constraint
pathconstraints = [...
falcon.Constraint('turnlb', -inf, 0);...
falcon.Constraint('turnub', -inf, 0)];
phase.addNewPathConstraint(@source_path, pathconstraints,tau);
```

In this code, first two `falcon.Constraint` objects are created specifying the names and the lower and upper bounds of the newly added constraints. Afterwards, the path constraint function is added to the phase of the problem. The input arguments to `addNewPathConstraint` are:

1. `@source_path`: A MATLAB `function_handle` to the (not yet existing) path function.
2. `pathconstraints`: The objects defining the outputs of the path function including their limits.
3. `tau`: The normalized grid to evaluate the path constraint function on. In this case the same grid as for the states and controls is selected.

Run the script again and let FALCON.m create the function interface for you. After FALCON.m created the interface file and you implemented equations (30) and (31), the resulting path constraint function should look the following:

```
function [constraints] = source_path(states, controls)
% constraint interface created by falcon.m

% Extract states
x      = states(1);
y      = states(2);
V      = states(3);
chi    = states(4);

% Extract controls
Vdot   = controls(1);
chidot = controls(2);

% ---------------------------- %
% implement the constraint here %
% ---------------------------- %

% implement constraint values here
turnlb = -0.5/V - chidot;
turnub = chidot - 0.5/V;
constraints = [turnlb; turnub];

end
```

Now, the problem can be solved again, resulting in histories as displayed in Figure 4. Feel free to play with the example and to learn more about FALCON.m. The next example should give a more detailed overview of the features of FALCON.m and how to use them.

### 3.3.4 Using the Path Constraint Builder

We will now examine the case where we do not use the automatic function creation method provided by FALCON.m, but build the path constraint “by hand”.
In order to do this add the code

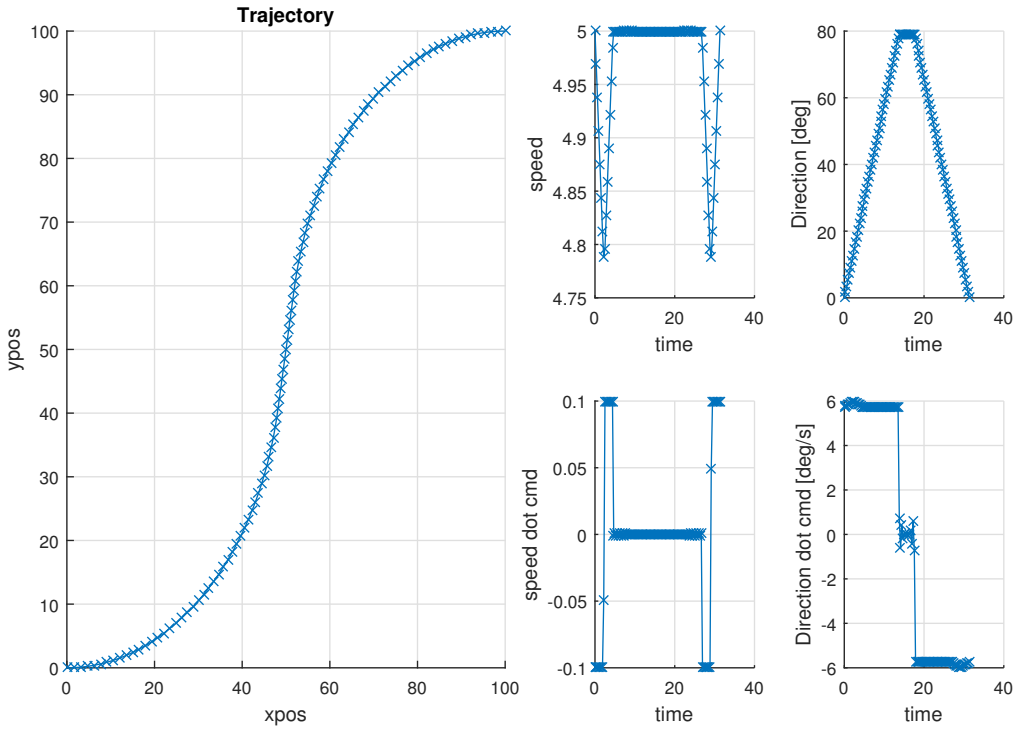


Figure 4: Results for the time minimal car trajectory fulfilling the path turn rate limits

```
pconMdl = falcon.PathConstraintBuilder('CarPCon', [], x_vec(3),
    u_vec(2), [], @source_path_reduced);
pconMdl.Build();
```

to the script.

You should observe that we only input the required values for the path constraint to be evaluated, i.e., the velocity state and the course derivative control. This is done by adding the required data types at the appropriate positions in the path constraint builder environment that has the general interface:

```
pathCon = falcon.PathConstraintBuilder('Name',Outputs,States,
Controls,Parameters,Handle)
```

Adding only the required inputs can significantly speed up the build and evaluation procedure as the number of necessary symbolic differentiations and not required evaluations is significantly reduced.

Now, you can copy the original path constraint function and change it to the following form such that it suits the path constraint builder interface:

```
function [constraints] = source_path_reduced(V, chidot)

% ---------------------------- %
% implement the constraint here %
% ---------------------------- %

% implement constraint values here
turnlb = -0.5/V - chidot;
```

```
turnub = chidot - 0.5/V;
constraints = [turnlb; turnub];

end
```

Finally, you must exchange the call to the added path constraint in the script

```
phase.addNewPathConstraint(@CarPCon, pathconstraints, tau);
```

Again, we add the path constraint to the standard discretized time grid.

Run the optimization: The results should be the same, but you should see some differences in how the `Bake` is conducted and in the convergence time.

It should be noted that parameters, in cases where you have these entering the model of the path constraint, must be added to the path constraint object manually. Fir this purpose, the following command structure can be used:

```
pathConObj = phase.addNewPathConstraint(@CarPCon, pathconstraints, tau);
pathConObj.setParameters(Parameters);
```

This gives the possibility to only add the required parameters to the path constraint. Take into account that an error is issued if you have parameters in the model but not in the constraint object and vice versa. Note that this behavior is the same as when adding a model with parameters to a phase, where you also have to manually specify the parameters (and outputs) in the `Phase` object.

### 3.3.5 Simple Multi-phase Problem

In order to use the car model within a multi-phase environment, we first of all must define additional time parameters:

```
%% Define parameter
tfint = falcon.Parameter('IntermediateTime', 20, 0, 40, 0.1);       %
    Intermediate time = final time of phase 1
tf = falcon.Parameter('FinalTime', 40, 0, 80, 0.1);     % Final time =
    final time of phase 2
```

Furthermore, it is necessary to update the state boundaries such that the second phase is feasible:

```
%% Define States
x_vec = [...
    falcon.State('x',       0,    200, 0.01);...
    falcon.State('y',       0,    200, 0.01);...
    falcon.State('V',       0,      5,    1);...
    falcon.State('chi', -2*pi,   2*pi,    1)];
```

The next thing is to adapt the already defined first phase, which now ends at the intermediate time:

```
% Change the first phase
phase = problem.addNewPhase(@source_car, x_vec, tau, 0, tfint);
```

Afterwards, we must define a second phase as follows:

```
% Add a second phase Phase
phase2 = problem.addNewPhase(@source_car, x_vec, tau, tfint, tf);
phase2.addNewControlGrid(u_vec, tau);
```

```
phase2.Model.setModelOutputs([falcon.Output('dummyout');
    falcon.Output('dummy1')]);

% Set final Boundary Condition
phase2.setFinalBoundaries([200;0;5;0]);

% Path Constraint
pathconstraints = [...
    falcon.Constraint('turnlb', -inf, 0);...
    falcon.Constraint('turnub', -inf, 0)];
phase2.addNewPathConstraint(@source_path, pathconstraints, tau);
```

Note that the second phase now starts at the intermediate time and ends with the final time. Additionally, the second phase only has a final boundary condition defined. Therefore, the command

```
% Connect the phases
problem.ConnectAllPhases();
```

must be used such that the first and the second phase are connected and we get a smooth trajectory. You are now able to solve a multi-phase problem.

### 3.3.6 Multi-phase Problem using Pointconstraint Builder

In Section 3.3.5, we use the `problem.ConnectAllPhases();` command to automatically connect the phases and define an appropriate point constraint. In this example, we will work with the pointconstraint builder directly to show that the procedure is the same and the builder can connect two arbitrary phases at two arbitrary time points.

Therefore, remove the `problem.ConnectAllPhases();` command and instead implement the point constraint as follows:

```
% Connect the phases by the point constraint builder
pconObj = falcon.PointConstraintBuilder('ConnectPhases');

% Add the phase inputs (i.e., the states) that are required for one time
% step in each phase
pconObj.addPhaseInput(0,x_vec,0,1);     % This is the first phase
pconObj.addPhaseInput(0,x_vec,0,1);     % This is the second phase
    (that could also have another input structure)

% Now use an anonymous function handle to build the constraint
% Remember that the phase inputs are automatically split up into grids
% (_g*)
pconObj.addSubsystem(@(x,y) x - y,...
    {'states_g1', 'states_g2'},...
    {'phaseDefect'});

% Split the vector in single constraints
pconObj.SplitVariable('phaseDefect',{'x_PhaseDefect';
    'y_PhaseDefect'; 'V_PhaseDefect';'chi_PhaseDefect'});

% Constraint value names
pconObj.setConstraintValueNames({'x_PhaseDefect';
    'y_PhaseDefect';'V_PhaseDefect';'chi_PhaseDefect'});
```

```
23
24 % Build the constraint
25 pconObj.Build;
```

After building the point constraint, we have to include the created point constraint model in the optimal control problem. Therefore, the following lines of code must be added:

```
1  % Define the constraint data types (all must be zero)
2  connectConstraints = [falcon.Constraint('x_PhaseDefect',0,0,1,0,true);
3      falcon.Constraint('y_PhaseDefect',0,0,1,0,true);
4      falcon.Constraint('V_PhaseDefect',0,0,1,0,true);
5      falcon.Constraint('chi_PhaseDefect',0,0,1,0,true)];
6
7  % Add the point constraint to the problem
8  % Within the first phase the last normalized time step (tau = 1) must be
9  % added, while we have the first normalized time step (tau = 0) in the
10 % second phase
11 problem.addNewPointConstraint(@ConnectPhases,connectConstraints,...
12     problem.Phases(1),1,problem.Phases(2),0);
```

Overall, the problem can now be solved again, the results are hopefully the same as before and you should actually also see a very similar convergence behavior.

Again, in case you want to connect phases in the beginning and end, it is much easier to just use the `problem.ConnectAllPhases();` command instead of the point constraint builder. But in cases, where you want to connect different intermediate time points (e.g., for symmetry or periodicity), the point constraint builder provides you with a viable option.

Take into account that parameters, in cases where you have these entering the model of the point constraint, must be added to the point constraint object manually once more. This is done as with the path constraint case.

## 3.4 Full Example: Optimal Aircraft Trajectories

In the following sections, an aircraft related optimal control problem will be presented. Starting with a pretty simple model and no additional constraints, the problem will step by step be extended to show a large part of the feature set of FALCON.m.

### 3.4.1 2-D Kinematic Aircraft Approach

First, a simple aircraft model approaching an airport will be considered, minimizing the thrust applied during the flight. The dynamic model contains four states and is controlled by two controls as listed in Table 2.

Table 2: States and controls of a first, simple, kinematic aircraft model

| State | Description | Lower limit | Upper limit |
|---|---|---|---|
| $x$ | Lateral position of the aircraft | $-\inf$ | $\inf$ |
| $z$ | Vertical position of the aircraft | $-12000\,m$ | $-304\,m$ |
| $V$ | Absolute velocity of the aircraft | $60\,\frac{m}{s}$ | $300\,\frac{m}{s}$ |
| $\gamma$ | Climb angle | $-0.15\,rad$ | $0.15\,rad$ |

| Control | Description | Lower limit | Upper limit |
|---|---|---|---|
| $T$ | Thrust force | $0\,N$ | $200\,kN$ |
| $C_L$ | Lift coefficient | $-0.5$ | $1.5$ |

The model equations used are

$$\dot{x} = V \cdot \cos\gamma \tag{32}$$

$$\dot{z} = -V \cdot \sin\gamma \tag{33}$$

$$\dot{V} = \frac{1}{m} \cdot \left(T - \left(\frac{\rho}{2} \cdot V^2 \cdot S \cdot (C_{D0} + k \cdot C_L^2)\right)\right) - g\sin\gamma \tag{34}$$

$$\dot{\gamma} = \frac{1}{m \cdot V} \cdot \left(\frac{\rho}{2} \cdot V^2 \cdot S \cdot C_L\right) - \frac{g}{V}\cos\gamma \tag{35}$$

assuming the aircraft mass $m = 55000\,kg$, the gravitational acceleration $g = 9.81\,\frac{m}{s}$, the air density $\rho = 1.225\,\frac{kg}{m^3}$, and the wing reference area $S = 123\,m^2$. The aerodynamic drag is quantified by $C_D0 = 0.03$ and $k = 0.04$. Similarly to the car example in section 3.3, first the states, controls and the final time parameter are defined:

```
% Create the states and controls
states = [falcon.State('x',     -inf,  inf, 1e-3);
          falcon.State('z',    -12e3, -304, 1e-3);
          falcon.State('V',       60,  200, 1e-2);
          falcon.State('gamma',-0.15, 0.15, 1)];

controls = [falcon.Control('T',      0, 2e5, 1e-5);
            falcon.Control('C_L', -0.5, 1.5, 1)];

% Create the final time parameter
tf = falcon.Parameter('Final_Time', 1000, 20, 4e3, 1e-3);
```

Afterwards, the problem is created, the normalized time discretization is set up and the required phase, including its control grid is added to the problem:

```
%% Create the problem
problem = falcon.Problem('AC_Approach');

% Specify Discretization
tau = linspace(0,1,1001);

% Add a new Phase
phase = problem.addNewPhase(@source_aircraft, states, tau, 0, tf);
phase.addNewControlGrid(controls, tau);
```

In this example, the initial boundary conditions are fixed to the following values, while the final condition may very between the given boundaries:

```
21 % Set Boundary Condition
22 phase.setInitialBoundaries([-250e3; -10e3; 200; 0]);
23 phase.setFinalBoundaries([0; -304; 80; -0.05], [0; -304; 100; 0.05]);
```

As defined by the boundary conditions, the problem describes a descent to a specific point, e.g. the touchdown point on the runway. A relevant objective function is the fuel consumption of the aircraft during this phase, which can be approximated by the thrust applied over the entire time. To this end, we introduce a control cost, i.e. the sum of squares of the selected control at every time point is penalized:

```
24 % Add Cost Function
25 problem.addNewControlCost(controls(1));
```

After solving the problem with

```
26 % Solve Problem
27 problem.Solve();
```

the results can be displayed and analyzed using the plot GUI of FALCON.m by simply calling

```
28 %% Plot
29 problem.PlotGUI;
```

Note, that again when running the above listed code the first time, the problem may not be solved, as the model dynamics added in line 19 do not yet exist. Let FALCON.m create the required function interface for you by selecting `y` when asked. Put the following model dynamics into the function file:

```
1  % Constants
2  m = 55e3; % kg
3  rho = 1.225; % kg/m^3
4  S = 123; % m^2
5
6  % Calculate drag
7  C_D = 0.03 + 0.04 * C_L^2;
8  D = rho/2 * V^2 * S * C_D;
9
10 % implement state derivatives here
11 x_dot     =  V * cos(gamma);
12 z_dot     = -V * sin(gamma);
13 V_dot     = 1/m * (T-D) - g*sin(gamma);
14 gamma_dot = 1/m * rho/2 * V * S * C_L - g/V*cos(gamma);
15
16 % Combine state_dot values
17 states_dot = [x_dot; z_dot; V_dot; gamma_dot];
```

Now, you should be able to solve the problem by running the script again. The plot GUI may now be used to create plots similar to those in Figure 5.

### 3.4.2 3-D Point Mass Aircraft Approach

Now we extend the two dimensional model from the previous example to a three dimensional aircraft model. The goal of the following example is to show the more advanced

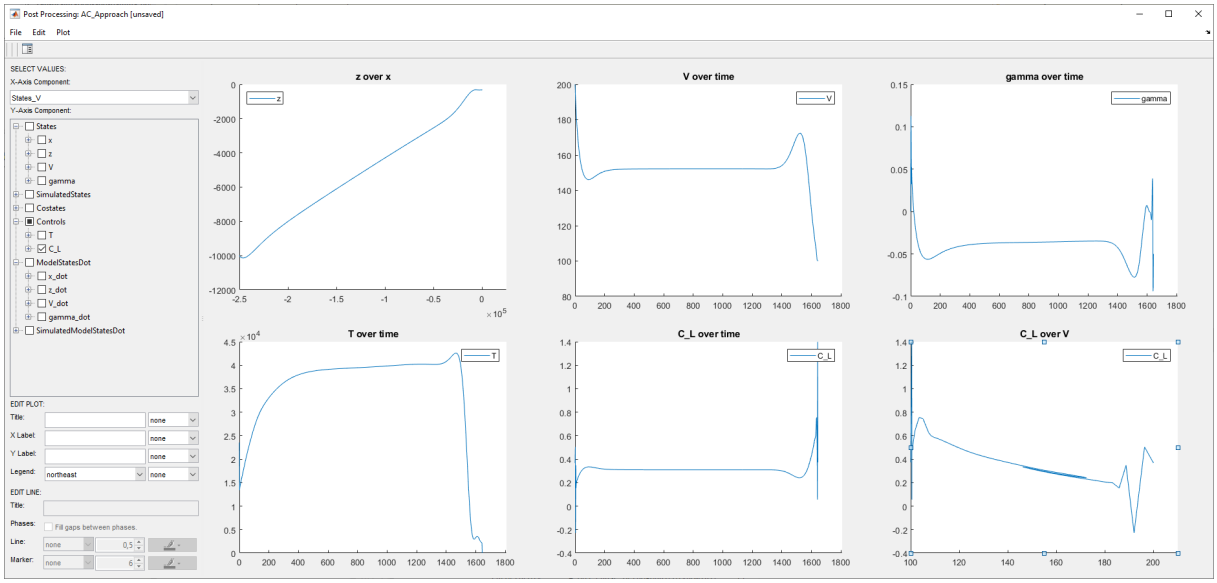


Figure 5: Results of the first simple aircraft trajectory optimization problem.

capabilities in FALCON.m regarding modeling and solving optimal control problems. Please note that we chose a dynamic model with a relatively low order of complexity to keep the focus on the implementation in FALCON.m rather than blurring the important aspects by complex physical relationships. The three dimensional model now has a total number of six states by adding the equations for the lateral position $y$ and the lateral dynamics of the course angle $\chi$:

$$\dot{x} = V \cdot \cos\chi \cdot \cos\gamma \tag{36}$$

$$\dot{y} = V \cdot \sin\chi \cdot \cos\gamma \tag{37}$$

$$\dot{z} = -V \cdot \sin\gamma \tag{38}$$

$$\dot{V} = \frac{1}{m} \cdot (T - D - W\sin\gamma) \tag{39}$$

$$\dot{\chi} = \frac{1}{m \cdot V \cdot \cos\gamma} \cdot L \cdot \sin\mu \tag{40}$$

$$\dot{\gamma} = \frac{1}{m \cdot V} \cdot (L - W\cos\gamma) \tag{41}$$

with the lift force $L$ the drag $D$ and the weight $W$, defined as:

$$L = \frac{\rho}{2} \cdot V^2 \cdot S \cdot C_L \tag{42}$$

$$D = \frac{\rho}{2} \cdot V^2 \cdot S \cdot C_D\,(C_L) \tag{43}$$

$$W = m \cdot g \tag{44}$$

Additionally to the control variables we used in the previous example, namely the thrust force $T$ and the lift coefficient $C_L$, we add the bank angle $\mu$ to control the plane in which the lift force acts on the aircraft.

Please note that the way the drag coefficient is calculated also differs from the 2D case. For this example we want to assume that $C_D$ is given as tabular data as some function of the lift coefficient $C_L$ to show how to implement functions that are not analytically differentiable.
Moreover we now want the model to return the value for the lift $L$. This output will be used later to introduce a pathconstraint for the scalar load factor in the $z$-direction $n_z$.

The modeling philosophy in FALCON.m follows a subsystem based approach. This means that the basic components of our dynamic model are encapsulated in MATLAB functions with user defined inputs and outputs. These inputs and outputs may be scalar or vectorized values that, for the ease of implentation, may be combined or split to obtain new variables. The different types of variables that can be defined for modeling the state space dynamics $[y, \dot{x}] = f(x, u, p)$ in the modeling part of FALCON.m are:

- States $x$
- Controls $y$
- Parameters $p$

and to define the return arguments of the model:

- Outputs $y$
- State-Derivatives $\dot{x}$

Aditionally one can introduce constant values $c$ that are fix and may not change during the optimization. Please be aware that the difference between the parameters $p$ and the constant values $c$ is that the parameters $p$ are optimizable (if not fixed in a later modeling step) and the constant values are specified only when the model is built and can not be changed after compiling the model.
To implement the model lets first define the states, the controls and parameters as arrays of falcon.State, falcon.Control and falcon.Parameter objects:

```
% Create the states and controls
states = [falcon.State('x',     -inf,  inf, 1e-3);
falcon.State('y',    -inf,   inf, 1e-3);
falcon.State('z',    -12e3,    0, 1e-3);
falcon.State('V',       60,  300, 1e-2);
falcon.State('chi',-inf, inf, 1)
falcon.State('gamma',-0.15, 0.15, 1)];

controls = [falcon.Control('T',   0, 2e5, 1e-5);
falcon.Control('C_L', 0,   1, 1);
falcon.Control('mue', -pi/4,   pi/4, 1)];

parameters = [falcon.Parameter('m',   55e3);
```

To set the lift force $L$ as model output $y$ we have to define a falcon.Constraint-object and hand it over to our subsystem derivative instance by using the method FALCON-model.addOutputs:

```
outputs = falcon.Constraint('L',   0, 0, 80e3);
falcon_model.addOutputs(outputs);
```

# 4 Theoretical Fundamentals

Matlab class library Numerical collocation Uses third party numerical optimization algorithms Uses mex and C files for maximum performance

## 4.1 Optimal Control Problem

When using FALCON.m, optimal control problems of the form presented in section 3.1 are to be solved.

## 4.2 Collocation

[1] Optimization parameter vector $\mathbf{z}$ is built up from

$$\mathbf{z} = \begin{pmatrix} \mathbf{x}_0 & \mathbf{x}_1 & \dots & \mathbf{x}_K & \mathbf{u}_0 & \dots & \mathbf{u}_N & \mathbf{p} \end{pmatrix}^\mathsf{T} \tag{45}$$

Integration defect

$$\mathbf{c}_k(\mathbf{z}) = \mathbf{x}_k(\mathbf{z}) - \mathbf{x}_{k+1}(\mathbf{z}) + h_k \cdot \Phi(\mathbf{x}_k(\mathbf{z}), \mathbf{x}_{k+1}(\mathbf{z}), \mathbf{u}_k(\mathbf{z}), \mathbf{u}_{k+1}(\mathbf{z}), \mathbf{p}(\mathbf{z})) \stackrel{!}{=} 0 \tag{46}$$

for general integration scheme $\Phi(\mathbf{x}_k(\mathbf{z}), \mathbf{x}_{k+1}(\mathbf{z}), \mathbf{u}_k(\mathbf{z}), \mathbf{u}_{k+1}(\mathbf{z}), \mathbf{p}(\mathbf{z}))$.
Examples:
Euler backward collocation

$$\mathbf{x}_k(\mathbf{z}) - \mathbf{x}_{k+1}(\mathbf{z}) + h_k \cdot \mathbf{f}(\mathbf{x}_{k+1}(\mathbf{z}), \mathbf{u}_{k+1}(\mathbf{z}), \mathbf{p}(\mathbf{z})) \stackrel{!}{=} 0. \tag{47}$$

Trapezoidal collocation

$$\mathbf{x}_k(\mathbf{z}) - \mathbf{x}_{k+1}(\mathbf{z}) + \frac{h_k}{2} \cdot (\mathbf{f}(\mathbf{x}_k(\mathbf{z}), \mathbf{u}_k(\mathbf{z}), \mathbf{p}(\mathbf{z})) + \mathbf{f}(\mathbf{x}_{k+1}(\mathbf{z}), \mathbf{u}_{k+1}(\mathbf{z}), \mathbf{p}(\mathbf{z}))) \stackrel{!}{=} 0. \tag{48}$$

Resulting Constraint vector

$$\mathbf{C}(\mathbf{z}) = \begin{pmatrix} \mathbf{x}_0(\mathbf{z}) \\ \mathbf{x}_f(\mathbf{z}) \\ \mathbf{x}_k(\mathbf{z}) - \mathbf{x}_{k+1}(\mathbf{z}) + h_k \cdot \Phi(\mathbf{x}_k(\mathbf{z}), \mathbf{x}_{k+1}(\mathbf{z}), \mathbf{u}_k(\mathbf{z}), \mathbf{u}_{k+1}(\mathbf{z}), \mathbf{p}(\mathbf{z})) \\ \mathbf{C}(\mathbf{x}(\mathbf{z}), \mathbf{u}(\mathbf{z}), \mathbf{p}(\mathbf{z}), t(\mathbf{z})) \end{pmatrix} \tag{49}$$

### 4.2.1 Time Transformation

linear time transformation Normalized time $\tau \in [0, 1]$ $t_0$ and $t_f$: initial and final time linear transformation

$$t = t_0 + (t_f - t_0) \cdot \tau. \tag{50}$$

transformed time derivative

$$\frac{\mathrm{d}t}{\mathrm{d}\tau} = (t_f - t_0) \tag{51}$$

transformed state dynamics

$$\frac{\mathrm{d}\mathbf{x}(t(\tau))}{\mathrm{d}\tau} = \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} \cdot \frac{\mathrm{d}t}{\mathrm{d}\tau} = \mathbf{f}\left(\mathbf{x}(t(\tau)), \mathbf{u}(t(\tau)), \mathbf{p}, t(\tau)\right) \cdot (t_f - t_0). \tag{52}$$

See e.g. [3, 2]

### 4.3 Numerical Optimization

FALCON.m does not provide own numerical optimization algorithms but features a general purpose interface that may be adapted to be used with *IPOPT*, *SNOPT*, and *FMINCON*.

## 5 Problem Structure Used in FALCON.m

FALCON.m represents each optimal control problem by a set of MATLAB classes. This algorithmic model of the problem allows for great flexibility while maintaining high computational performance.

### 5.1 Optimization Problem Structure

The problem structure use by FALCON.m is introduced in Figure 6. The next subsections are going to introduce the definition of these structure elements

### 5.2 Command Line Interface

FALCON.m also provides a command line interface (CLI) that can be used to directly define FALCON.m problems using the MATLAB command line. The important command are illustrated in Figure 7.

The red boxed command is the main command for creating a new project: Here all user defined variables and the general workflow of the FALCON.m problem statement are defined in separate files. Thus, the user must only concentrate on setting correct boundaries and implementing the model rather than on having to deal with basic formatting and workflow. This behavior is also illustrated in Figure 8.

### 5.3 falcon.Problem

Parent Classes: falcon.core.HasToStruct, falcon.core.Handle, falcon.core.HasName

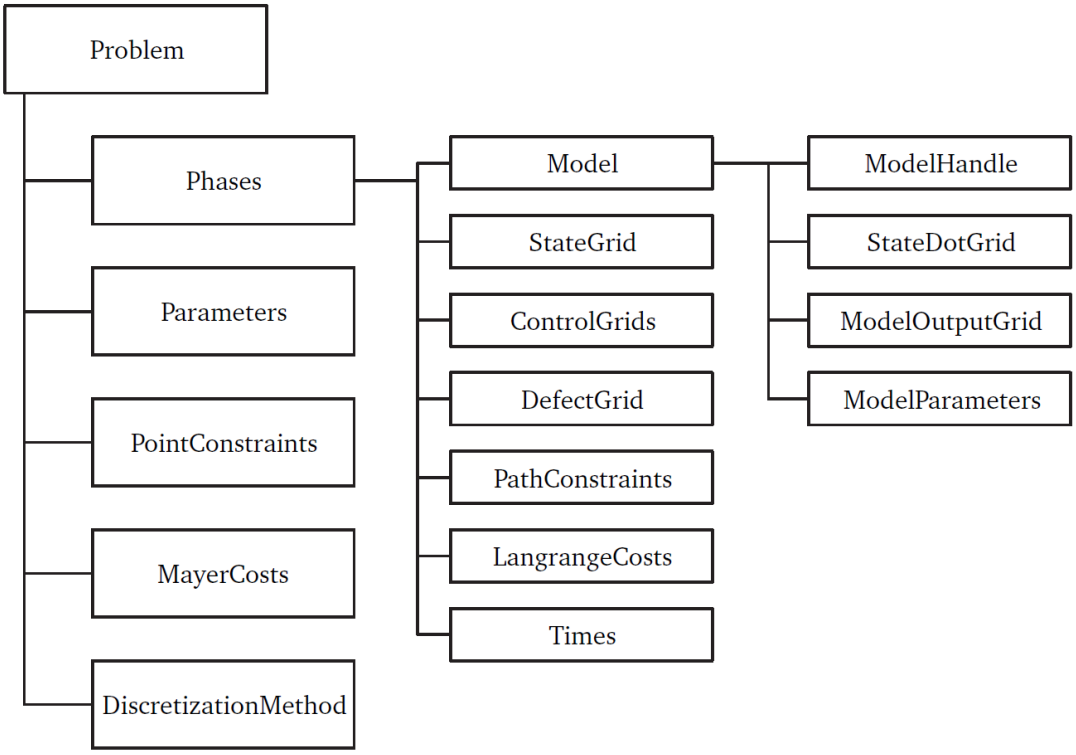


Figure 6: Structure of a problem represented in FALCON.m.

- `> falcon check`: performs initial checks on `falcon` setup
- `> falcon help`: displays the help for the specified command
- `> falcon license`: displays the license agreement
- `> falcon list`: lists available console interface commands
- `> falcon make:constraint`: create a constraint function template
- `> falcon make:cost`: create a cost function template
- `> falcon make:model`: create a model function template
- `> falcon make:system`: create a subsystem function template
- `> falcon make:test`: create a unit test interface
- `> falcon new`: create a template for a new project
- `> falcon prefer`: set some global preferences for `FALCON.m` (e.g., standard solver)
- `> falcon version`: displays the version and release date
- Help can be called using the `'-h'` or `'--help'` argument as an input

Figure 7: Important commands of FALCON.m command line interface.

### Properties

**+ `Phases`** (read-only)
Array keeping all falcon.core.Phases of this problem.

```
>> falcon new myproject -x vel_abs_K chi_K gamma_K -u F_P
Created file myproject\main.m
Created file myproject\steps\allvariables.m
Created file myproject\steps\buildmodel.m
Created file myproject\steps\createproblem.m
Created file myproject\steps\solveproblem.m
Created file myproject\steps\viewsolution.m
Created file myproject\models\mainmodel.m
Created file myproject\vars\states.m
Created file myproject\vars\outputs.m
Created file myproject\vars\controls.m
Created file myproject\vars\parameters.m
Created file myproject\models\lagrangecost.m
```

Figure 8: Example command to create a complete FALCON.m project using the command line interface.

**+ `Parameters`** (read-only)
Array keeping all falcon.Parameters of this problem.

**+ `PointConstraintFunctions`** (read-only)
Array keeping all falcon.core.PointFunction objects of this problem.

**+ `MayerCostFunctions`** (read-only)
Array keeping all falcon.core.PointFunction objects used as Mayer cost functions of this problem.

**+ `isCrunchy`** (read-only)
Determines if the optimization problem has already been baked. If true, the problem can no longer be altered. Use falcon.Problem.UnBake to make it editable again.

**+ `DiscretizationMethod`** (read-only)
The falcon.discretization.DiscretizationMethod method used to discretize this optimal control problem. Use falcon.Problem.setDiscretizationMethod to set this property. Available discretization methods can be found in falcon.discretization.'name'. (default = falcon.discretization.Trapezoidal)

**+ `CostScaling`** (read-only, Default = 1)
The numeric scaling factor for the sum of all cost functions.

**+ `UseHessian`** (read-only)
Flag that determines if the analytic Hessian (2nd Derivative) of the problem is calculated (ipopt only). The Hessian mode is invoked in the falcon.Problem constructor.

**+ `DisableAbort`** (read-only)
Flag that determines use of drawnow to allow for optimization abort in OptiFunc.

**+ `isSolved`** (read-only)
Flag to determine whether the problem is already solved

**+ `doSimInitGuess`** (read-only)
Use simulation as initial guess for state

**+ `MajorIterLimit`** (read-only, Default = 500)
MajorIterLimit for shortcut problem.Solve

**+ `MajorFeasTol`** (read-only, Default = 1e-05)
MajorFeasTol for shortcut problem.Solve

**+ `MajorOptTol`** (read-only, Default = 1e-05)
MajorOptTol for shortcut problem.Solve

**+ `gSparsity`** (Dependent)
Get the sparsity pattern of the gradient matrix as a sparse matrix

**+ `HSparsity`** (Dependent)
Get the sparsity pattern of the Hessian matrix as a sparse matrix

**+ `StateValues`** (Dependent)
extract and concatenate all state values

**+ `StateNames`** (Dependent)
extract and concatenate all state names

**+ `ControlValues`** (Dependent)
extract and concatenate all control values

**+ `ControlNames`** (Dependent)
extract and concatenate all control names

**+ `OutputValues`** (Dependent)
extract and concatenate all output values

**+ `OutputNames`** (Dependent)
extract and concatenate all output names

**+ `StateDotValues`** (Dependent)
extract and concatenate all state_dot values

**+ `StateDotNames`** (Dependent)
extract and concatenate all output names

**+ `CostateValues`** (Dependent)
extract and concatenate all costates values

**+ `RealTime`** (Dependent)
extract real time of problem

**+ `Name`** (read-only)
Name of object.

### Methods

**Problem** (Constructor)
Constructs a falcon.Problem object.

**> addNewMayerCost**
Add a new Mayer cost function to the Problem.

**> addNewParameterCost**
Minimize (default) or maximize a specific parameter

**> addNewPhase**
Add a new Phase to the Problem.

**> addNewPointConstraint**
Add a new point constraint to the Problem.

**> addNewStateCost**
Minimize (default) or maximize a specific state

**> addPostProcessingStep**
Add a post processing step to be performed after the problem has been solved.

**> addProblemExtension**
Adds an existing problem extension to the stack

**> Bake**
Bake the problem. (Prepare it for being solved).

**> CheckGradient**
Check the analytic gradient matrix of the problem against finite differences.

**> CheckScaling**
Check the scaling of the cost function, constraints values, optimization variables as well as the gradient of the problem.

**> clearPostProcessing**
falcon.Problem/clearPostProcessing is a function. clearPostProcessing(obj)

**> ConnectAllPhases**
Connect all phases in the problem.

**> ConnectPhases**
Connect two phases in the problem.

**> CreateDebugInfo**
Create additional information helpful for debugging.

**> getTimeSeries**
Extract the states, controls, outputs, statesdot and postprocessed values into Matlab TimeSeries object.

> **OpenPlotGUI** (Static)
Opens the plot graphical user interface with the given problem.

> **PlotGUI**
Opens the graphical user interface for plotting of the current problem.

> **setCostScaling**
Set the numeric scaling of the overall cost function value

> **setDiscretizationMethod**
Set the discretization method of this problem.

> **setMajorFeasTol**
Set the major feasibility tolerance for the problem.Solve method.

> **setMajorIterLimit**
Set the major iteration limit for the problem.Solve method.

> **setMajorOptTol**
Set the major optimality tolerance for the problem.Solve method.

> **setSimInitGuess**
Set the flag that determines whether to use a simulation for the initial guess of the states within each phase or not.

> **setUseHessian**
falcon.Problem/setUseHessian is a function. obj = setUseHessian(obj, useHessian)

> **showCostFunctionValues**
Extracts the values of the Mayer cost functions and prints them to the console.

> **Simulate**
Integrate the optimized trajectory using standard matlab solvers. The result will be a cell array containing all the simulated states for the phases.

> **SingleCallOptiFunc**
Calls the function used to perform the numerical optimization.

> **Solve**
Solve the problem.

> **ToStruct**
Create a struct from this problem.

> **UnBake**
UnBake the problem. Make it editable again.

### Constructor falcon.Problem

Creates a new FALCON.m problem ready to be used to solve an optimal control problem.
*Keywords*: Constructor Problem

#### - Syntax -

```
obj = Problem(name)
```

#### - Inputs -

**name** The name of the problem as a string.

#### - Name Value -

**UseHessian** Use the analytic Hessian to solve this problem (if available). (default: false)

### Method addNewMayerCost falcon.Problem

Creates a new PointFunction and adds it to the list of Mayer cost functions of the problem.
*Keywords*: Problem Cost Mayer

#### - Syntax -

```
obj.addNewMayerCost(FunctionHandle)
obj.addNewMayerCost(FunctionHandle, Cost)
obj.addNewMayerCost(FunctionHandle, Cost, Phase1, NormalizedTime1)
obj.addNewMayerCost(FunctionHandle, Cost, Phase1, NormalizedTime1,
    Phase2, NormalizedTime2)
obj.addNewMayerCost(FunctionHandle, Cost, ..., PhaseN, NormalizedTimeN)
```

#### - Inputs -

**FunctionHandle** The function handle to the function used to calculate this path constraint. For more information on the function handle see falcon.PointFunction or falcon.PointConstraintBuilder. If not specified via a point constraint builder file, the function handle must fulfill the following header convention (if in doubt, please let Falcon.m create the function interface for you): Cost = FunctionHandle(outputs, states, controls, parameters); Cost: a scalar cost value outputs: column vector of outputs, if the model has some (otherwise this input is omitted) states: column vector of states controls: column vector of controls, if the model has some (otherwise this input is omitted) parameters: column vector of parameters, that were set by setParameters method (otherwise the input is omitted)

**Cost** The Cost object defining the name of the output.

**Phase..** The phases where the data for this PointConstraint should be taken from.

**NormalizedTime..** The normalized time points where the data should be taken. Each input phase requires its own normalized time vector.

### Method addNewParameterCost falcon.Problem

Uses a parameter to create a new cost function for it.
*Keywords*: Problem Cost Parameter

#### - Syntax -

```
obj.addNewParameterCost(Parameter)
obj.addNewParameterCost(Parameter, Type)
obj.addNewParameterCost(..., 'Name', Value)
```

#### - Inputs -

**Parameter** The parameter to be minimized or maximized.

#### - Name Value -

**Scaling** The scaling value used for this cost function. (default = parameter.Scaling)

**Type** The type of cost as a string. Allowed values are 'min' and 'max'. (default: 'min').

### Method addNewPhase falcon.Problem

Creates a new phase, adds it to the list of phases of this problem and returns a handle to it.
*Keywords*: Problem Phase

#### - Syntax -

```
obj.addNewPhase(ModelHandle, States, NormalizedTime, StartTime,
    FinalTime)
```

#### - Inputs -

**ModelHandle** The function handle to the simulation model.

**States** The falcon.State objects used in this phase.

**NormalizedTime** The normalized time spacing of this phase.

**StartTime** The start time of this phase in realtime. Input must either be a positive scalar (time is fixed) or an falcon.Parameter

**FinalTime** The final time of this phase in realtime. Input must either be a positive scalar (time is fixed) or an falcon.Parameter

### Method addNewPointConstraint falcon.Problem

Creates a new PointFunction object and adds it to the problem.
*Keywords*: Problem Point Constraint

*- Syntax -*

```
obj.addNewPointConstraint(FunctionHandle, Constraints, Phase1,
    NormalizedTime1)
obj.addNewPointConstraint(FunctionHandle, Constraints, Phase1,
    NormalizedTime1, Phase2, NormalizedTime2)
obj.addNewPointConstraint(FunctionHandle, Constraints, ..., PhaseN,
    NormalizedTimeN)
```

*- Inputs -*

**FunctionHandle** The function handle to the function used to calculate this path constraint. For more information on the function handle see falcon.PointFunction or falcon.PointConstraintBuilder. If not specified via a point constraint builder file, the function handle must fulfill the following header convention (if in doubt, please let Falcon.m create the function interface for you): constraints = FunctionHandle(outputs, states, controls, parameters); constraints: column vector of constraints outputs: column vector of outputs, if the model has some (otherwise this input is omitted) states: column vector of states controls: column vector of controls, if the model has some (otherwise this input is omitted) parameters: column vector of parameters, that were set by setParameters method (otherwise the input is omitted)

**Constraints** The Constraint objects defining the name, boundaries, scaling and offset of the output

**Phase..** The phases where the data for this PointConstraint should be taken from.

**NormalizedTime..** The normalized time points where the data should be taken. Each input phase requires its own normalized time vector.

### Method addNewStateCost falcon.Problem

Uses a state to create a new cost function for it.
*Keywords*: Problem Cost State

*- Syntax -*

```
obj.addNewStateCost(State)
obj.addNewStateCost(State, Type)
obj.addNewStateCost(State, Type, Phase)
obj.addNewStateCost(State, Type, Phase, Tau)
obj.addNewStateCost(..., 'Name', Value)
```

*- Inputs -*

**State** The state to be minimized or maximized.

*- Name Value -*

**Type** The type of cost as a string. Allowed values are 'min' and 'max'. The default is 'min'.

**Phase** The phase from which the state value should be taken. (default: The last phase in the problem)

**Tau** The point in normalized time to take the state at. (default: 1, being final state of the respective phase)

**Scaling** The scaling value used for this cost function.

### Method addPostProcessingStep falcon.Problem

Add a post processing step to be performed after the problem has been solved. The post-processing is always applied to the full time interval and to all phases. Thus, it must support element-wise operations.

*Keywords*: Problem Post Process Add

#### - Syntax -

```
obj.addPostProcessingStep(func, inargscell, calcValues)

>func: Function handle (anonymous and standard) with multiple
inputs and a output(s) calculating the post-processing
value(s).
>inargscell: Cell array containing the function input
arguments (in correct order) that are required to calculate
the post-processing value. All falcon objects can be used as
inputs including already calculated post-processed values.
>calcValues: A falcon.Value object containg the name of the
post-processed value as saved in the PostProcessedGrid of
the phase.
```

### Method addProblemExtension falcon.Problem

*Keywords*: Problem Extension

#### - Syntax -

```
obj.addProblemExtension(probExt)
```

#### - Inputs -

**probExt** Problem extension instance

### Method Bake falcon.Problem

Bake the problem in order to prepare it for solving. The method needs to be called before the problem can be solved by any numeric solver. If you try to solve the problem with default settings using the method Problem.Solve() it will be baked automatically. Altering a problem after baking it is not possible. If you still want to change it, use Problem.UnBake().

*Keywords*: Problem Bake

*- Syntax -*

```
obj.Bake()
```

**Method CheckGradient** falcon.Problem

Calculates the analytic derivative of the problem and compares it to finite differences.
*Keywords*: Problem Checks Gradient

*- Syntax -*

```
[g, gnum] = obj.CheckGradient('Name', Value)
```

*- Name Value -*

**z** The optimization parameter vector to be used for the analysis. (default: zInitial)

**Delta_Z** The stepsize used for the finite differences. (default: sqrt(eps))

**Tolerance** The tolerance for comparison of the numeric and the analytic values. (default: 1e-4)

**Range** The range in z to be checked. (default: The whole z)

**Visualize** A boolean specifying if the result of the computation should be displayed. (default: true)

**doRandomize** Flag that allows a randomization of the optimization parameter vector considering bounds and magnitude (default: false).

**noSamples** Number of samples to randomly check the matrix with respect to finite differences (default: 1).

*- Outputs -*

**G** The sparse analytic gradient matrix of the overall problem. For multiple samples the last value is returned.

**Gnum** The sparse numeric gradient matrix of the overall problem. For multiple samples the last value is returned.

**gradientIsCorrect** Flag indicating that, if true, the matrix checks were successful for the respective samples.

**Method CheckScaling** falcon.Problem

This function checks the scaling of the z and the F vectors as well as the gradient matrix G. A wellformed optimization problem yields in the fact that all optimization parameters, constraints, cost function as well as the entries of G have the same magnitude (desired is abs(entry) $< 10$).
*Keywords*: Problem Checks Scaling

### - Syntax -

```
[F, G, z] = CheckScaling( z, 'Name', Value, ...)
[F, G, z] = CheckScaling( 'Name', Value, ...)
```

**z** The optimization parameter vector to be used for the analysis. (default: zOpt if available, else zInitial)

### - Name Value -

**zRange** Which optimization variables are checked. (default: 1:zLength)

**fRange** Which constraint values are checked. (default: 1:fLength)

**Tolerance** The tolerance for marking entry values as too large (default: 10).

**SolverTol** The solver tolerance for checking violated constraints (default: 1e-5).

**MinTolerance** The tolerance for marking entry values as too small (default: 1e-1).

**doMinBound** Flag for visualizing small entries to the matrix (default: false).

## Method clearPostProcessing falcon.Problem

*Keywords*: none

## Method ConnectAllPhases falcon.Problem

Creates constraints that link each phase in the problem to the next phase in the problem. The ordering is equal to the ordering in which the phases have been added to the problem.

*Keywords*: Problem Phase Connect All

### - Syntax -

```
obj.ConnectAllPhases()
```

## Method ConnectPhases falcon.Problem

Creates constraints that link two phases to each other. In the converged result, the states at the end of the left phase will be equal to those at the beginning of the right phase.

*Keywords*: Problem Phase Connect

### - Syntax -

```
obj.ConnectPhases(prevPhase, nextPhase)
```

### - Inputs -

**leftPhase** The left one of the two falcon.core.Phase objects to be connected. The last state of this phase will be forced to be equal to the first one of the right phase.

**rightPhase** The right one of the two falcon.core.Phase objects to be connected. The first state of this phase will be forced to be equal to the last one of the left phase.

### Method CreateDebugInfo falcon.Problem

Creates Information not required for optimization but mainly for debugging of the code. Information created includes zNames and fNames.

*Keywords*: Debugging Problem Information

#### - Syntax -

```
obj.CreateDebugInfo()
```

### Method getTimeSeries falcon.Problem

Takes the states, controls, outputs, state derivatives and postprocessed time histories of all phases in the problem and creates Matlab TimeSeries objects from them.

*Keywords*: Debugging Problem Time Series

#### - Syntax -

```
[statesTS, controlTS, outputTS, statesdotTS, postprocessedTS] =
    obj.getTimeSeries()
```

#### - Outputs -

**statesTS** A TimeSeries object holding all states in the problem.

**controlTS** A TimeSeries object holding all controls in the problem.

**outputTS** A TimeSeries object holding all outputs in the problem.

**statesdotTS** A TimeSeries object holding all state derivatives in the problem.

**postprocessedTS** A TimeSeries object holding all postprocessed data in the problem.

### Method OpenPlotGUI falcon.Problem

Creates a new instance of the plot GUI using the given problem. In the plot GUI, the user can select all available time histories in the problem and create plots from them.

*Keywords*: Problem GUI Open

#### - Syntax -

```
falcon.Problem.OpenPlotGUI(problem);
falcon.Problem.OpenPlotGUI(structure);
```

#### - Inputs -

**problem** The falcon.Problem or a struct created from a problem (Using the ToStruct() method) to be plotted.

### Method PlotGUI falcon.Problem

Creates a new instance of the plot GUI using the current problem. In the plot GUI, the user can select all available time histories in the problem and create plots from them.

*Keywords*: Problem GUI

#### - Syntax -

```
obj.PlotGUI();
```

### Method setCostScaling falcon.Problem

Sets the numeric value as the numeric scaling value for the overall cost function.

*Keywords*: Problem Cost Scaling

#### - Syntax -

```
obj.setCostScaling(CostScaling)
```

#### - Inputs -

**CostScaling** A numeric value used for scaling the cost function.

### Method setDiscretizationMethod falcon.Problem

Sets the given discretization method as the discretization method used to solve this optimal control problem.

*Keywords*: Problem Discretization Method, Solver Discretization Method,

#### - Syntax -

```
obj.setDiscretizationMethod(DiscretizationMethod)
```

#### - Inputs -

**DiscretizationMethod** A handle to a child class of falcon.discretization.DiscretizationMethod containing all the functions required to discretize an optimal control problem before solving it. The usable classes shipped with falcon can be found in falcon.discretization.

### Method setMajorFeasTol falcon.Problem

Set the major feasibility tolerance for the method problem.Solve. Value will be assigned there to the automatically created solver object.

*Keywords*: Solver Tolerance Feasibility

#### - Syntax -

```
obj.setMajorFeasTol(MajorFeasTolVal)
```

#### - Name Value -

**MajorFeasTolVal** Numeric value of the tolerance.

### Method setMajorIterLimit falcon.Problem

Set the major iteration limit for the method problem.Solve. Value will be assigned there to the automatically created solver object.

*Keywords*: Solver Limit Iteration

*- Syntax -*

```
obj.setMajorIterLimit(MajorIterLimitVal)
```

*- Name Value -*

**`MajorIterLimitVal`** Numeric value of the limit.

### Method setMajorOptTol falcon.Problem

Set the major optimality tolerance for the method problem.Solve. Value will be assigned there to the automatically created solver object.

*Keywords*: Solver Tolerance Optimality

*- Syntax -*

```
obj.setMajorOptTol(MajorOptTolVal)
```

*- Name Value -*

**`MajorOptTolVal`** Numeric value of the tolerance.

### Method setSimInitGuess falcon.Problem

Sets the boolean value of the simulation flag for the initial guess.

*Keywords*: Problem Simulation Flag

*- Syntax -*

```
obj.setSimInitGuess(SimInitGuessFlag)
```

*- Inputs -*

**`SimInitGuessFlag`** Boolean determining whether to use a simulation for the initial guess (true) or not (false). (defaul: false)

### Method setUseHessian falcon.Problem

*Keywords*: none

### Method showCostFunctionValues falcon.Problem

Show all Mayer cost functions in the console.

*Keywords*: Debugging Problem Cost Values

### - Syntax -

```
obj.showCostFunctionValues()
```

### Method Simulate falcon.Problem

Simulate the system with controls and initial state of the optimization.
*Keywords*: Problem Simulation

### - Syntax -

```
[states,outputs,simTime,statesDot] = obj.Simulate()
[states,outputs,simTime,statesDot] = obj.Simulate(init_states)
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history)
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver)
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options)
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime)
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime,Split
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime,Split
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime,Split
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime,Split
[states,outputs,simTime,statesDot] =
    obj.Simulate(init_states,control_history,ode_solver,ode_options,UseRealTime,Split
```

### - Name Value -

**`init_states`** The initial state vector for the split intervals. If backward integration is used the final values. Default are values from the current state grid.

**`control_history`** The control history to be used for the simulation. Default is the interpolated control grid.

**`ode_solver`** Solver type that should be used for the integration (default: ode45). Specified as a string.

**`ode_options`** Specify an ode options struct used for the ode solver. As default the standard ode settings are used.

**`UseRealTime`** Use the real, i.e. physical, time for the simulation instead of the non-dimensional time tau (default: true).

**`SplitIntervals`** Number of split intervals, i.e. intervals that split the integration domain. This generally makes the integration more stable.

**`UseODETimeStep`** Flag that specifies whether to use the internal ode time step for the measurements or the time steps from the optimization. Result might be helpful to determine the required number of collocation steps (default: false).

`UseBackwardInt` Flag that specifies whether to use the backward or forward integration in time (default: false).

`DoVisualization` Flag to visualize the results (default: false)

*- Outputs -*

`states` Simulated states with the initial condition and controls from the object.

`outputs` Simulated outputs with the initial condition and controls from the object.

`simTime` The time grid the integration is carried out.

`statesDot` The state derivatives.

### Method SingleCallOptiFunc falcon.Problem

The problem is solved by iteratively improving the solution. This method calls the function used in this iteration once an calculates the current constraint values and the current gradient. In case the problem was solved, this method uses the optimal solution, otherwise it uses the initial guess.

*Keywords*: Problem Opti Func, Discretization Opti Func

*- Syntax -*

```
[F, G] = obj.SingleCallOptiFunc()
```

**z** The optimization parameter vector. If not specified the initial guess or, if available, the optimal results us used.

*- Outputs -*

`F` The cost and constraint vector in the evaluated point. The first entry holds the cost function.

`G` The sparse Jacobian df/dz of the problem.

### Method Solve falcon.Problem

Solve the problem numerically either using the default solver and the default settings or using the given numeric solver.

*Keywords*: Problem Solve, Solver Solve

*- Syntax -*

```
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.Solve()
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.Solve(Solver)
```

### Method ToStruct falcon.Problem

Extracts all relevant information from this problem and stores it in the returned struct.

*Keywords*: Debugging Problem

*- Syntax -*

```
strc = obj.ToStruct()
strc = obj.ToStruct('Name', Value)
```

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

### Method UnBake falcon.Problem

Make a problem editable again, that was already baked. This is especially usefull when solving similar problems again and again.

*Keywords*: Problem UnBake

*- Syntax -*

```
obj.UnBake()
```

## 5.4 falcon.core.Phase

Parent Classes: falcon.core.Handle, falcon.core.HasToStruct, falcon.core.HasProblem

### Properties

**+ PhaseNumber** (read-only, Default = 0)
The number of this phase. Unique phase identifier.

**+ StateGrid** (read-only)
State grid of this phase (falcon.core.Grid)

**+ ControlGrids** (read-only)
Array of Control grids of this phase (falcon.core.Grid)

**+ DefectGrid** (read-only)
The grid holding the state defects in this phase

**+ CostateGrid** (read-only)
The grid holding the costates

**+ PathConstraintFunctions** (read-only)
Path Functions store fPathfunction which limit for instance a state over the whole phase

**+ LagrangeCostFunctions** (read-only)
Lagrange cost functions store fPathfunction which are added to the overall vost of this problem

**+ Model** (read-only)
Model calculate the dynamics either by using

**+ StartTime** (read-only)
The real start time of this phase (The normalized time always runs from 0 to 1, while the realtime runs from StartTime to FinalTime).

**+ FinalTime** (read-only)
The real final time of this phase (The normalized time always runs from 0 to 1, while the realtime runs from StartTime to FinalTime).

**+ InitialBoundaries** (read-only)
The initial boundary values for the states of this phase

**+ FinalBoundaries** (read-only)
The final boundary values for the states of this phase

**+ InterpolatedControlGrid** (read-only)
The interpolated values of the control grids w.r.t the state grid normalized time

**+ PostProcessedGrid** (read-only)
The grid holding the post processed values of this phase.

**+ SimulatedStateGrid** (read-only)
The grid holding the simulated states (post-processed) of this phase.

**+ ConnectedNextPhase** (read-only)
Next Phase to which the phase defect is constructed

**+ ConnectedStates** (read-only)
Connected states, for which phase defect is constructed

**+ zIndexStart** (read-only)
Start z index of phase

**+ fIndexStart** (read-only)
Start f index of phase

**+ PhaseExtensions**
falcon.core.Phase/PhaseExtensions is a property.

**+ ControlDataTypes** (Dependent)
(dependent) All combined DataTypes of the controls

**+ RealTime** (Dependent)
(dependent) Real time vector of the grid

### Methods

**> addNewControlGrid**
Adds a new control grid to the list of control grids of this phase.

**> addNewLagrangeCost**
Adds a new Lagrange cost function to the list of cost functions of this phase.

**> addNewPathConstraint**
Adds a new path function to the list of path functions of this phase.

**> addPhaseExtension**

**> addPostProcessingStep**
Add a post processing step to be performed after the problem has been solved.

**> ConnectToNextPhase**

**> setDurationLimit**
Sets the limits of the duration of this Phase.

**> setFinalBoundaries**
Sets the final boundary conditions for the states of this phase.

**> setInitialBoundaries**
Sets the initial boundary conditions for the states of this phase.

**> SimulatePhase**
Integrate the optimized trajectory using standard matlab solvers. The result will be a forward simulation of phase.

**> ToStruct**
Create a struct of the falcon.core.Phase-object.

### Method addNewControlGrid falcon.core.Phase

Creates a new control grid of type falcon.core.Grid based on the given normalized time and the given controls, adds it to the list of control grids of this phase and returns it. If no normalized time is specified the state-grid discretization is used.
*Keywords*: Phase Grid Control

*- Syntax -*

```
controlGrid = obj.addNewControlGrid(controls)
controlGrid = obj.addNewControlGrid(controls, normalizedTime)
```

*- Inputs -*

**controls** A vector of falcon.Control objects defining the controls to be used on this control grid.

**normalizedTime** The points in normalized time where the control values are defined. If no normalized time is specified the state-grid discretization is used.

*- Outputs -*

**controlGrid** Control-grid object.

### Method addNewLagrangeCost falcon.core.Phase

Creates a new Lagrange cost function based on the provided function handle, constraints and normalized time. The function is added to the list of path functions of this phase and is returned. If no normalized time is specified the state-grid discretization is used.

*Keywords*: Phase Lagrange Cost, Path Function Phase Cost

*- Syntax -*

```
lagrangeFunction = obj.addNewLagrangeCost(functionHandle, cost)
lagrangeFunction = obj.addNewLagrangeCost(functionHandle, cost,
    normalizedTime)
```

*- Inputs -*

**functionHandle** The function handle to the function used to calculate this Lagrange cost. For more information on the function handle see see falcon.PathFunction or falcon.PathConstraintBuilder. If not specified via a path constraint builder file, the function handle must fulfill the following header convention (if in doubt, please let Falcon.m create the function interface for you): cost = functionHandle(outputs, states, controls, parameters); cost: a scalar cost value outputs: column vector of outputs, if the model has some (otherwise this input is omitted) states: column vector of states controls: column vector of controls, if the model has some (otherwise this input is omitted) parameters: column vector of parameters, that were set by setParameters method (otherwise the input is omitted)

**cost** A vector of falcon.Cost objects defining the output of this cost function. The size of the vector has to fit the outputs of the function.

**normalizedTime** The points in normalized time to evaluate the path function on. If no normalized time is specified the StateGrid discretization is used.

*- Outputs -*

**lagrangeFunction** Lagrange-function object.

### Method addNewPathConstraint falcon.core.Phase

Creates a new path function based on the provided function handle, constraints and normalized time. The function is added to the list of path functions of this phase and is returned. If no normalized time is specified the state-grid discretization is used.

*Keywords*: Phase Path Constraint, Path Function Phase Constraint

*- Syntax -*

```
pathFunction = obj.addNewPathConstraint(functionHandle, constraints)
pathFunction = obj.addNewPathConstraint(functionHandle, constraints,
    normalizedTime)
```

*- Inputs -*

**functionHandle** The function handle to the function used to calculate this path constraint. For more information on the function handle see falcon.PathFunction or falcon.PathConstraintBuilder. If not specified via a path constraint builder file, the function handle must fulfill the following header convention (if in doubt, please let Falcon.m create the function interface for you): constraints = functionHandle(outputs, states, controls, parameters); constraints: column vector of constraints outputs: column vector of outputs, if the model has some (otherwise this input is omitted) states: column vector of states controls: column vector of controls, if the model has some (otherwise this input is omitted) parameters: column vector of parameters, that were set by setParameters method (otherwise the input is omitted)

**constraints** A vector of falcon.Constraint objects defining the boundaries, the scaling etc. of the values calculated by this path function. The size of the vector has to fit the outputs of the function.

**normalizedTime** The points in normalized time to evaluate the path function on. If no normalized time is specified the StateGrid discretization is used.

*- Outputs -*

**pathFunction** Path-function object.

### Method addPhaseExtension falcon.core.Phase

*Keywords*: Phase Extension

### Method addPostProcessingStep falcon.core.Phase

Add a post processing step to be performed after the problem has been solved. The post-processing is always applied to the full time interval and may differ for different phases. Thus, it must support element-wise operations.

*Keywords*: Phase Post Process Add

*- Syntax -*

```
obj.addPostProcessingStep(func, inargscell, calcValues)

>func: Function handle (anonymous and standard) with multiple
inputs and a single output calculating the post-processing
value.
>inargscell: Cell array containing the function input
arguments (in correct order) that are required to calculate
```

```
the post-processing value. All falcon objects can be used as
inputs including already calculated post-processed values.
>calcValues: A falcon.Value object containg the name of the
post-processed value as saved in the PostProcessedGrid of
the phase.
```

### Method ConnectToNextPhase falcon.core.Phase

*Keywords*: Phase Connect

### Method setDurationLimit falcon.core.Phase

Set the scalar, real valued limits of the duration of this phase. Please note that the lower bound of the phase duration must be smaller than the upper bound.

*Keywords*: Phase Duration Limit

#### - Syntax -

```
obj.setDurationLimit(DurationLowerBound, DurationUpperBound)
obj.setInitialBoundaries(DurationLowerBound, DurationUpperBound,
    'Name', Value)
```

#### - Inputs -

**DurationLowerBound** Lower bound of the duration of this phase.

**DurationUpperBound** Upper bound of the duration of this phase.

#### - Name Value -

**Offset** The offset of the phase duration value. (default: 0)

**Scaling** The scaling of the phase duration value. (default: Scaling of the final time - parameter of this phase)

### Method setFinalBoundaries falcon.core.Phase

Sets the final boundary conditions of the states to the provided values. There are several possibilities to set the boundary conditions for the states of this phase depending of the number and type of function arguments used for the call. Please note that this function may be called more than once. Each time this function is called the newly specified values are overwritten but conserving already set values!

*Keywords*: Phase Boundaries Final

#### - Syntax -

```
obj.setFinalBoundaries(EqualityBoundaries)
obj.setFinalBoundaries(LowerBounds, UpperBounds)
obj.setFinalBoundaries(falcon.State, EqualityBoundary)
obj.setFinalBoundaries(falcon.State, LowerBounds, UpperBounds)
```

**Case I** One function argument - numeric, real valued column vector with number of entries equal to the number of states in the phase. The lower and upper bounds are set to the same value specified in the column vector.

**Case II** Two function arguments - The first argument is an column vector of falcon.State objects. The second argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument. The lower and upper bounds for the falcon.State objects are set to the same value specified in the column vector.

**Case III** Two function arguments - The first argument is a numeric, real valued column vector specifying the lower bounds of the final boundary condition. The second argument is a numeric, real valued column vector specifying the upper bounds, respectively. Please note that the number of entries in the column vector for the lower and upper bounds have to be equal to the number of states in this phase. The boundaries are set according to the entries in the two column vectors.

**Case IV** Three function arguments - The first argument is an column vector of falcon.State objects. The second argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument and specifies the lower bounds for these states. The third argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument and specifies the upper bounds for these states.

*- Inputs -*

**EqualityBoundaries** A vector of the same size as the state vector for this phase that contains the values for the final boundary conditon. Lower and upper bounds are set to the same value specified in the array. If the vector contains inf or - inf values, these are ignored and replaced by the regular boundaries for the states.

**LowerBounds** A vector of the same size as the state vector for this pase that contains the lower boundaries for the final state values. If the vector contains -inf values, these are ignored and replaced by the regular boundaries for the states.

**UpperBounds** A vector of the same size as the state vector for this phase that contains the upper boundaries for the final state values. If the vector contains inf values, these are ignored and replaced by the regular boundaries for the states.

### Method setInitialBoundaries falcon.core.Phase

Sets the initial boundary conditions of the states to the provided values. There are several possibilities to set the boundary conditions for the states of this phase depending of the number and type of function arguments used for the call. Please note that this function may be called more than once. Each time this function is called the newly specified values are overwritten but conserving already set values!

*Keywords*: Phase Boundaries Initial

*- Syntax -*

```
obj.setInitialBoundaries(EqualityBoundaries)
obj.setInitialBoundaries(LowerBounds, UpperBounds)
obj.setInitialBoundaries(falcon.State, EqualityBoundary)
obj.setInitialBoundaries(falcon.State, LowerBounds, UpperBounds)
```

**Case I** One function argument - numeric, real valued column vector with number of entries equal to the number of states in the phase. The lower and upper bounds are set to the same value specified in the column vector.

**Case II** Two function arguments - The first argument is an column vector of falcon.State objects. The second argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument. The lower and upper bounds for the falcon.State objects are set to the same value specified in the column vector.

**Case III** Two function arguments - The first argument is a numeric, real valued column vector specifying the lower bounds of the final boundary condition. The second argument is a numeric, real valued column vector specifying the upper bounds, respectively. Please note that the number of entries in the column vector for the lower and upper bounds have to be equal to the number of states in this phase. The boundaries are set according to the entries in the two column vectors.

**Case IV** Three function arguments - The first argument is an column vector of falcon.State objects. The second argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument and specifies the lower bounds for these states. The third argument is a numeric, real valued column vector with the number of entries equal to the number of falcon.State objects in the first argument and specifies the upper bounds for these states.

*- Inputs -*

**EqualityBoundaries** A vector of the same size as the state vector for this phase that contains the values for the initial boundary conditon. Lower and upper bounds are set to the same value specified in the array. If the vector contains inf or - inf values, these are ignored and replaced by the regular boundaries for the states.

**LowerBounds** A vector of the same size as the state vector for this pase that contains the lower boundaries for the final state values. If the vector contains -inf values, these are ignored and replaced by the regular boundaries for the states.

**UpperBounds** A vector of the same size as the state vector for this phase that contains the upper boundaries for the final state values. If the vector contains inf values, these are ignored and replaced by the regular boundaries for the states.

### Method SimulatePhase falcon.core.Phase

Simulate the system with controls and initial state of the optimization.
*Keywords*: Phase Simulate, Problem Simulate Phase

### - Syntax -

```
[states,outputs,time,statesDot] = obj.SimulatePhase()
[states,outputs,time,statesDot] = obj.SimulatePhase(init_states)
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history)
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver)
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options)
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime)
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime,
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime,
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime,
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime,
[states,outputs,time,statesDot] =
    obj.SimulatePhase(init_states,control_history,ode_solver,ode_options,UseRealTime,
```

### - Name Value -

**`init_states`** The initial state vector for the split intervals. If backward integration is used the final values. Default are values from the current state grid.

**`control_history`** The control history to be used for the simulation. Default is the interpolated control grid.

**`ode_solver`** Solver type that should be used for the integration (default: ode45). Specified as a string.

**`ode_options`** Specify an ode options struct used for the ode solver. As default the standard ode settings are used.

**`UseRealTime`** Use the real, i.e. physical, time for the simulation instead of the non-dimensional time tau (default: true).

**`SplitIntervals`** Number of split intervals, i.e. intervals that split the integration domain. This generally makes the integration more stable.

**`UseODETimeStep`** Flag that specifies whether to use the internal ode time step for the measurements or the time steps from the optimization. Result might be helpful to determine the required number of collocation steps (default: false).

**`UseBackwardInt`** Flag that specifies whether to use the backward or forward integration in time (default: false).

**`DoVisualization`** Flag to visualize the results (default: false)

*- Outputs -*

**states** Simulated states with the initial condition and controls from the object.

**outputs** Simulated outputs with the initial condition and controls from the object.

**time** The time grid the integration is carried out.

**statesDot** Simulated state derivatives with the initial condition and controls from the object.

#### Method ToStruct falcon.core.Phase

This metod creates a struct of the falcon.core.Phase-object including all the necessary information of the phase.

*Keywords*: Debugging Phase

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Inputs -*

**obj** falcon.core.Phase-object to be transformed in a struct.

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of the falcon.core.Phase-object

## 5.5 falcon.core.Grid

Parent Classes: falcon.core.HasProblem, falcon.core.HasToStruct, falcon.core.Handle

#### Properties

**+ GRID_INTERPOLATION_LINEAR** (Constant, read-only, Default = linear)
Constant used to set the linear grid interpolation method

**+ GRID_INTERPOLATION_PREVIOUS** (Constant, read-only, Default = previous)
Constant used to set the previous grid interpolation method

**+ Phase** (read-only)
The phase this falcon.core.Grid belongs to

**+ DataTypes** (read-only)
Array to hold the datatypes of this grid

**+ `Type`** (read-only, Default = Base)
The type of this grid. Valid types are listed in the field ValidTypes

**+ `Values`** (read-only)
The time history values stored in the grid

**+ `NormalizedTime`** (read-only)
The normalized time points of this grid

**+ `Index`** (read-only)
The indices of this grid in either z or f

**+ `FBCIndex`** (read-only)
The indices of the final boundary condition for the multiple shooting in the f vector

**+ `Jacobian`** (read-only)
Jacobian Gradient of the Grid

**+ `Hessian`** (read-only)
Hessian Gradient of the Grid

**+ `InterpolationGradient`** (read-only)
The gradient of the interpolation scheme of this grid

**+ `InterpolationMethod`** (read-only, Default = linear)
The interpolation method for this grid

**+ `RelevantIndices`** (read-only)
The indices that are neither fixed nor disabled. RelevantIndices holds the indices of the states as numerical values (not logical).

**+ `StateGridIndices`** (read-only)
The indices of the points in time in this grid with respect to the stategrid of the respective phase

**+ `UpperBounds`** (Dependent)
(dependent) The upper bounds of the values

**+ `LowerBounds`** (Dependent)
(dependent) The lower bounds of the values

**+ `Scaling`** (Dependent)
(dependent) The scaling of the values

**+ `Offset`** (Dependent)
(dependent) The offsets of the values

**+ `Name`** (Dependent)
(dependent) The names for the values

**+ `NameStr`** (Dependent)
(dependent) The names for the values as a string

### Methods

**> getInterpolatedValues**
Method handing back the interpolated values of this grid

**> setholdSpecificValues**
Set the values of this grid.

**> setInterpolationMethod**
Set the interpolation method for the control grid

**> setSpecificValues**
Set the values of this grid.

**> setValues**
Set the values of this grid.

**> ToStruct**
Create a struct of the falcon.core.Grid-object.

### Method getInterpolatedValues falcon.core.Grid

Interpolates and returns these interpolated values of the falcon.core.Grid object.
*Keywords*: Grid Interpolation Values

*- Syntax -*

```
InterpValues = getInterpolatedValues(obj)
```

*- Outputs -*

**InterpValues** Numeric array of the interpolated values for this grid.

### Method setholdSpecificValues falcon.core.Grid

Used to set the initial guess for specific states. Additionally, specific states can also be set on hold and are not changed. The values are interpolated to the values needed in this grid using linear interpolation with extrapolation turned on. All non-specified values are set to NaN and post-processed in the phase.checkConsistency function, where they are set to the default values
*Keywords*: Grid Set and Hold Specific Values

*- Syntax -*

```
obj.setholdSpecificValues(setStatesControls,holdStatesControls,ConstantValues)
obj.setholdSpecificValues(setStatesControls,holdStatesControls,InitialValues,
    FinalValues)
obj.setholdSpecificValues(setStatesControls,holdStatesControls,NormalizedTime,
    Values)
obj.setholdSpecificValues(setStatesControls,holdStatesControls,RealTime,
    Values, 'Realtime', true)
```

*- Inputs -*

**setStatesControls** falcon state/control object containing the states/controls to be set (can also be empty)

**holdStatesControls** falcon state/control object containing the states/controls to be hold (i.e. not set to NaN) (can also be empty)

**ConstantValues** One vector of values copied to all points in time.

**InitialValues** The value of this grid for normalized time tau=0. Needs to have the exact same size as DataTypes.

**FinalValues** The value of this grid for normalized time tau=1. Needs to have the exact same size as DataTypes.

**NormalizedTime** A list of points in normalized time for which the values are given.

**Values** An array of size [DataTypes, length(time)] holding the values to be stored in this grid.

*- Name Value -*

**Realtime** Switch to change the time vector from normalized time to real time (default: false).

### Method setInterpolationMethod falcon.core.Grid

Used to set the interpolation method
*Keywords*: Grid Interpolation Method

*- Syntax -*

```
setInterpolationMethod(obj, method)
```

*- Inputs -*

**method** The interpolation methods supported by FALCON.m are 'linear' and 'previous'. Alternatively, the class constants GRID_INTERPOLATION_LINEAR and GRID_INTERPOLATION_ may be used.

### Method setSpecificValues falcon.core.Grid

Used to set the initial guess for specific states. The values are interpolated to the values needed in this grid using linear interpolation with extrapolation turned on. All non-specified values are set to NaN and post-processed in the phase.checkConsistency function, where they are set to the default values
*Keywords*: Grid Set Specific Values

*- Syntax -*

```
obj.setSpecificValues(setStatesControls,ConstantValues)
obj.setSpecificValues(setStatesControls,InitialValues, FinalValues)
obj.setSpecificValues(setStatesControls,NormalizedTime, Values)
obj.setSpecificValues(setStatesControls,RealTime, Values, 'Realtime',
    true)
```

*- Inputs -*

**setStatesControls** falcon state/control object containing the states/controls to be set

**ConstantValues** One vector of values copied to all points in time.

**InitialValues** The value of this grid for normalized time tau=0. Needs to have the exact same size as DataTypes.

**FinalValues** The value of this grid for normalized time tau=1. Needs to have the exact same size as DataTypes.

**NormalizedTime** A list of points in normalized time for which the values are given.

**Values** An array of size [DataTypes, length(time)] holding the values to be stored in this grid.

*- Name Value -*

**Realtime** Switch to change the time vector from normalized time to real time (default: false).

### Method setValues falcon.core.Grid

Used to set the initial guess. The values are interpolated to the values needed in this grid using linear interpolation with extrapolation turned on.

*Keywords*: Grid Set Values

*- Syntax -*

```
obj.setValues(ConstantValues)
obj.setValues(InitialValues, FinalValues)
obj.setValues(NormalizedTime, Values)
obj.setValues(RealTime, Values, 'Realtime', true)
```

*- Inputs -*

**ConstantValues** One vector of values copied to all points in time.

**InitialValues** The value of this grid for normalized time tau=0. Needs to have the exact same size as DataTypes.

**FinalValues** The value of this grid for normalized time tau=1. Needs to have the exact same size as DataTypes.

**NormalizedTime** A list of points in normalized time for which the values are given.

**Values** An array of size [DataTypes, length(time)] holding the values to be stored in this grid.

*- Name Value -*

**Realtime** Switch to change the time vector from normalized time to real time (default: false).

#### Method ToStruct falcon.core.Grid

This metod creates a struct of the falcon.core.Grid-object including all the necessary information of the grid.

*Keywords*: Debugging Grid

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Inputs -*

**obj** falcon.core.Grid-object to be transformed in a struct.

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of the falcon.core.Grid-object

## 5.6 falcon.core.Model

Parent Classes: falcon.core.Handle, falcon.core.HasToStruct, falcon.core.HasProblem

### Properties

**+ Phase** (read-only)
The phase this falcon.core.Model belongs to

**+ StateDotGrid** (read-only)
The grid for the state dot values

**+ ModelOutputGrid** (read-only)
The grid for the model outputs

**+ SimulatedStateDotGrid** (read-only)
The grid for the simulated state dot values

**+ SimulatedOutputGrid** (read-only)
The grid for the simulated model outputs

**+ ModelHandle** (read-only)
Model calculate the dynamics either by using Model.Simulate() for shooting methods or Model.Evaluate() for Collocation methods

**+ ModelInfoStruct** (read-only)
The struct holding all information about the model function.

**+ ModelParameters** (read-only)
The parameters used in the simulation of the model

**+ ModelConstants** (read-only)
The constants used for the simulation of the model

**+ ShootingIndices** (read-only)
The indices of the states to be integrated using the refined multiple shooting grid.

**+ SlowIndices** (read-only)
The indices of the states to be integrated using the coarse collocation grid.

**+ real_out_avail** (read-only)
Flag that determines whether we have a real, physical output of the system

**+ hasOutputs** (Dependent)
Specifies whether the Model has Outputs

### Methods

**> addModelConstants**
Add the given numbers to the list of constants used in the simulation model.

**> overwriteConstants**
Overwrites the given numbers in the list of constants used in this simulation model.

**> setModelOutputs**
Set the given falcon.Outputs as the outputs of the model.

**> setModelParameters**
Set the given parameters as the parameters used in the simulation model.

**> ToStruct**
Create a struct of the falcon.core.Model-object.

### Method addModelConstants falcon.core.Model

Adds the given array of constants to the list of constants relevant for the simulation model.

*Keywords*: Model Constants

*- Syntax -*

```
obj.addModelConstants(Constant1, Constant2, ..)
```

*- Inputs -*

**Constant** An array of constant numbers used in the simulation of the dynamic model.

### Method overwriteConstants falcon.core.Model

Overwrites the given cell of constants in the list of constants relevant for this simulation model.

*Keywords*: Model Overwrite Constants

*- Syntax -*

```
obj.overwriteConstants(Constant,...)
```

*- Inputs -*

**ConstantsCell** A cell of constant numbers used in this PathFunction.

**ConstantsIdx** An array of the size of constants cell containing the indices of the constants that should be overwritten.

### Method setModelOutputs falcon.core.Model

Sets the given array of falcon.Output objects as the outputs additional outputs for the model. In case there are finite limits defined for the outputs, the outputs are automatically limited to the respective values.

*Keywords*: Model Outputs

*- Syntax -*

```
obj.setModelOutputs(Outputs)
```

*- Inputs -*

**Outputs** An array of falcon.Output objects used to store the outputs of the model.

### Method setModelParameters falcon.core.Model

Sets the given array of parameters as the parameters relevant for the simulation model. All parameters given here will be used as the thrid input to the simulation model dynamics.

*Keywords*: Model Parameters

*- Syntax -*

```
obj.setModelParameters(Parameters)
```

*- Inputs -*

**Parameters** An array of falcon.Parameter objects used in the simulation of the dynamic model.

### Method ToStruct falcon.core.Model

This metod creates a struct of the falcon.core.Model-object including all the necessary information of the Model.

*Keywords*: Debugging Model

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Inputs -*

**obj** falcon.core.Model-object to be transformed in a struct.

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of the falcon.core.Model-object

## 5.7 falcon.State

Parent Classes: falcon.core.OVC, falcon.core.HasProblem

### Properties

**+ Scaling** (read-only, Default = 1)
Scaling Parameters initialized with 1. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ Offset** (read-only, Default = 0)
Offset parameter which is initialized with 0. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ Name** (read-only)
Name of object.

**+ LowerBound** (read-only, Default = -Inf)
Lower bound of the falcon.core.OVC value. It is initialized with minus infinity. Scaling and Offset are applied to the LowerBound as well.

**+ UpperBound** (read-only, Default = Inf)
Upper bound of the falcon.core.OVC value. It is initialized with plus infinity. Scaling and Offset are applied to the UpperBound as well.

### Methods

**State** (Constructor)
Constructor for falcon.State object.

**> eq**
== (EQ) Test handle equality. Handles are equal if they are handles for the same object. H1 == H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = EQ(H1, H2) stores the result in a logical array of the same dimensions.

**> ne**
= (NE) Not equal relation for handles. Handles are equal if they are handles for the same object and are unequal otherwise. H1 = H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = NE(H1, H2) stores the result in a logical array of the same dimensions.

**> setLowerBound**
Set the lower bound of this object.

**> setOffset**
Set the offset of this object.

**> setScaling**
Set the scaling of this object.

**> setUpperBound**
Set the upper bound of this object.

**> ToStruct**
Create a struct from this object.

### Constructor falcon.State

Constructs a new falcon.State object and returns it. Each state needs to have at least a valid name.

*Keywords*: Constructor State

*- Syntax -*

```
obj = falcon.State(name)
obj = falcon.State(name, lowerBound)
obj = falcon.State(name, lowerBound, upperBound)
obj = falcon.State(name, lowerBound, upperBound, scaling)
obj = falcon.State(name, lowerBound, upperBound, scaling, offset)
obj = falcon.State(name, 'Name', Value)
```

*- Inputs -*

**lowerBound** The lower boundary for this state. this value needs to be bigger than the upper boudn (default: -inf)

**upperBound** The upper boundary for this state. (default: inf)

**scaling** The scaling factor for this state. (default: 1)

**offset** The offset value for this state. (default: 0)

### Method eq falcon.State

*Keywords*: none

### Method ne falcon.State

*Keywords*: none

### Method setLowerBound falcon.State

Set the lower bound of this object. The input must be a real, scalar scalar and smaller than the upper bound.

*Keywords*: OVC Bound Lower

*- Syntax -*

```
obj.setLowerBound(lowerBound)
```

*- Inputs -*

**lowerBound** Lower bound of this object.

### Method setOffset falcon.State

Set the offset of this object. Please note that the input must be a real, scalar value.

*Keywords*: OVC Offset

*- Syntax -*

```
obj.setOffset(offset)
```

*- Inputs -*

**offset** Offset of this object.

### Method setScaling falcon.State

The input must be a real positive scalar value and should scale the value of this object to a range between -1 and 1.

*Keywords*: OVC Scaling

#### - Syntax -

```
obj.setScaling(scaling)
```

#### - Inputs -

**scaling** Scaling of the this object.

### Method setUpperBound falcon.State

Set the upper bound of this object. The input must be a real, scalar value and bigger than the lower bound.

*Keywords*: OVC Bound Upper

#### - Syntax -

```
obj.setUpperBound(upperBound)
```

#### - Inputs -

**upperBound** Upper bound of this object.

### Method ToStruct falcon.State

This metod creates a struct of this object including the fields: Name, LowerBound, UpperBound, Scaling and Offset.

*Keywords*: Debugging OVC

#### - Syntax -

```
strc = obj.ToStruct()
```

#### - Name Value -

**DebugData** Setting this option to true enables debug data in the ToStruct method.

#### - Outputs -

**strc** struct containing the inherent properties of this object.

## 5.8 falcon.Control

Parent Classes: falcon.core.OVC, falcon.core.HasProblem, falcon.core.HasFixed, falcon.core.HasSensit

### Properties

**+ `Scaling`** (read-only, Default = 1)
Scaling Parameters initialized with 1. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ `Offset`** (read-only, Default = 0)
Offset parameter which is initialized with 0. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ `Name`** (read-only)
Name of object.

**+ `LowerBound`** (read-only, Default = -Inf)
Lower bound of the falcon.core.OVC value. It is initialized with minus infinity. Scaling and Offset are applied to the LowerBound as well.

**+ `UpperBound`** (read-only, Default = Inf)
Upper bound of the falcon.core.OVC value. It is initialized with plus infinity. Scaling and Offset are applied to the UpperBound as well.

**+ `isFixed`** (read-only)
Whether this value is fixed or not.

**+ `isSensitive`** (read-only)
Whether this value is uncertain or not.

### Methods

**`Control`** (Constructor)
Constructor for falcon.Control object. Each control needs to have at least a valid name.

**> `eq`**
== (EQ) Test handle equality. Handles are equal if they are handles for the same object. H1 == H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = EQ(H1, H2) stores the result in a logical array of the same dimensions.

**> `ne`**
= (NE) Not equal relation for handles. Handles are equal if they are handles for the same object and are unequal otherwise. H1 = H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of

the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = NE(H1, H2) stores the result in a logical array of the same dimensions.

**> setFixed**
Sets the isFixed property of this object. Fixed objects will not be optimized.

**> setLowerBound**
Set the lower bound of this object.

**> setOffset**
Set the offset of this object.

**> setScaling**
Set the scaling of this object.

**> setSensitive**
Sets the isSensitive property of this object. Sensitive objects will be analysed by a Fiacco sensitivity analysis.

**> setUpperBound**
Set the upper bound of this object.

**> ToStruct**
Create a struct of the falcon.core.Control-object.

### Constructor falcon.Control

*Keywords*: Constructor Control

*- Syntax -*

```
obj = falcon.Control(Name)
obj = falcon.Control(Name, LowerBound)
obj = falcon.Control(Name, LowerBound, UpperBound)
obj = falcon.Control(Name, LowerBound, UpperBound, Scaling)
obj = falcon.Control(Name, LowerBound, UpperBound, Scaling, Offset)
obj = falcon.Control(Name, LowerBound, UpperBound, Scaling, Offset,
    Fixed)
obj = falcon.Control(Name, LowerBound, UpperBound, Scaling, Offset,
    Fixed, Sensitive)
obj = falcon.Control(Name, 'Name', Value)
```

*- Inputs -*

**LowerBound** The lower boundary for this control. (default: -inf)

**UpperBound** The upper boundary for this control. (default: inf)

**Scaling** The scaling factor for this control. (default: 1)

**Offset** The offset value for this control. (default: 0)

**Fixed** true or false, determines whether this control is subject to optimization or not. (default: false)

**Sensitive** true or false, determines whether this parameter is used within the sensitivity analysis or not. (default: false)

### Method eq falcon.Control

*Keywords*: none

### Method ne falcon.Control

*Keywords*: none

### Method setFixed falcon.Control

Sets if this object can be optimized or not.
*Keywords*: Flags Fixed

*- Syntax -*

```
obj.setFixed(fixed)
```

*- Inputs -*

**fixed** A scalar boolean specifying if the object is fixed or not. Fixed objects are not subject to optimization.

### Method setLowerBound falcon.Control

Set the lower bound of this object. The input must be a real, scalar scalar and smaller than the upper bound.
*Keywords*: OVC Bound Lower

*- Syntax -*

```
obj.setLowerBound(lowerBound)
```

*- Inputs -*

**lowerBound** Lower bound of this object.

### Method setOffset falcon.Control

Set the offset of this object. Please note that the input must be a real, scalar value.
*Keywords*: OVC Offset

*- Syntax -*

```
obj.setOffset(offset)
```

*- Inputs -*

**offset** Offset of this object.

### Method setScaling falcon.Control

The input must be a real positive scalar value and should scale the value of this object to a range between -1 and 1.

*Keywords*: OVC Scaling

*- Syntax -*

```
obj.setScaling(scaling)
```

*- Inputs -*

**scaling** Scaling of the this object.

### Method setSensitive falcon.Control

Sets if this object is sensitive or not.

*Keywords*: Flags Sensitive

*- Syntax -*

```
obj.setSensitive(Sensitive)
```

*- Inputs -*

**Sensitive** A scalar boolean specifying if the object is sensitive or not. Sensitive objects will be subject to a sensitivity analysis via a Fiacco update.

### Method setUpperBound falcon.Control

Set the upper bound of this object. The input must be a real, scalar value and bigger than the lower bound.

*Keywords*: OVC Bound Upper

*- Syntax -*

```
obj.setUpperBound(upperBound)
```

*- Inputs -*

**upperBound** Upper bound of this object.

### Method ToStruct falcon.Control

This metod creates a struct of the falcon.core.Control-object including all the necessary information of this Control.

*Keywords*: Debugging Control

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of the falcon.core.Control-object

## 5.9 falcon.Parameter

Parent Classes: falcon.core.OVC, falcon.core.HasFixed, falcon.core.HasProblem, falcon.core.HasSensit

### Properties

**+ Value** (read-only, Default = 0)
The current value of this parameter

**+ Index** (read-only, Default = 0)
Index the index of this parameter in the z-Vector

**+ Scaling** (read-only, Default = 1)
Scaling Parameters initialized with 1. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ Offset** (read-only, Default = 0)
Offset parameter which is initialized with 0. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ Name** (read-only)
Name of object.

**+ LowerBound** (read-only, Default = -Inf)
Lower bound of the falcon.core.OVC value. It is initialized with minus infinity. Scaling and Offset are applied to the LowerBound as well.

**+ UpperBound** (read-only, Default = Inf)
Upper bound of the falcon.core.OVC value. It is initialized with plus infinity. Scaling and Offset are applied to the UpperBound as well.

**+ isFixed** (read-only)
Whether this value is fixed or not.

**+ isSensitive** (read-only)
Whether this value is uncertain or not.

### Methods

**Parameter** (Constructor)
Constructor for falcon.Parameter object. Each parameter needs to have at

**> eq**
== (EQ) Test handle equality. Handles are equal if they are handles for the same object. H1 == H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = EQ(H1, H2) stores the result in a logical array of the same dimensions.

**> ne**
= (NE) Not equal relation for handles. Handles are equal if they are handles for the same object and are unequal otherwise. H1 = H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = NE(H1, H2) stores the result in a logical array of the same dimensions.

**> setFixed**
Sets the isFixed property of this object. Fixed objects will not be optimized.

**> setLowerBound**
Set the lower bound of this object.

**> setOffset**
Set the offset of this object.

**> setScaling**
Set the scaling of this object.

**> setSensitive**
Sets the isSensitive property of this object. Sensitive objects will be analysed by a Fiacco sensitivity analysis.

**> setUpperBound**
Set the upper bound of this object.

**> setValue**
Sets the current value of this falcon.Parameter.

**> ToStruct**
Create a struct from this parameter.

### Constructor falcon.Parameter

*Keywords*: Constructor Parameter

#### - Syntax -

```
obj = falcon.Parameter(name)
obj = falcon.Parameter(name, value)
obj = falcon.Parameter(name, value, lowerBound)
obj = falcon.Parameter(name, value, lowerBound, upperBound)
obj = falcon.Parameter(name, value, lowerBound, upperBound, scaling)
obj = falcon.Parameter(name, value, lowerBound, upperBound, scaling,
    offset)
obj = falcon.Parameter(name, value, lowerBound, upperBound, scaling,
    offset, Fixed)
obj = falcon.Parameter(name, value, lowerBound, upperBound, scaling,
    offset, Fixed, Sensitive)
obj = falcon.Parameter(..., 'Name', Value)
```

#### - Inputs -

**`Value`** The current (initial) value of this parameter. (default: 0)

**`LowerBound`** The lower boundary for this parameter. (default: -inf)

**`UpperBound`** The upper boundary for this parameter. (default: inf)

**`Scaling`** The scaling factor for this parameter. (default: 1)

**`Offset`** The offset value for this parameter. (default: 0)

**`Fixed`** true or false, determines whether this parameter is subject to optimization or not. (default: false)

**`Sensitive`** true or false, determines whether this parameter is used within the sensitivity analysis or not. (default: false)

### Method eq falcon.Parameter

*Keywords*: none

### Method ne falcon.Parameter

*Keywords*: none

### Method setFixed falcon.Parameter

Sets if this object can be optimized or not.
*Keywords*: Flags Fixed

*- Syntax -*

```
obj.setFixed(fixed)
```

*- Inputs -*

**fixed** A scalar boolean specifying if the object is fixed or not. Fixed objects are not subject to optimization.

### Method setLowerBound falcon.Parameter

Set the lower bound of this object. The input must be a real, scalar scalar and smaller than the upper bound.

*Keywords*: OVC Bound Lower

*- Syntax -*

```
obj.setLowerBound(lowerBound)
```

*- Inputs -*

**lowerBound** Lower bound of this object.

### Method setOffset falcon.Parameter

Set the offset of this object. Please note that the input must be a real, scalar value.

*Keywords*: OVC Offset

*- Syntax -*

```
obj.setOffset(offset)
```

*- Inputs -*

**offset** Offset of this object.

### Method setScaling falcon.Parameter

The input must be a real positive scalar value and should scale the value of this object to a range between -1 and 1.

*Keywords*: OVC Scaling

*- Syntax -*

```
obj.setScaling(scaling)
```

*- Inputs -*

**scaling** Scaling of the this object.

### Method setSensitive falcon.Parameter

Sets if this object is sensitive or not.

*Keywords*: Flags Sensitive

*- Syntax -*

```
obj.setSensitive(Sensitive)
```

*- Inputs -*

**Sensitive** A scalar boolean specifying if the object is sensitive or not. Sensitive objects will be subject to a sensitivity analysis via a Fiacco update.

### Method setUpperBound falcon.Parameter

Set the upper bound of this object. The input must be a real, scalar value and bigger than the lower bound.

*Keywords*: OVC Bound Upper

*- Syntax -*

```
obj.setUpperBound(upperBound)
```

*- Inputs -*

**upperBound** Upper bound of this object.

### Method setValue falcon.Parameter

Sets the current value of this parameter to the given value.

*Keywords*: Parameter Value

*- Syntax -*

```
obj.setValue(Value)
```

*- Inputs -*

**Value** The numeric value of this parameter. Needs to be scalar.

### Method ToStruct falcon.Parameter

Extracts all relevant information from this parameter and stores it in the returned struct.

*Keywords*: Debugging Parameter

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of this object.

## 5.10 falcon.Constraint

Parent Classes: falcon.core.OVC, falcon.core.HasProblem, falcon.core.HasActive

### Properties

**+ `Scaling`** (read-only, Default = 1)
Scaling Parameters initialized with 1. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ `Offset`** (read-only, Default = 0)
Offset parameter which is initialized with 0. The offset and scaling is assigned in the following way:

x_scaled = (x_unscaled - Offset) * Scaling

**+ `Name`** (read-only)
Name of object.

**+ `LowerBound`** (read-only, Default = -Inf)
Lower bound of the falcon.core.OVC value. It is initialized with minus infinity. Scaling and Offset are applied to the LowerBound as well.

**+ `UpperBound`** (read-only, Default = Inf)
Upper bound of the falcon.core.OVC value. It is initialized with plus infinity. Scaling and Offset are applied to the UpperBound as well.

**+ `isActive`** (read-only)
Whether this value is active or not.

### Methods

**`Constraint`** (Constructor)
Constructor for falcon.Constraint object. Each constraint needs to have at least a valid name.

**> `ArrayWith`** (Static)
Create array of falcon.Constraint objects

**> `eq`**
== (EQ) Test handle equality. Handles are equal if they are handles for the same object. H1 == H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = EQ(H1, H2) stores the result in a logical array of the same dimensions.

**> ne**
= (NE) Not equal relation for handles. Handles are equal if they are handles for the same object and are unequal otherwise. H1 = H2 performs element-wise comparisons between handle arrays H1 and H2. H1 and H2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of H1 or H2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar. TF = NE(H1, H2) stores the result in a logical array of the same dimensions.

**> setActive**
Sets the isActive property of this object.

**> setLowerBound**
Set the lower bound of this object.

**> setOffset**
Set the offset of this object.

**> setScaling**
Set the scaling of this object.

**> setUpperBound**
Set the upper bound of this object.

**> ToStruct**
Create a struct from this constraint.

### Constructor falcon.Constraint

*Keywords*: Constructor Constraint

*- Syntax -*

```
obj = falcon.Constraint(name)
obj = falcon.Constraint(name, lowerBound)
obj = falcon.Constraint(name, lowerBound, upperBound)
obj = falcon.Constraint(name, lowerBound, upperBound, scaling)
obj = falcon.Constraint(name, lowerBound, upperBound, scaling, offset)
obj = falcon.Constraint(name, lowerBound, upperBound, scaling,
    offset, active)
obj = falcon.Constraint(name, 'Name', Value)
```

*- Inputs -*

**name** The name of this constraint object.

**lowerBound** The lower boundary for this constraint. (default: -inf)

**upperBound** The upper boundary for this constraint. (default: inf)

**scaling** The scaling factor for this constraint. (default: 1)

**offset** The offset value for this constraint. (default: 0)

**active** true or false, determines whether this constraint is respected in the optimization or not. (default: true)

### Method ArrayWith falcon.Constraint

Creates an array of falcon.Constraint objects. This method is to a shortcut to create a vector of the constraint objects without creating each object individually.

*Keywords*: Constraint Array

#### - Syntax -

```
arr = falcon.Constraint.ArrayWith(names)
arr = falcon.Constraint.ArrayWith(names,LowerBound)
arr = falcon.Constraint.ArrayWith(names,LowerBound,UpperBound)
arr = falcon.Constraint.ArrayWith(names,LowerBound,UpperBound,Scaling)
arr =
    falcon.Constraint.ArrayWith(names,LowerBound,UpperBound,Scaling,Offset)
arr = falcon.Constraint.ArrayWith(names,LowerBound,UpperBound,Scaling,
    Offset,Active)
```

#### - Inputs -

**names** The names of the falcon.Constraint object as a cell array.

**LowerBound** The sorted lower bounds of the constraint object. The size needs to match the number of constraint names.

**UpperBound** The sorted upper bounds of the constraint object. The size needs to match the number of constraint names.

**Scaling** The sorted scalings of the constraint object. The size needs to match the number of constraint names.

**Offset** The sorted offsets of the constraint object. The size needs to match the number of constraint names.

**Active** The sorted active flags of the constraint object. The size needs to match the number of constraint names.

### Method eq falcon.Constraint

*Keywords*: none

### Method ne falcon.Constraint

*Keywords*: none

### Method setActive falcon.Constraint

Set whether this object is active. In case the object is not active it will be ignored during optimization.

*Keywords*: Flags Active

#### - Syntax -

```
obj.setActive(isActive)
```

### Method setLowerBound falcon.Constraint

Set the lower bound of this object. The input must be a real, scalar scalar and smaller than the upper bound.

*Keywords*: OVC Bound Lower

#### - Syntax -

```
obj.setLowerBound(lowerBound)
```

#### - Inputs -

**`lowerBound`** Lower bound of this object.

### Method setOffset falcon.Constraint

Set the offset of this object. Please note that the input must be a real, scalar value.

*Keywords*: OVC Offset

#### - Syntax -

```
obj.setOffset(offset)
```

#### - Inputs -

**`offset`** Offset of this object.

### Method setScaling falcon.Constraint

The input must be a real positive scalar value and should scale the value of this object to a range between -1 and 1.

*Keywords*: OVC Scaling

#### - Syntax -

```
obj.setScaling(scaling)
```

#### - Inputs -

**`scaling`** Scaling of the this object.

### Method setUpperBound falcon.Constraint

Set the upper bound of this object. The input must be a real, scalar value and bigger than the lower bound.

*Keywords*: OVC Bound Upper

*- Syntax -*

```
obj.setUpperBound(upperBound)
```

*- Inputs -*

**upperBound** Upper bound of this object.

### Method ToStruct falcon.Constraint

Extracts all relevant information from this constraint and stores it in the returned struct.

*Keywords*: Debugging Constraint

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

## 5.11 falcon.core.PointFunction

Parent Classes: falcon.core.PathFunction

### Properties

**+ RelevantPhases** (read-only)
The relevant phases for this pointfunction

**+ RelevantStateGridIndices** (read-only)
The relevant state grid indices for this pointfunction

**+ RelevantNormalizedTimeSteps** (read-only)
The relevant normalized times for this grid

**+ Phase** (read-only)
The phase this falcon.core.PathFunction belongs to

**+ FunctionHandle** (read-only)
Functionhandle to the function to be called

**+ FunctionInfoStruct** (read-only)
The struct keeping the information about the inputs and outputs of the used function.

**+ `OutputGrid`** (read-only)
Grids for the outputs of the path function. In case of a path function the time points are also used for the inputs.

**+ `Parameters`** (read-only)
Relevant parameters

**+ `Constants`** (read-only)
Relevant constants

**+ `RelevantStateIndices`** (read-only)
The indices of the states required for this function

**+ `RelevantControlIndices`** (read-only)
The indices of the controls required for this function

**+ `RelevantModelOutputIndices`** (read-only)
The indices of the model outputs required for this function

**+ `RelevantParameterIndices`** (read-only)
The indices of the function parameters required by this function. Sorts the Parameters of the function to the parameters of the derivative model.

**+ `OutputMultipliers`** (read-only)
The multipliers of the output constraints for the Hamiltonian of the problem.

### Methods

**> `addConstants`**
Add the given numbers to the list of constants used in this PathFunction.

**> `overwriteConstants`**
Overwrites the given numbers in the list of constants used in this PathFunction.

**> `setParameters`**
Set the given parameters as the parameters required in this PathFunction.

**> `ToStruct`**
Create a struct of this PathFunction object.

### Method addConstants falcon.core.PointFunction

Adds the given array of constants to the list of constants relevant for this PathFunction.
*Keywords*: Path Function Constants

*- Syntax -*

```
obj.addConstants(Constant, ..)
```

*- Inputs -*

**`Constants`** An array of constant numbers used in this PathFunction.

### Method overwriteConstants falcon.core.PointFunction

Overwrites the given cell of constants in the list of constants relevant for this PathFunction.

*Keywords*: Path Function Overwrite Constants

#### - Syntax -

```
obj.overwriteConstants(Constant,...)
```

#### - Inputs -

**ConstantsCell** A cell of constant numbers used in this PathFunction.

**ConstantsIdx** An array of the size of constants cell containing the indices of the constants that should be overwritten.

### Method setParameters falcon.core.PointFunction

Sets the given array of parameters as the parameters relevant for this PathFunction. All parameters given here will be used as the thrid input to the PathFunction.

*Keywords*: Path Function Parameters

#### - Syntax -

```
obj.setParameters(Parameters)
```

#### - Inputs -

**Parameters** An array of falcon.Parameter objects used in this PathFunction.

### Method ToStruct falcon.core.PointFunction

This metod creates a struct of the falcon.core.PathFunction-object including all the necessary information of the path function.

*Keywords*: Debugging Path Function

#### - Syntax -

```
strc = obj.ToStruct()
```

#### - Inputs -

**obj** falcon.core.PathFunction-object to be transformed in a struct.

#### - Name Value -

**DebugData** Setting this option to true enables debug data in the ToStruct method.

#### - Outputs -

**strc** struct containing the inherent properties of the falcon.core.PathFunction-object

## 5.12 falcon.core.PathFunction

Parent Classes: falcon.core.Handle, falcon.core.HasToStruct, falcon.core.HasProblem, matlab.mixin.Heterogeneous

### Properties

**+ `Phase`** (read-only)
The phase this falcon.core.PathFunction belongs to

**+ `FunctionHandle`** (read-only)
Functionhandle to the function to be called

**+ `FunctionInfoStruct`** (read-only)
The struct keeping the information about the inputs and outputs of the used function.

**+ `OutputGrid`** (read-only)
Grids for the outputs of the path function. In case of a path function the time points are also used for the inputs.

**+ `Parameters`** (read-only)
Relevant parameters

**+ `Constants`** (read-only)
Relevant constants

**+ `RelevantStateIndices`** (read-only)
The indices of the states required for this function

**+ `RelevantControlIndices`** (read-only)
The indices of the controls required for this function

**+ `RelevantModelOutputIndices`** (read-only)
The indices of the model outputs required for this function

**+ `RelevantParameterIndices`** (read-only)
The indices of the function parameters required by this function. Sorts the Parameters of the function to the parameters of the derivative model.

**+ `OutputMultipliers`** (read-only)
The multipliers of the output constraints for the Hamiltonian of the problem.

### Methods

**> `addConstants`**
Add the given numbers to the list of constants used in this PathFunction.

**> `overwriteConstants`**
Overwrites the given numbers in the list of constants used in this PathFunction.

**> setParameters**
Set the given parameters as the parameters required in this PathFunction.

**> ToStruct**
Create a struct of this PathFunction object.

### Method addConstants falcon.core.PathFunction

Adds the given array of constants to the list of constants relevant for this PathFunction.
*Keywords*: Path Function Constants

*- Syntax -*

```
obj.addConstants(Constant, ..)
```

*- Inputs -*

**Constants** An array of constant numbers used in this PathFunction.

### Method overwriteConstants falcon.core.PathFunction

Overwrites the given cell of constants in the list of constants relevant for this PathFunction.
*Keywords*: Path Function Overwrite Constants

*- Syntax -*

```
obj.overwriteConstants(Constant,...)
```

*- Inputs -*

**ConstantsCell** A cell of constant numbers used in this PathFunction.

**ConstantsIdx** An array of the size of constants cell containing the indices of the constants that should be overwritten.

### Method setParameters falcon.core.PathFunction

Sets the given array of parameters as the parameters relevant for this PathFunction. All parameters given here will be used as the thrid input to the PathFunction.
*Keywords*: Path Function Parameters

*- Syntax -*

```
obj.setParameters(Parameters)
```

*- Inputs -*

**Parameters** An array of falcon.Parameter objects used in this PathFunction.

#### Method ToStruct falcon.core.PathFunction

This metod creates a struct of the falcon.core.PathFunction-object including all the necessary information of the path function.

*Keywords*: Debugging Path Function

*- Syntax -*

```
strc = obj.ToStruct()
```

*- Inputs -*

**obj** falcon.core.PathFunction-object to be transformed in a struct.

*- Name Value -*

**DebugData** Setting this option to true enables debug data in the ToStruct method.

*- Outputs -*

**strc** struct containing the inherent properties of the falcon.core.PathFunction-object

## 5.13 falcon.discretization.Trapezoidal

Parent Classes: falcon.discretization.DiscretizationMethod

#### Methods

**Trapezoidal** (Constructor)
This class represents a trapezoidal collocation method

**> evaluateC**
Evaluate the constraints of the optimal control problem

**> evaluateF**
Evaluate the residual vector of the problem for testing purposes.

**> evaluateFandG**
Evaluate the residual vector and gradient of the problem for testing purposes.

**> evaluateG**
Evaluate the gradient matrix of the problem for testing purposes.

**> evaluateJ**
Evaluate the cost function for the differential evolution.

#### Constructor falcon.discretization.Trapezoidal

*Keywords*: none

### Method evaluateC falcon.discretization.Trapezoidal

Uses the parameter vector z to return the current constraints of the optimal control problem.

*Keywords*: Discretization Evaluate Constraint

#### - Syntax -

```
C = obj.evaluateC(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateF falcon.discretization.Trapezoidal

Uses the given parameter vector z to evaluate the residual vector of the discretized problem.

*Keywords*: Discretization Evaluate Residual

#### - Syntax -

```
f = obj.evaluateF(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateFandG falcon.discretization.Trapezoidal

Uses the given parameter vector z to evaluate the residual vector and gradient of the discretized problem.

*Keywords*: Discretization Evaluate Residual, Discretization Evaluate Gradient

#### - Syntax -

```
[F,G] = obj.evaluateF(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateG falcon.discretization.Trapezoidal

Uses the given parameter vector z to evaluate the sparse gradient matrix of the discretized problem.

*Keywords*: Discretization Evaluate Gradient

#### - Syntax -

```
grad = obj.evaluateG(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateJ falcon.discretization.Trapezoidal

Uses the parameter vector z to return the current cost functional.
*Keywords*: Discretization Evaluate Cost

*- Syntax -*

```
J = obj.evaluateJ(z)
```

*- Inputs -*

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

## 5.14 falcon.discretization.BackwardEuler

Parent Classes: falcon.discretization.DiscretizationMethod

### Methods

**BackwardEuler** (Constructor)
This class represents a backward Euler collocation method

**> evaluateC**
Evaluate the constraints of the optimal control problem

**> evaluateDE**
Evaluate the residual vector of the problem for testing purposes.

**> evaluateF**
Evaluate the residual vector of the problem for testing purposes.

**> evaluateFandG**
Evaluate the residual vector and gradient of the problem for testing purposes.

**> evaluateG**
Evaluate the gradient matrix of the problem for testing purposes.

**> evaluateGandH**
Evaluates the jacobian and the hessian.

**> evaluateJ**
Evaluate the cost function for the differential evolution.

### Constructor falcon.discretization.BackwardEuler

*Keywords*: none

### Method evaluateC falcon.discretization.BackwardEuler

Uses the parameter vector z to return the current constraints of the optimal control problem.

*Keywords*: Discretization Evaluate Constraint

#### - Syntax -

```
C = obj.evaluateC(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateDE falcon.discretization.BackwardEuler

Uses the given parameter vector z to evaluate the residual vector of the discretized problem.

*Keywords*: Discretization Evaluate DE

#### - Syntax -

```
f = obj.evaluateF(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateF falcon.discretization.BackwardEuler

Uses the given parameter vector z to evaluate the residual vector of the discretized problem.

*Keywords*: Discretization Evaluate Residual

#### - Syntax -

```
f = obj.evaluateF(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateFandG falcon.discretization.BackwardEuler

Uses the given parameter vector z to evaluate the residual vector and gradient of the discretized problem.

*Keywords*: Discretization Evaluate Residual, Discretization Evaluate Gradient

#### - Syntax -

```
[F,G] = obj.evaluateF(z)
```

#### - Inputs -

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateG falcon.discretization.BackwardEuler

Uses the given parameter vector z to evaluate the sparse gradient matrix of the discretized problem.

*Keywords*: Discretization Evaluate Gradient

*- Syntax -*

```
grad = obj.evaluateG(z)
```

*- Inputs -*

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

### Method evaluateGandH falcon.discretization.BackwardEuler

*Keywords*: Discretization Evaluate Hessian, Discretization Evaluate Gradient

### Method evaluateJ falcon.discretization.BackwardEuler

Uses the parameter vector z to return the current cost functional.

*Keywords*: Discretization Evaluate Cost

*- Syntax -*

```
J = obj.evaluateJ(z)
```

*- Inputs -*

**z** A parameter vector for the discretized problem (use e.g. problem.zInitial).

## 5.15 falcon.solver.ipopt

Parent Classes: falcon.solver.Optimizer

### Properties

**+ MU_STRATEGY_MONOTONE** (Constant, read-only, Default = monotone)
Identifier to set the mu strategy in ipopt to monotone.

**+ MU_STRATEGY_ADAPTIVE** (Constant, read-only, Default = adaptive)
Identifier to set the my strategy in ipopt to adaptive.

**+ WarmStartBoundPush** (read-only, Default = 1e-15)
falcon.solver.ipopt/WarmStartBoundPush is a property.

**+ WarmStartBoundFrac** (read-only, Default = 1e-15)
falcon.solver.ipopt/WarmStartBoundFrac is a property.

**+ WarmStartSlackBoundFrac** (read-only, Default = 1e-15)
falcon.solver.ipopt/WarmStartSlackBoundFrac is a property.

**+ `WarmStartSlackBoundPush`** (read-only, Default = 1e-15)
falcon.solver.ipopt/WarmStartSlackBoundPush is a property.

**+ `WarmStartMultBoundPush`** (read-only, Default = 1e-15)
falcon.solver.ipopt/WarmStartMultBoundPush is a property.

**+ `WarmStartMultInitMax`** (read-only, Default = 2e+20)
falcon.solver.ipopt/WarmStartMultInitMax is a property.

**+ `BoundRelaxFactor`** (read-only, Default = 0)
falcon.solver.ipopt/BoundRelaxFactor is a property.

**+ `mu_init`** (read-only, Default = 0.1)
falcon.solver.ipopt/mu_init is a property.

**+ `mu_target`** (read-only, Default = 0)
falcon.solver.ipopt/mu_target is a property.

**+ `mu_min`** (read-only, Default = 1e-11)
falcon.solver.ipopt/mu_min is a property.

**+ `mu_max`** (read-only, Default = 100000)
falcon.solver.ipopt/mu_max is a property.

**+ `mu_max_fact`** (read-only, Default = 1000)
falcon.solver.ipopt/mu_max_fact is a property.

**+ `barrier_tol_factor`** (read-only, Default = 1)
falcon.solver.ipopt/barrier_tol_factor is a property.

**+ `mu_linear_decrease_factor`** (read-only, Default = 0.2)
falcon.solver.ipopt/mu_linear_decrease_factor is a property.

**+ `mu_superlinear_decrease_power`** (read-only, Default = 1.5)
falcon.solver.ipopt/mu_superlinear_decrease_power is a property.

**+ `bound_frac`** (read-only, Default = 0.01)
falcon.solver.ipopt/bound_frac is a property.

**+ `bound_push`** (read-only, Default = 0.01)
falcon.solver.ipopt/bound_push is a property.

**+ `slack_bound_frac`** (read-only, Default = 0.01)
falcon.solver.ipopt/slack_bound_frac is a property.

**+ `slack_bound_push`** (read-only, Default = 0.01)
falcon.solver.ipopt/slack_bound_push is a property.

**+ `bound_mult_init_val`** (read-only, Default = 1)
falcon.solver.ipopt/bound_mult_init_val is a property.

**+ constr_mult_init_max** (read-only, Default = 1000)
falcon.solver.ipopt/constr_mult_init_max is a property.

**+ bound_mult_init_method** (read-only, Default = constant)
falcon.solver.ipopt/bound_mult_init_method is a property.

**+ LinearSolver** (read-only, Default = ma57)
The linear solver used with IPOPT.

**+ MuStrategy** (read-only, Default = adaptive)
The mu update strategy in IPOPT. Allowed values are given in the class constants MU_STRATEGY_MONOTONE and MU_STRATEGY_ADAPTIVE.

**+ MaxCPUTime** (read-only, Default = 1000000)
maximum cpu time in seconds

**+ CallsJ** (read-only, Default = 0)
The number of calls of the cost function

**+ CallsJgrad** (read-only, Default = 0)
The number of calls of the cost function gradient

**+ CallsGgrad** (read-only, Default = 0)
The number of calls of the constraint function gradient

**+ CallsG** (read-only, Default = 0)
The number of calls of the constraint function

**+ CallsH** (read-only, Default = 0)
The number of calls of the Hessian function

**+ doSolverWarmStart** (read-only)
If IPOPT is in WarmStart-mode or not.

**+ userIterFunc** (read-only)
iteration function of user

**+ Problem** (read-only)
The problem to be solved by this solver

**+ recalcZFVec** (read-only)
Flag to recalculate z and f vectors

**+ Options**
Struct keeping the main optimization options, being MajorOptTol, MinorOptTol, MajorFeasTol, MinorFeasTol, ComplTol, ActIdxTol, MajorIterLimit, MinorIterLimit, PrintLevel, OverwriteSol, needH, MCASamples, gPCOrder, sgrule, min_apprlevel.

**+ OptimizationResults**
falcon.solver.Optimizer/OptimizationResults is a property.

**+ output**
A struct holding the optimization output

### Methods

**ipopt** (Constructor)
Constructs a falcon.solver.ipopt object.

**> AnalyzeSolverResult**
Make some analysis on the (optimal) solver results.

**> CheckKKT**
Check the KKT conditions of the problem.

**> ParseConsoleOutput** (Static)
Extract information on the iterations of the optimization from the console output created.

**> setBarrierTolFactor**
Factor for mu in barrier stop test.

**> setFlagRecalculcZFVec**
Set the flag to recalculate the optimization parameter and residual vector.

**> setLinearSolver**
Sets the linear solver used in ipopt.

**> setMaximumCPUTime**
Sets the maximum cpu time (seconds) for ipopt. (http://www.coin-or.org/Ipopt/documentation/

**> setMuInit**
Sets the initial mu value.

**> setMuLinearDecreaseFactor**
Determines linear decrease rate of barrier parameter.

**> setMuMax**
Sets the maximum mu value.

**> setMuMaxFact**
Factor for initialization of maximum value for barrier parameter.

**> setMuMin**
Sets the minimum mu value.

**> setMuStrategy**
Sets the mu update strategy used in ipopt.

**> setMuSuperLinearDecreaseFactor**
Determines superlinear decrease rate of barrier parameter.

**> setMuTarget**
Sets the mu target value.

**> setProblem**
Set the problem to be solved.

**> setStandardStart**
Sets the standard bound and push start options.

**> setWarmStart**
Sets the warm start feature of ipopt.

**> Solve**
Solve the given optimal control problem using IPOPT

**> WarmStart**
Continue solving the project starting from the last iterate.

### Constructor falcon.solver.ipopt

Creates a new ipopt interface object used to numerically solve an optimal control problem. The problem can either directly be set, or can later be added using the method setProblem.

*Keywords*: Constructor Ipopt

#### - Syntax -

```
obj = ipopt()
obj = ipopt(Problem)
```

#### - Inputs -

**Problem** The problem to be solved using this numerical solver.

### Method AnalyzeSolverResult falcon.solver.ipopt

This function makes some analysis on the (optimal) results from the solver. It specifically checks KKT conditions, constraint fulfillment, scalings,...

*Keywords*: Optimizer Checks Analyze

#### - Syntax -

```
obj.AnalyzeSolverResult()
```

#### - Name Value -

**checkGradient** Makes a gradient check by comparison to finite differences (default: false).

**checkScaling** Makes a scaling check (default: false).

**doSimulation** Simulate the problem with the optimal control history and find e.g., numerical instabilities or stiff integrations (default: false).

### Method CheckKKT falcon.solver.ipopt

Calculate the Jacobian of the Lagrange function and extract the largest value. This is an approximate KKT condition check.

*Keywords*: Optimizer Checks KKT

#### - Syntax -

```
dLdz = obj.CheckKKT()
dLdz = obj.CheckKKT('Name', Value)
```

#### - Name Value -

**lambda** The multipliers for the constraints f in the problem. (default: the values from the optimal solution, if the problem was already solved.)

**zl** The multipliers for the lower bounds of z. (default: the values from the optimal solution, if the problem was already solved.)

**zu** The multipliers for the upper bounds of z. (default: the values from the optimal solution, if the problem was already solved.)

**mu** The combined multipliers for the bounds of z: mu = -zl + zu. (default: the values from the optimal solution, if the problem was already solved.)

#### - Outputs -

**dLdz** The Jacobian of the Lagrange function with respect to the parameter vector z.

### Method ParseConsoleOutput falcon.solver.ipopt

Parse the console output created by the optimization and automatically analyze it. The resulting data on the iteration history is returned in a struct.

*Keywords*: Ipopt Parser

#### - Syntax -

```
data = falcon.solver.ipopt.ParseConsoleOutput(str)
```

#### - Outputs -

**data** The data struct containing information about the iteration history while solving the problem.

### Method setBarrierTolFactor falcon.solver.ipopt

The convergence tolerance for each barrier problem in the monotone mode is the value of the barrier parameter times "barrier tol factor". This option is also used in the adaptive mu strategy during the monotone mode. (This is kappa epsilon in implementation paper). The valid range for this real option is 0 < barrier tol factor < inf and its default value is 10.

*Keywords*: Ipopt Settings Mu Barrier Tolerance

*- Syntax -*

```
obj.setBarrierTolFactor(Value)
```

*- Inputs -*

**Value** The numerical barrier tolerance factor value

### Method setFlagRecalculcZFVec falcon.solver.ipopt

When the flag is true, the z and f vectors are recalculated each time the solve command is invoked. This allows fast initial guess studies or different bounds. It should be noted that the general problem is not allowed to change.

*Keywords*: Optimizer RecalcZFFlag

*- Syntax -*

```
obj.setFlagRecalculcZFVec(flag)
```

*- Inputs -*

**flag** Flag to recalculate z and f vector (default: false).

### Method setLinearSolver falcon.solver.ipopt

Sets the linear solver used by ipopt to solve the NLP.

*Keywords*: Ipopt Settings Linear Solver

*- Syntax -*

```
obj.setLinearSolver(LinSolver)
```

*- Inputs -*

**LinSolver** A char specifiying the linear solver. The default is ma57.

### Method setMaximumCPUTime falcon.solver.ipopt

Sets the maximum cpu time in the ipopt instance used here.

*Keywords*: Ipopt Settings CPU Time

*- Syntax -*

```
obj.setMaximumCPUTime(Seconds)
```

*- Inputs -*

**Seconds** The maximum cpu time ipopt is allowed to use to solve the problem. Limit is checked during conversion check.

### Method setMuInit falcon.solver.ipopt

Sets the initial mu value, i.e., the iteration start point.

*Keywords*: Ipopt Settings Mu Initial

*- Syntax -*

```
obj.setMuInit(Target)
```

*- Inputs -*

**Target** The numerical initial value

### Method setMuLinearDecreaseFactor falcon.solver.ipopt

For the Fiacco-McCormick update procedure the new barrier parameter mu is obtained by taking the minimum of mu times "mu linear decrease factor" and mu"superlinear decrease power". (This is kappa mu in implementation paper.) This option is also used in the adaptive mu strategy during the monotone mode. The valid range for this real option is 0 < mu linear decrease factor < 1 and its default value is 0.2.

*Keywords*: Ipopt Settings Mu Linear Decrease

*- Syntax -*

```
obj.setMuLinearDecreaseFactor(Value)
```

*- Inputs -*

**Value** The numerical linear barrier decrease value

### Method setMuMax falcon.solver.ipopt

Sets the maximum mu value, i.e., the upper bound of the barrier parameter (mainly for adaptive strategies)

*Keywords*: Ipopt Settings Mu Maximum

*- Syntax -*

```
obj.setMuMax(Value)
```

*- Inputs -*

**Value** The numerical maximum value

### Method setMuMaxFact falcon.solver.ipopt

This option determines the upper bound on the barrier parameter. This upper bound is computed as the average complementarity at the initial point times the value of this option. (Only used if option "mu strategy" is chosen as "adaptive".) The valid range for this real option is 0 < mu max fact < inf and its default value is 1000.

*Keywords*: Ipopt Settings Mu Maximum Factor

*- Syntax -*

```
obj.setMuMaxFact(Value)
```

*- Inputs -*

**Value** The numerical maximum factor value

### Method setMuMin falcon.solver.ipopt

Sets the minimum mu value, i.e., the lower bound of the barrier parameter (mainly for adaptive strategies)

*Keywords*: Ipopt Settings Mu Minimum

#### - Syntax -

```
obj.setMuMin(Value)
```

#### - Inputs -

**Value** The numerical minimum value

### Method setMuStrategy falcon.solver.ipopt

Sets the mu update strategy in the ipopt instance used here.

*Keywords*: Ipopt Settings Mu Strategy

#### - Syntax -

```
obj.setMuStrategy(Strategy)
```

#### - Inputs -

**Strategy** The mu update strategy to be used for solving the problem. Supported values can be found in the constants MU_STRATEGY_MONOTONE and MU_STRATEGY_ADAPTIVE in this class.

### Method setMuSuperLinearDecreaseFactor falcon.solver.ipopt

For the Fiacco-McCormick update procedure the new barrier parameter mu is obtained by taking the minimum of mu times "mu linear decrease factor" and mu"superlinear decrease power". (This is theta mu in implementation paper.) This option is also used in the adaptive mu strategy during the monotone mode. The valid range for this real option is $1 <$ mu superlinear decrease power $< 2$ and its default value is 1.5.

*Keywords*: Ipopt Settings Mu Superlinear Decrease

#### - Syntax -

```
obj.setMuSuperLinearDecreaseFactor(Value)
```

#### - Inputs -

**Value** The numerical superlinear barrier decrease value

### Method setMuTarget falcon.solver.ipopt

Sets the mu target value, i.e., the value of that defines to which extend the complementary slackness conditions must be fulfilled to view a constraint as "fulfilled". A larger value leads to an easier to solve problem, but might be unphysical.

*Keywords*: Ipopt Settings Mu Target

*- Syntax -*

```
obj.setMuTarget(Target)
```

*- Inputs -*

**Target** The numerical target value

### Method setProblem falcon.solver.ipopt

Sets the optimal control problem to be numerically solved using this solver.
*Keywords*: Optimizer Problem

*- Syntax -*

```
obj.setProblem(Problem)
```

*- Inputs -*

**Problem** The problem to be solved using this numerical solver.

### Method setStandardStart falcon.solver.ipopt

Resets the values for the standard bound values in ipopt (default values as in ipopt manual).
*Keywords*: Ipopt Standard Start

*- Syntax -*

```
obj.setStandardStart('Name',Value)
```

*- Name Value -*

**bound_frac** Desired minimum absolute distance from the initial point to bound (together with "bound push").

**bound_push** Desired minimum absolute distance from the initial point to bound (together with "bound frac").

**slack_bound_frac** Desired minimum relative distance from the initial slack to bound(together with "slack bound push").

**slack_bound_push** Desired minimum relative distance from the initial slack to bound(together with "slack bound frac").

**bound_mult_init_val** Initial value for the bound multipliers.

**constr_mult_init_max** Maximum allowed least-square guess of constraint multipliers.

**bound_mult_init_method** Initialization method for bound multipliers.

### Method setWarmStart falcon.solver.ipopt

Enables or disables the warm start feature of IPOPT. Sets the flag to specify warm start bounds and relaxation.

*Keywords*: Ipopt Warm Start

#### - Syntax -

```
obj.setWarmStart(flag, 'Name',Value)
```

#### - Inputs -

**flag** A bool, enabling or disabling the warmstart feature of IPOPT.

#### - Name Value -

**WarmStartBoundPush** same as bound push for the regular initializer.

**WarmStartBoundFrac** same as bound frac for the regular initializer.

**WarmStartSlackBoundFrac** same as slack bound frac for the regular initializer.

**WarmStartSlackBoundPush** same as slack bound push for the regular initializer.

**WarmStartMultBoundPush** same as mult bound push for the regular initializer.

**WarmStartMultInitMax** Maximum initial value for the equality multipliers.

**BoundRelaxFactor** Factor for initial relaxation of the bounds.

**mu_init** Initial value for the barrier parameter.

### Method Solve falcon.solver.ipopt

Solve the given optimal control problem numerically using the numerical solver ipopt.

*Keywords*: Ipopt Solve

#### - Syntax -

```
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.Solve()
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.Solve(zInitial)
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.Solve(..., 'Name',
    Value)
```

#### - Name Value -

**zInitial** The initial parameter vector to start the solution.

**lambda** The initial Lagrange multipliers

**zl** The initial multipliers for the lower constraints on the parameter vector z.

**zu** The initial multipliers for the upper constraints on the parameter vector z.

*- Outputs -*

**z_opt** If the problem converged, the optimal parameter vector for the problem, otherwise the current iterate.

**F_opt** If the problem converged, the optimal constraint vector for the problem, otherwise the current iterate.

**status** The status of the optimization. Contains the stopping criteria.

**lambda** If the problem converged, the optimal Lagrange multipliers for the constraints of problem, otherwise the current iterate.

**mu** If the problem converged, the optimal Lagrange multipliers for the box constraints on z of the problem, otherwise the current iterate.

**zl** If the problem converged, the optimal Lagrange multipliers for the lower bounds of the box constraints on z of the problem, otherwise the current iterate.

**zu** If the problem converged, the optimal Lagrange multipliers for the upper bounds of the box constraints on z of the problem, otherwise the current iterate.

**IterationFunction** The iteration function that should be called in each Ipopt iteration callback specified by the user. The function requires exactly three inputs and one output: function b = iterfunc(nIter, f, auxdata)

### Method WarmStart falcon.solver.ipopt

Solve the given optimal control problem numerically using the numerical solver ipopt.
The WarmStartMode of IPOPT is not changed within this function. Try changing obj.setWarmStart() to true in case you have problems warm starting the solver.

*Keywords*: Ipopt Warm Start

*- Syntax -*

```
[z_opt, F_opt, status, lambda, mu, zl, zu] = obj.WarmStart('Name',Value)
```

*- Name Value -*

**zInitial** Initial guess for the optimization variables.

**lInitial** Initial guess for the constraint multiplier.

**zlInitial** Initial guess for the lower bound multiplier of the optimization parameter.

**zuInitial** Initial guess for the upper bound multiplier of the optimization parameter.

*- Outputs -*

`z_opt` If the problem converged, the optimal parameter vector for the problem, otherwise the current iterate.

`F_opt` If the problem converged, the optimal constraint vector for the problem, otherwise the current iterate.

`status` The status of the optimization. Contains the stopping criteria.

`lambda` If the problem converged, the optimal Lagrange multipliers for the constraints of problem, otherwise the current iterate.

`mu` If the problem converged, the optimal Lagrange multipliers for the box constraints on z of the problem, otherwise the current iterate.

`zl` If the problem converged, the optimal Lagrange multipliers for the lower bounds of the box constraints on z of the problem, otherwise the current iterate.

`zu` If the problem converged, the optimal Lagrange multipliers for the upper bounds of the box constraints on z of the problem, otherwise the current iterate.

# 6 Derivative Construction

Why is a construction necessary?

- FALCON.m uses gradient base optimization algorithms to solve the problems, therefore the gradient of the dynamic models, constraints and cost functions are required. To achieve this, models, constraints and cost functions are preprocessed to return derivatives together with the regular outputs. This is a unique feature that differentiates FALCON.m from many other tools. However, this preprocessing step is one of the main reason why FALCON.m is very fast. FALCON.m handles all derivative generation automatically!

- FALCON.m can calculate first and second order derivatives either analytically or using finite differences. If the Symbolic Math Toolbox is not present, FALCON.mswitches to compatibility model (finite differences) automatically. However, dependent on the dynamic model or constraints, finite differences may be substantially slower.

- The generated models / constraints need to be evaluated many times. Therefore, for fast evaluation, FALCON.m generates C/C++ code which is compiled to a mex file to ensure fastest possible evaluation. In case the MATLAB Coder is not present, FALCON.m automatially switches to compatibility mode by evaluating the model / constraint within a for-loop. For complex dynamic models, the evaluation in a compiled mex file is substantially faster.

- After preprocessing, FALCON.mcreates an additional MATLAB or mex file (dependent on the evaluation mode mex or matlab) in the current working directory. These files implement functions which are passed to the passed to FALCON.m in `problem.addNewPhase`, `phase.addNewPathFunction`, and so on.

  Please note: In the quickstart example (see **??**) not the preprocessed model but the MATLAB file containing the source model was given to the new phase. If this is the case, FALCON.m will automatically try to preprocess the given source file. However, this method is recommended only if the optimal control problem is very simple.

There are two main ways to preprocess the dynamic models / constraints and cost functions.

**Function Mode** In this case the dynamic model / constraint or cost function is defined by a single MATLAB source function. This is the preferred way and can be easily achieved. (usage of the Function Mode is described in section 6.1)

**Subsystem Mode** This mode is important especially for large dynamic models using analytic derivatives. If the model becomes very large, at some point, the Symbolic Math Toolbox cannot calculate the analytic derivatives anymore. Models become large if the differentiation takes longer than a minute (RULE OF THUMB!). The following properties define a "large" model / constraint if it contains:

- multiple matrix vector multiplications
- multiple high order polynomials
- non-continuities such as lookup tables

In this case, the dynamic model can usually be split into smaller subsystems which can be differentiated locally, giving the Subsystem Mode its name. (usage of the Subsystem Mode is described in section **??**)

All models / constraints and cost functions are created using the builder classes in FALCON.m. These are

- `falcon.SimulationModelBuilder` for dynamic models
- `falcon.PathConstraintBuilder` for path constraints
- `falcon.PointConstraintBuilder` for point constraints and cost functions

In the following, the builder classes are explained in more detail. All builders support the function mode and the subsystem mode.

## 6.1 Function Mode

The function mode is invoked by passing the function handle to the builder in the constructor. See the documentation of Simulation Model Builder, Path Constraint Builder and Point Constraint Builder for more details.

## 6.2 System Mode

Large high fidelity models cannot be differentiated by the symbolic math toolbox directly. Therefore, FALCON.m offers the subsystem mode for model generation. The basic idea is that the user splits the dynamic model / constraint / cost function into simpler subsystems. The subsystems are implemented as MATLAB functions. FALCON.m will automatically create the derivatives for the whole model / constraint or cost function. In the following the principles of the subsystem mode and the method provided by the builder classes are described. Please note that for better understanding only the basic principles are presented. See the actual documentation of the classes for detailed information.

For simplicity the explanation will be given with an application to a dynamic model in mind. All principles and methods described can be transfered to constraints and cost functions too.

### 6.2.1 Principles

Apart from the subsystems, the user needs to define how these are connected. In FALCON.m every signal / variable is represented by a string. For every variable available in the model, FALCON.m stores its size (scalar, row column vector or matrix).

```
1 states   = [falcon.State('x'), falcon.State('y')];
2 controls = [falcon.Control('V'), falcon.Control('alpha')];
3
4 mdl = falcon.SimulationModelBuilder('name', states, controls);
5 mdl.addSubsystem(@myfunc,...          % Subsystem Function Handle
6     {'x', 'y', 'V', 'alpha'},...      % Inputs  to Subsystem
7     {'xdot', 'ydot'})                 % Outputs of Subsystem
8 mdl.setStateDerivativeNames({'xdot', 'ydot'});
9 mdl.Build();
```

In the example above the states and controls are defined by an array of `falcon.State` and `falcon.Control` objects which are passed to the constructor of the `falcon.SimulationMod` instance. Within the builder all states and controls will be registered as individual available scalar variables. The user can now add an arbitrary number of subsystems to the model. For every subsystem the source function (function handle or anonymous function, first argument), input arguments (cell array of strings, second argument) and the output arguments (cell array of strings, third arguments) have to be specified. The method `setStateDerivativeNames` tells the builder which variables hold the state derivative information. Every construction is finalized by the `Build` command.

Parameters and Outputs are registered to the builder in the same way. Additionally, constants can be defined.

### 6.2.2 Constants

There are basically three ways how constants can be used in the subsystem mode.

**addConstantInput** This method of the builder instance adds an additional input to the model dynamics. The name as string as well as the size needs to be specified. Use

this method if a constant in the model shall be changeable after the generation of the model derivatives.

**addConstant** This method adds an internal constant to the list of variable that cannot be altered after the construction of the model. The name as a string as well as the constant value needs to be provided. Use this method is a constant in the model shall be created which is reused in different subsystems.

**Numeric Variable** Apart from strings, subsystem inputs be also be numeric variables (scalar, vector or matrix). Use this feature if a constant input to a subsystem has many zero entries. Thus, especially in the analytic derivative generation, the Symbolic Math Toolbox can highly optimize the code. See the builders addSubsystem method for more information.

### 6.2.3 Subsystems

Subsystems are added to a model using the `addSubsystem` method. A subsystem can be a

- function handle to a MATLAB function
- anonymous function handle

NOT supported are: MATLAB builtin functions, nested functions, local functions (e.g. below class definition). If you wish to use any of these function you can do so by wrapping it with an anonymous function handle.

During the build process FALCON.m creates the derivatives of the source functions. For every subsystem source function a fingerprint value (hash value) is generated. This speeds up the derivative generation process if a small change was made and the model is reconstructed. Subsystems that have not changed will not get their derivatives recalculated. The fingerprint is calculated only for the top-level function, meaning the function behind the function handle or anonymous functions. Any other subfunctions called by these are not taken into account. In order to force a new generation of the derivatives, delete the `fm_models` and `fm_constraints` folder in current working directory.

Derivative Subsystems can be used in case a function cannot be differentiated analytically (e.g. table data, minor discontinuities). Using the `addDerivativeSubsystem` method it is possible to add any kind of subsystem to the model. However, in this case the derivative need to be supplied by the user (e.g. using finite differences).

### 6.2.4 Variable Manipulation

It often occurs that a variable is available as a vector but individual values are required. On the other hand, sometimes variables need to be stiched together (e.g. to form a matrix or vector). For these cases, the subsystem derivative builder in FALCON.m offers two methods:

**SplitVariable** Splits a matrix or vector into subparts

**CombineVariables** Combines multiple variables into a single new variable.

### 6.2.5 Important Remarks

- The use of global variables in subsystems has not been tested throughoutly. After the built process it is very likely that global variable is no longer available within the subsystem.

## 6.3 Simulation Model Builder

Parent Classes: falcon.core.builder.BaseBuilder

**Properties**

**+ `HasOutputs`** (Dependent)
Flag if the Simulation Model has Outputs

**+ `DERIVATIVE_ANALYTIC`** (Constant, read-only, Default = analytic)
Flag for setting builder to analytic derivative mode.

**+ `DERIVATIVE_FINITE_DIFFERENCE`** (Constant, read-only, Default = finite_difference)
Flat for setting builder to finite difference derivative mode.

**+ `EVALUATION_MEX`** (Constant, read-only, Default = mex)
Flag for settin builder to mex evaluation mode.

**+ `EVALUATION_MATLAB`** (Constant, read-only, Default = matlab)
Flag for setting builder to matlab evaluation mode.

**+ `EVALUATION_NONE`** (Constant, read-only, Default = none)
Flag for setting builder to no evaluation mode. (No wrapper is created)

**+ `TYPE_OUTPUT`** (Constant, read-only, Default = OUTPUT)
falcon.core.builder.BaseBuilder.TYPE_OUTPUT is a property.

**+ `TYPE_STATE`** (Constant, read-only, Default = STATE)
falcon.core.builder.BaseBuilder.TYPE_STATE is a property.

**+ `TYPE_CONTROL`** (Constant, read-only, Default = CONTROL)
falcon.core.builder.BaseBuilder.TYPE_CONTROL is a property.

**+ `TYPE_PARAMETER`** (Constant, read-only, Default = PARAMETER)
falcon.core.builder.BaseBuilder.TYPE_PARAMETER is a property.

**+ `TYPE_VALUE`** (Constant, read-only, Default = VALUE)
falcon.core.builder.BaseBuilder.TYPE_VALUE is a property.

**+ TYPE_DISCRETE** (Constant, read-only, Default = DISCRETE)
falcon.core.builder.BaseBuilder.TYPE_DISCRETE is a property.

**+ TYPE_CONSTANT** (Constant, read-only, Default = CONSTANT)
falcon.core.builder.BaseBuilder.TYPE_CONSTANT is a property.

**+ ProjectName** (read-only)
Name of the model or function project

**+ SimpleFunctionHandle** (read-only)
Holds the handle if a single function is used to define the function or Model / Constraint or Cost

**+ OptimizeCode** (read-only)
Perform Code Optimization

**+ isBuilt** (read-only)
Flag that determined if the project was already build

**+ isSimulinkModel** (read-only)
Flag that determined if the model is a Simulink model

**+ Handle** (Dependent)
Handle to the constructed model / constraint function.

### Methods

**SimulationModelBuilder** (Constructor)
Class to construct dynamic models in falcon.

**> addConstant**
Add a internal constant to the project

**> addConstantInput**
Add a constant input to the dynamic model.

**> addDerivativeSubsystem**
Add Subsystem which already provides derivatives to the project

**> addSubsystem**
Add Subsystem to the project to create its derivatives.

**> Build**
Builds the current project

**> CheckDerivatives**
Check Derivatives of the generated project

**> CombineVariables**
Combine multiple variables to a single variable

**> setOutputs**
Set the output of the model

**> setStateDerivativeNames**
Set the state derivative names used in subsystem mode.

**> SimpleModeOutputVariableProcessing** (Static)
falcon.core.builder.BaseBuilder.SimpleModeOutputVariableProcessing is a function.
arr = SimpleModeOutputVariableProcessing(arr)

**> SplitVariable**
Split a single variable into multiple parts

### Constructor

*Keywords*: Constructor Model Builder

*- Syntax -*

```
obj = falcon.SimulationModelBuilder(ProjectName, States)
obj = falcon.SimulationModelBuilder(ProjectName, States, Controls)
obj = falcon.SimulationModelBuilder(ProjectName, States, Controls,
    Parameters)
obj = falcon.SimulationModelBuilder(ProjectName, States, Controls,
    Parameters, Handle)
obj = falcon.SimulationModelBuilder(ProjectName, States, 'Name', Value)
obj = falcon.SimulationModelBuilder(..., 'Name', Value)
```

*- Inputs -*

**ProjectName** The name of the to be generated model. This is the filename of the generated model.

**States** State input of the model. Column vector of falcon.State objects or integer for number of states.

**Controls** Control input of the model. Column vector of falcon.Control objects or integer for number of controls. Use [] or 0 to set no controls. (default: [])

**Parameters** Parameter input of the model. Column vector of falcon.Parameter objects or integer for number of parameters. Use [] or 0 to set no parameters. (default: [])

**Handle** Function Handle for models that are described using a single matlab function (Function Mode). Leave empty if you want to construct a model using subsystems (Subsystem Mode). (default: [])

*- Name Value -*

**DerivativeMode** Flag that defines if the derivatives are calculated using symbolic differentiation ('analytic') or using finite differences('finite_difference'). (default = 'analytic')

**Optimize** Set the Optimization option for symbolic differentiation. Only available in MATLAB 2014b or later. (Function Mode default=false, Subsystem Model default = true)

**DoDependencyCheck** Flag that enables a check if a subsystem is dependent on other subsystems. (default = false)

*- Outputs -*

**obj** The falcon.SimulationModelBuilder instance.

The models used in `falcon.SimulationModelBuilder` can be of the form

$$\dot{x} = f(x, u, p, c_1, c_2, \dots) \tag{53}$$
$$y = h(x, u, p, c_1, c_2, \dots) \tag{54}$$

where $f$ implements the state derivatives $\dot{x}$ and $h$ additional model outputs $y$. The implementation of the model outputs is optional. $c_1, \dots$ are additional constant inputs (see 6.3). Controls $u$, parameters $p$ and constants $c$ are optional and may not appear in the model interface. In this case the model dynamics simplifies to $f(x)$. However, the order of the model inputs must hold, e.g. $f(p, x)$ is not possible and has to be $f(x, p)$.

### Method setOutputs

Set the size of outputs of the model. This information is used for the construction of the derivatives and wrapper file. In case of subsystem mode the actual names of the outputs have to be provided using falcon.Output.

*Keywords*: Model Builder Outputs

*- Syntax -*

```
obj.setOutputs(outputs)
```

*- Inputs -*

**outputs** numeric value specifying the number of outputs (simple mode only). Alternatively use falcon.Output column vector to set the size and names of outputs (required for subsystem mode).

### Method addConstantInput

Additional constant inputs to the model / constraint can be set using this function. They will be added in order of occurence after the main input sequence f(x,u,p,c1,c2,...). These constant inputs can be set and changed externally and are not hard-coded. This

makes testing different model types efficient. They must be added to the model in the problem using the setConstant-method and thus, only the size is specified here.

*Keywords*: Base Builder Constant Input

*- Syntax -*

```
obj.addConstantInput(Name)
obj.addConstantInput(Name, 'Name', Value)
```

**Name** Name of input. (string)

**VariableSize** Either a numeric dimension [m,n] (must be 1 by 2 row vector) or a cell array of string specifying multiple [1,1] entries to the model. This is the size used for each non-dimensional time step.

### Method addConstant (Subsystem Mode)

Set constant values in Subsystem Mode. Values are internal and cannot be influence from the outside. For additional inputs use the addConstantInput method. This method throws an error if called in Function Mode.

*Keywords*: Base Builder Constant

*- Syntax -*

```
obj.addConstant(Name, Value)
```

*- Inputs -*

**Name** Name of the constant. (string)

**Value** Value of the constant. (numeric, scalar, vector or matrix).

### Method addSubsystem (Subsystem Mode)

Add a subsystem to the model / constraint. The subsystems are chained in order of appearance. Therefore, it is crucial that all data required by the subsystem is already present (states, controls, outputs of other subsystems).

*Keywords*: Base Builder Subsystem

*- Syntax -*

```
obj.addSubsystem(Subsys, Inputs, Outputs)
obj.addSubsystem(Subsys, 'Inputs', Inputs, 'Outputs', Outputs)
obj.addSubsystem(Subsys, {matrix, 'varstr', {'a','b';'c','d'}})
obj.addSubsystem(.., 'Name', Value)
```

*- Inputs -*

**Subsys** anonymous function, simple function handle or matlab.System class instance.

**Inputs** Input arguments cell array. Entries in the cell can either be numeric (scalar, vector, matrix), a variable string, cell array of variable strings (variables in cell array must be concatable).

**Outputs** Output arguments cell array. Entries in the cell can either be a variable string, a cell array of variable strings. In the latter case the size of the cell array must fit the output size. Additionally, a ' ' can be used to ignore an output.

*- Name Value -*

**Optimize** Flag that sets the optimization option for the derivative creation (analytic derivative mode only). (default = true)

### Method addDerivativeSubsystem (Subsystem Mode)

Adds a subsystem to the subsystem chain of the project which already calculates derivatives. This enables the use of lookup tables or similar function in the subsystem chain. A funtion handel to the subsystem, inputs and outputs have to be specified. In case Name Value pairs are not set (OutputSizes) FALCON.m uses a nan call to the function to determine the output sizes, jacobian (and hessian) sparsity structure. If the function call cannot handle nan inputs, output sizes and sparsity patterns have to be provided.

*Keywords*: Base Builder Derivative Subsystem

*- Syntax -*

```
obj.addDerivativeSubsystem(Subsystem, Inputs, Outputs)
obj.addDerivativeSubsystem(Subsystem, 'Inputs', Inputs, 'Outputs',
    Outputs)
obj.addDerivativeSubsystem(.., 'Name', Value)
```

*- Inputs -*

**Subsystem** Must be a simple function handle (anonymous functions or matlab.System classes are not supported)

**Inputs** Input arguments cell array. Entries in the cell can either be numeric, a variable string, cell array of variable strings. Constant inputs (numeric, constant values) must not have their derivatives returned by the derivative subsystem.

**Outputs** Output arguments cell array. Entries in the cell can either be a variable string. Cell array of variable strings are not supported. Ignoring an output using ' ' is not supported.

*- Name Value -*

The following name value pairs are optional, but in case on is set the all relevant information has to be given.

**OutputSizes** The size of each output value.

**OutputJacobianSparsity** The sparsity pattern of the output jacobian given as a matrix of zeros and ones.

**OutputHessianSparsity** The sparsity pattern of the output hessian given as a matrix of zeros and ones.

### Method setStateDerivativeNames (Subsystem Mode)

This function sets the names of the state derivatives for the subsystem mode. These names are used to build the state derivatives correctly. For simple mode this function will cause an exception.

*Keywords*: Model Builder Deriv Names

*- Syntax -*

```
obj.setStateDerivativeNames(names)
```

**names** cell array of strings or single string (one state dynamic models only)

### Method SplitVariable (Subsystem Mode)

Split large variables into smaller heaps. Since the method uses mat2cell interally the block structure and summation of rows and columns must fit. If the size of entries is the same as the size of the variable name, rowsplit and colsplit do not have to be provided. Otherwise the sum of rowsplit and sum of colsplit must fit the size of the variable name respectively. Not available in SimpleMode.

*Keywords*: Base Builder Variables Split

*- Syntax -*

```
obj.SplitVariable(name, entries)
obj.SplitVariable(name, entries, rowsplit, colsplit)
```

*- Inputs -*

**name** name of original variable

**entries** name of new entries, which is orientation sensitive.

**rowsplit** row distribution

**colsplit** column distribution

### Method CombineVariables (Subsystem Mode)

Combine multiple variables to a single variable to simplify the construction code. Not available in SimpleMode.

*Keywords*: Base Builder Variables Combine

*- Syntax -*

```
obj.CombineVariables(name, vars)
```

*- Inputs -*

**name** Name of the new variable

**vars** Cell array of strings. vars is orientation sensitive, meaning 'a', 'b', 'c' and 'a'; 'b'; 'c' will create different variables. Variables must have a matching block structure (see mat2cell).

### Method Build

Builds the current project, which means the derivative function interface is constructed. Afterwards the evaluation function is created. Additional settings for the evaluation function can be set.

*Keywords*: Base Builder Build

*- Syntax -*

```
handle = obj.Build()
handle = obj.Build('Name', Value)
```

*- Name Value -*

**EvaluationProvider** Parameter Value Pair. 'mex' generates a c++ mex file wrapper and mex function. 'none' will prevent the construction of the evaluation wrapper. Any other input invokes a MATLAB wrapper. (default = 'mex')

**MultiThreading** Flag to compile the model with multi-threading (default: false)

**OutputFolder** Folder to which the compiled/generated model is going to be saved (default: pwd).

*- Outputs -*

**handle** see get.Handle. In case 'none' was chosen for the evaluation provider, handle is empty []

## 6.4 Path Constraint Builder

Parent Classes: falcon.core.builder.BaseBuilder

### Properties

**+ DERIVATIVE_ANALYTIC** (Constant, read-only, Default = analytic)
Flag for setting builder to analytic derivative mode.

**+ DERIVATIVE_FINITE_DIFFERENCE** (Constant, read-only, Default = finite_difference)
Flat for setting builder to finite difference derivative mode.

**+ EVALUATION_MEX** (Constant, read-only, Default = mex)
Flag for settin builder to mex evaluation mode.

**+ EVALUATION_MATLAB** (Constant, read-only, Default = matlab)
Flag for setting builder to matlab evaluation mode.

**+ EVALUATION_NONE** (Constant, read-only, Default = none)
Flag for setting builder to no evaluation mode. (No wrapper is created)

**+ TYPE_OUTPUT** (Constant, read-only, Default = OUTPUT)
falcon.core.builder.BaseBuilder.TYPE_OUTPUT is a property.

**+ TYPE_STATE** (Constant, read-only, Default = STATE)
falcon.core.builder.BaseBuilder.TYPE_STATE is a property.

**+ TYPE_CONTROL** (Constant, read-only, Default = CONTROL)
falcon.core.builder.BaseBuilder.TYPE_CONTROL is a property.

**+ TYPE_PARAMETER** (Constant, read-only, Default = PARAMETER)
falcon.core.builder.BaseBuilder.TYPE_PARAMETER is a property.

**+ TYPE_VALUE** (Constant, read-only, Default = VALUE)
falcon.core.builder.BaseBuilder.TYPE_VALUE is a property.

**+ TYPE_DISCRETE** (Constant, read-only, Default = DISCRETE)
falcon.core.builder.BaseBuilder.TYPE_DISCRETE is a property.

**+ TYPE_CONSTANT** (Constant, read-only, Default = CONSTANT)
falcon.core.builder.BaseBuilder.TYPE_CONSTANT is a property.

**+ ProjectName** (read-only)
Name of the model or function project

**+ SimpleFunctionHandle** (read-only)
Holds the handle if a single function is used to define the function or Model / Constraint or Cost

**+ OptimizeCode** (read-only)
Perform Code Optimization

**+ isBuilt** (read-only)
Flag that determined if the project was already build

**+ isSimulinkModel** (read-only)
Flag that determined if the model is a Simulink model

**+ Handle** (Dependent)
Handle to the constructed model / constraint function.

### Methods

**PathConstraintBuilder** (Constructor)
Constructs analytic falcon.PathConstraintBuilder object for path constraint generation.

**> addConstant**
Add a internal constant to the project

**> addConstantInput**
Add a constant input to the dynamic model.

**> addDerivativeSubsystem**
Add Subsystem which already provides derivatives to the project

**> addSubsystem**
Add Subsystem to the project to create its derivatives.

**> Build**
Builds the current project

**> CheckDerivatives**
Check Derivatives of the generated project

**> CombineVariables**
Combine multiple variables to a single variable

**> setConstraintValueNames**
Set the constraint value names for subsystem mode.

**> SimpleModeOutputVariableProcessing** (Static)
falcon.core.builder.BaseBuilder.SimpleModeOutputVariableProcessing is a function.
arr = SimpleModeOutputVariableProcessing(arr)

**> SplitVariable**
Split a single variable into multiple parts

### Constructor

*Keywords*: Constructor Path Constraint

*- Syntax -*

```
obj = falcon.PathConstraintBuilder(Name, ModelOutputs)
obj = falcon.PathConstraintBuilder(Name, ModelOutputs, States)
obj = falcon.PathConstraintBuilder(Name, ModelOutputs, States, Controls)
obj = falcon.PathConstraintBuilder(Name, ModelOutputs, States,
    Controls, Parameters)
obj = falcon.PathConstraintBuilder(Name, ModelOutputs, States,
    Controls,
    Parameters, Handle)
obj = falcon.PathConstraintBuilder(Name, ModelOutputs, States, 'Name',
    Value)
obj = falcon.PathConstraintBuilder(..., 'Param', Value)
```

*- Inputs -*

**Name** The name of the to be generated constraint.

**ModelOutputs** Output input for the constraint. Column vector of falcon.Constraints or integer for number of outputs.

**States** State input for the constraint. Column vector of falcon.States or integer for number of states. Use [] to set no states. (default: [])

**Controls** Control input for the constraint. Column vector of falcon.Controls or integer for number of controls. Use [] to set no controls. (default: [])

**Parameters** Parameter input for the constraint. Column vector of falcon.Parameters or integer for number of parameters. Use [] to set no parameters. (default: [])

**Handle** Function Handle for constraint that are described using a single function. Use [] if you want to construct a constraint using subsystems. (default: [])

The path constraints used in `falcon.PathConstraintBuilder` can be of the form

$$g(y, x, u, p, c_1, c_2, \dots) \tag{55}$$

Please note the following:

- All inputs of $g$ are optional (outputs $y$, states $x$, controls $u$, parameters $p$ and constant inputs $c_1, \dots$). However, at least one of them has to enter the dynamic path constraint and the order has to be fulfilled. (e.g. $g(p, y)$ is not possible).
- Normally, path constraint do not require all outputs, states, controls and parameters. Therefore, only the `falcon.Constraint`, `falcon.State`, `falcon.Control` and `falcon.Parameter` objects that are required in the path function need to be specified. During optimization, FALCON.m automatically, extracts the correct values from the optimal control problem. If just the number of e.g. outputs is specified, the number must be the same as the number of outputs of the dynamic model.
- Additional constant inputs can be specified using as described in **??**.

### Method addConstantInput

Additional constant inputs to the model / constraint can be set using this function. They will be added in order of occurence after the main input sequence f(x,u,p,c1,c2,...). These constant inputs can be set and changed externally and are not hard-coded. This makes testing different model types efficient. They must be added to the model in the problem using the setConstant-method and thus, only the size is specified here.

*Keywords*: Base Builder Constant Input

#### *- Syntax -*

```
obj.addConstantInput(Name)
obj.addConstantInput(Name, 'Name', Value)
```

**Name** Name of input. (string)

**VariableSize** Either a numeric dimension [m,n] (must be 1 by 2 row vector) or a cell array of string specifying multiple [1,1] entries to the model. This is the size used for each non-dimensional time step.

### Method addConstant (Subsystem Mode)

Set constant values in Subsystem Mode. Values are internal and cannot be influence from the outside. For additional inputs use the addConstantInput method. This method throws an error if called in Function Mode.

*Keywords*: Base Builder Constant

#### - Syntax -

```
obj.addConstant(Name, Value)
```

#### - Inputs -

**Name** Name of the constant. (string)

**Value** Value of the constant. (numeric, scalar, vector or matrix).

### Method addSubsystem (Subsystem Mode)

Add a subsystem to the model / constraint. The subsystems are chained in order of appearance. Therefore, it is crucial that all data required by the subsystem is already present (states, controls, outputs of other subsystems).

*Keywords*: Base Builder Subsystem

#### - Syntax -

```
obj.addSubsystem(Subsys, Inputs, Outputs)
obj.addSubsystem(Subsys, 'Inputs', Inputs, 'Outputs', Outputs)
obj.addSubsystem(Subsys, {matrix, 'varstr', {'a','b';'c','d'}})
obj.addSubsystem(.., 'Name', Value)
```

#### - Inputs -

**Subsys** anonymous function, simple function handle or matlab.System class instance.

**Inputs** Input arguments cell array. Entries in the cell can either be numeric (scalar, vector, matrix), a variable string, cell array of variable strings (variables in cell array must be concatable).

**Outputs** Output arguments cell array. Entries in the cell can either be a variable string, a cell array of variable strings. In the latter case the size of the cell array must fit the output size. Additionally, a ' ' can be used to ignore an output.

#### - Name Value -

**Optimize** Flag that sets the optimization option for the derivative creation (analytic derivative mode only). (default = true)

### Method addDerivativeSubsystem (Subsystem Mode)

Adds a subsystem to the subsystem chain of the project which already calculates derivatives. This enables the use of lookup tables or similar function in the subsystem chain. A funtion handel to the subsystem, inputs and outputs have to be specified. In case Name Value pairs are not set (OutputSizes) FALCON.m uses a nan call to the function to determine the output sizes, jacobian (and hessian) sparsity structure. If the function call cannot handle nan inputs, output sizes and sparsity patterns have to be provided.

*Keywords*: Base Builder Derivative Subsystem

#### *- Syntax -*

```
obj.addDerivativeSubsystem(Subsystem, Inputs, Outputs)
obj.addDerivativeSubsystem(Subsystem, 'Inputs', Inputs, 'Outputs',
    Outputs)
obj.addDerivativeSubsystem(.., 'Name', Value)
```

#### *- Inputs -*

**`Subsystem`** Must be a simple function handle (anonymous functions or matlab.System classes are not supported)

**`Inputs`** Input arguments cell array. Entries in the cell can either be numeric, a variable string, cell array of variable strings. Constant inputs (numeric, constant values) must not have their derivatives returned by the derivative subsystem.

**`Outputs`** Output arguments cell array. Entries in the cell can either be a variable string. Cell array of variable strings are not supported. Ignoring an output using ' ' is not supported.

#### *- Name Value -*

The following name value pairs are optional, but in case on is set the all relevant information has to be given.

**`OutputSizes`** The size of each output value.

**`OutputJacobianSparsity`** The sparsity pattern of the output jacobian given as a matrix of zeros and ones.

**`OutputHessianSparsity`** The sparsity pattern of the output hessian given as a matrix of zeros and ones.

### Method setConstraintValueNames (Subsystem Mode)

In case of subsystem mode, the names of the constraint values need to be provided to determine the output values of the constraint.

*Keywords*: Path Constraint Constraint Names

*- Syntax -*

```
obj.setConstraintValueNames({cell array of strings})
obj.setConstraintValueNames('name1', 'name2', ...)
```

*- Inputs -*

**Names** Cell array of strings or the names as individual inputs. A single name can be passed as a string.

### Method SplitVariable (Subsystem Mode)

Split large variables into smaller heaps. Since the method uses mat2cell interally the block structure and summation of rows and columns must fit. If the size of entries is the same as the size of the variable name, rowsplit and colsplit do not have to be provided. Otherwise the sum of rowsplit and sum of colsplit must fit the size of the variable name respectively. Not available in SimpleMode.

*Keywords*: Base Builder Variables Split

*- Syntax -*

```
obj.SplitVariable(name, entries)
obj.SplitVariable(name, entries, rowsplit, colsplit)
```

*- Inputs -*

**name** name of original variable

**entries** name of new entries, which is orientation sensitive.

**rowsplit** row distribution

**colsplit** column distribution

### Method CombineVariables (Subsystem Mode)

Combine multiple variables to a single variable to simplify the construction code. Not available in SimpleMode.

*Keywords*: Base Builder Variables Combine

*- Syntax -*

```
obj.CombineVariables(name, vars)
```

*- Inputs -*

**name** Name of the new variable

**vars** Cell array of strings. vars is orientation sensitive, meaning 'a', 'b', 'c' and 'a'; 'b'; 'c' will create different variables. Variables must have a matching block structure (see mat2cell).

### Method Build

Builds the current project, which means the derivative function interface is constructed. Afterwards the evaluation function is created. Additional settings for the evaluation function can be set.

*Keywords*: Base Builder Build

*- Syntax -*

```
handle = obj.Build()
handle = obj.Build('Name', Value)
```

*- Name Value -*

**EvaluationProvider** Parameter Value Pair. 'mex' generates a c++ mex file wrapper and mex function. 'none' will prevent the construction of the evaluation wrapper. Any other input invokes a MATLAB wrapper. (default = 'mex')

**MultiThreading** Flag to compile the model with multi-threading (default: false)

**OutputFolder** Folder to which the compiled/generated model is going to be saved (default: pwd).

*- Outputs -*

**handle** see get.Handle. In case 'none' was chosen for the evaluation provider, handle is empty []

## 6.5 Point Constraint Builder

Parent Classes: falcon.core.builder.BaseBuilder

### Properties

**+ DERIVATIVE_ANALYTIC** (Constant, read-only, Default = analytic)
Flag for setting builder to analytic derivative mode.

**+ DERIVATIVE_FINITE_DIFFERENCE** (Constant, read-only, Default = finite_difference)
Flat for setting builder to finite difference derivative mode.

**+ EVALUATION_MEX** (Constant, read-only, Default = mex)
Flag for settin builder to mex evaluation mode.

**+ EVALUATION_MATLAB** (Constant, read-only, Default = matlab)
Flag for setting builder to matlab evaluation mode.

**+ EVALUATION_NONE** (Constant, read-only, Default = none)
Flag for setting builder to no evaluation mode. (No wrapper is created)

**+ TYPE_OUTPUT** (Constant, read-only, Default = OUTPUT)
falcon.core.builder.BaseBuilder.TYPE_OUTPUT is a property.

**+ TYPE_STATE** (Constant, read-only, Default = STATE)
falcon.core.builder.BaseBuilder.TYPE_STATE is a property.

**+ TYPE_CONTROL** (Constant, read-only, Default = CONTROL)
falcon.core.builder.BaseBuilder.TYPE_CONTROL is a property.

**+ TYPE_PARAMETER** (Constant, read-only, Default = PARAMETER)
falcon.core.builder.BaseBuilder.TYPE_PARAMETER is a property.

**+ TYPE_VALUE** (Constant, read-only, Default = VALUE)
falcon.core.builder.BaseBuilder.TYPE_VALUE is a property.

**+ TYPE_DISCRETE** (Constant, read-only, Default = DISCRETE)
falcon.core.builder.BaseBuilder.TYPE_DISCRETE is a property.

**+ TYPE_CONSTANT** (Constant, read-only, Default = CONSTANT)
falcon.core.builder.BaseBuilder.TYPE_CONSTANT is a property.

**+ ProjectName** (read-only)
Name of the model or function project

**+ SimpleFunctionHandle** (read-only)
Holds the handle if a single function is used to define the function or Model / Constraint or Cost

**+ OptimizeCode** (read-only)
Perform Code Optimization

**+ isBuilt** (read-only)
Flag that determined if the project was already build

**+ isSimulinkModel** (read-only)
Flag that determined if the model is a Simulink model

**+ Handle** (Dependent)
Handle to the constructed model / constraint function.

### Methods

**PointConstraintBuilder** (Constructor)
Class to construct point constraints with derivatives in FALCON.m.

**> addConstant**
Add a internal constant to the project

**> addConstantInput**
Add a constant input to the dynamic model.

**> addDerivativeSubsystem**
Add Subsystem which already provides derivatives to the project

**> addPhaseInput**
Add a new phase input to the point constraint

**> addSubsystem**
Add Subsystem to the project to create its derivatives.

**> Build**
Builds the current project

**> CheckDerivatives**
Check Derivatives of the generated project

**> CombineVariables**
Combine multiple variables to a single variable

**> setConstraintValueNames**
Set the constraint value names for subsystem mode.

**> setParameters**
Sets the parameter objects entering the point constraint.

**> SimpleModeOutputVariableProcessing** (Static)
falcon.core.builder.BaseBuilder.SimpleModeOutputVariableProcessing is a function.
arr = SimpleModeOutputVariableProcessing(arr)

**> SplitVariable**
Split a single variable into multiple parts

### Constructor

This class prepares the a point constraint for the use in FALCON.m. It calculates the derivatives fully automatically. Two versions are supplied, the Function Mode and the Subsystem Mode.

*Keywords*: Constructor Point Constraint

*- Syntax -*

```
obj = falcon.PointConstraintBuilder(ProjectName)
obj = falcon.PointConstraintBuilder(ProjectName, Handle)
obj = falcon.PointConstraintBuilder(ProjectName, Handle, 'Name', Value)
```

*- Inputs -*

**ProjectName** The name of the generated constraint. This is the filename of the created constraint.

**Handle** Function Handle for constraints that are described using a single matlab function (Function Mode). Leave empty if you want to construct a model using subsystems (Subsystem Mode). (default: [])

*- Name Value -*

**DerivativeMode** Flag that defines if the derivatives are calculated using symbolic differentiation ('analytic') or using finite differences('finite_difference'). (default = 'analytic')

**Optimize** Set the Optimization option for symbolic differentiation. Only available in MATLAB 2014b or later. (Function Mode default=false, Subsystem Model default = true)

**DoDependencyCheck** Flag that enables a check if a subsystem is dependent on other subsystems. (default = false)

*- Outputs -*

**obj** The falcon.PointConstraintBuilder instance.

### Method addPhaseInput

This method adds a new set of inputs to the point constraints that belong to a phase. All inputs are optional but at least one of the outputs / states / controls need to be set. For each phase input block it can be specified how many input time steps will be expected (default = 1).

*Keywords*: Point Constraint Phase Input

*- Syntax -*

```
obj.addPhaseInput(Outputs, States, Controls, NumberOfTimeSteps)
obj.addPhaseInput(NumOutputs, NumStates, NumControls, NumberOfTimeSteps)
obj.addPhaseInput(States)
```

**Number of objects** If the number of the objects and not the actual objects are specified, then two conditions apply. The number of objects must be the same as the number of objects in the phase. Additionally, the order of the outputs, states, controls, timesteps arguments must be kept. This is true if at leas one output, state or control is specified using the number.

**Order of objects** In Function mode the order of the phase inputs specified is used to call the function handle. Thus it is possible to define a phase inputs in e.g. the following way: obj.addPhaseInput(states(1), outputs(2),states(2:4), 3) where there will be three inputs and the states are distributed into two seperate inputs.

**Basic Idea** In cases where the last syntax is used the point constraint is always added with exactly a single time step. It is important to note that at this stage it is not necessary (nor is it possible) to define the exact phase and normalized time that the phase input comes from. This is only possible (and mandatory) when adding the constraint to the problem. Thus, the constraint can be used modular within the problem.

*- Inputs -*

**Outputs** Array of falcon.Output objects or number of output expected to enter the point constraint. Specifying the input using a number is only available in Function Mode. Additionally, the number of outputs must match the number of model outputs in the phase. (default = no outputs)

**States** Array of falcon.State objects or number of states expected to enter the point constraint. Specifying the input using a number if only available in Function Model. Additionally, the number of states must match the number of states in the phase. (default = no states)

**Controls** Array of falcon.Control objects or number of controls expected to enter the point constraint. Specifying the input using a number is only available in Function Mode. Additionally, the number of controls must match the number of model outputs in the phase. (default = no controls)

**NumberOfTimeSteps** The number of time steps the phase input has. Here only the time steps and thus the size is specified. Which times are given to the point constraint is specified in falcon.Problem.addNewPointConstraint.

### Method setParameters

Set the parameter required by the constraint to calculate its values. Only call this method if the constraint requires parameters.

*Keywords*: Point Constraint Parameter Names

*- Syntax -*

```
obj.setParameters(Parameters)
```

*- Inputs -*

**Parameters** Array of falcon.Parameter objects or number of parameters (Function Mode only).

### Method addConstantInput

Additional constant inputs to the model / constraint can be set using this function. They will be added in order of occurence after the main input sequence f(x,u,p,c1,c2,...). These constant inputs can be set and changed externally and are not hard-coded. This makes testing different model types efficient. They must be added to the model in the problem using the setConstant-method and thus, only the size is specified here.

*Keywords*: Base Builder Constant Input

*- Syntax -*

```
obj.addConstantInput(Name)
obj.addConstantInput(Name, 'Name', Value)
```

**Name** Name of input. (string)

**VariableSize** Either a numeric dimension [m,n] (must be 1 by 2 row vector) or a cell array of string specifying multiple [1,1] entries to the model. This is the size used for each non-dimensional time step.

### Method addConstant (Subsystem Mode)

Set constant values in Subsystem Mode. Values are internal and cannot be influence from the outside. For additional inputs use the addConstantInput method. This method throws an error if called in Function Mode.

*Keywords*: Base Builder Constant

#### - Syntax -

```
obj.addConstant(Name, Value)
```

#### - Inputs -

**Name** Name of the constant. (string)

**Value** Value of the constant. (numeric, scalar, vector or matrix).

### Method addSubsystem (Subsystem Mode)

Add a subsystem to the model / constraint. The subsystems are chained in order of appearance. Therefore, it is crucial that all data required by the subsystem is already present (states, controls, outputs of other subsystems).

*Keywords*: Base Builder Subsystem

#### - Syntax -

```
obj.addSubsystem(Subsys, Inputs, Outputs)
obj.addSubsystem(Subsys, 'Inputs', Inputs, 'Outputs', Outputs)
obj.addSubsystem(Subsys, {matrix, 'varstr', {'a','b';'c','d'})
obj.addSubsystem(.., 'Name', Value)
```

#### - Inputs -

**Subsys** anonymous function, simple function handle or matlab.System class instance.

**Inputs** Input arguments cell array. Entries in the cell can either be numeric (scalar, vector, matrix), a variable string, cell array of variable strings (variables in cell array must be concatable).

**Outputs** Output arguments cell array. Entries in the cell can either be a variable string, a cell array of variable strings. In the latter case the size of the cell array must fit the output size. Additionally, a ' ' can be used to ignore an output.

#### - Name Value -

**Optimize** Flag that sets the optimization option for the derivative creation (analytic derivative mode only). (default = true)

### Method addDerivativeSubsystem (Subsystem Mode)

Adds a subsystem to the subsystem chain of the project which already calculates derivatives. This enables the use of lookup tables or similar function in the subsystem chain. A funtion handel to the subsystem, inputs and outputs have to be specified. In case Name Value pairs are not set (OutputSizes) FALCON.m uses a nan call to the function to determine the output sizes, jacobian (and hessian) sparsity structure. If the function call cannot handle nan inputs, output sizes and sparsity patterns have to be provided.

*Keywords*: Base Builder Derivative Subsystem

#### *- Syntax -*

```
obj.addDerivativeSubsystem(Subsystem, Inputs, Outputs)
obj.addDerivativeSubsystem(Subsystem, 'Inputs', Inputs, 'Outputs',
    Outputs)
obj.addDerivativeSubsystem(.., 'Name', Value)
```

#### *- Inputs -*

**`Subsystem`** Must be a simple function handle (anonymous functions or matlab.System classes are not supported)

**`Inputs`** Input arguments cell array. Entries in the cell can either be numeric, a variable string, cell array of variable strings. Constant inputs (numeric, constant values) must not have their derivatives returned by the derivative subsystem.

**`Outputs`** Output arguments cell array. Entries in the cell can either be a variable string. Cell array of variable strings are not supported. Ignoring an output using ' ' is not supported.

#### *- Name Value -*

The following name value pairs are optional, but in case on is set the all relevant information has to be given.

**`OutputSizes`** The size of each output value.

**`OutputJacobianSparsity`** The sparsity pattern of the output jacobian given as a matrix of zeros and ones.

**`OutputHessianSparsity`** The sparsity pattern of the output hessian given as a matrix of zeros and ones.

### Method setConstraintValueNames (Subsystem Mode)

In case of subsystem mode, the names of the constraint values need to be provided to determine the output values of the constraint.

*Keywords*: Point Constraint Constraint Names

*- Syntax -*

```
obj.setConstraintValueNames({cell array of strings})
obj.setConstraintValueNames('name1', 'name2', ...)
```

*- Inputs -*

**Names** Cell array of strings or the names as individual inputs. A single name can be passed as a string.

### Method SplitVariable (Subsystem Mode)

Split large variables into smaller heaps. Since the method uses mat2cell interally the block structure and summation of rows and columns must fit. If the size of entries is the same as the size of the variable name, rowsplit and colsplit do not have to be provided. Otherwise the sum of rowsplit and sum of colsplit must fit the size of the variable name respectively. Not available in SimpleMode.

*Keywords*: Base Builder Variables Split

*- Syntax -*

```
obj.SplitVariable(name, entries)
obj.SplitVariable(name, entries, rowsplit, colsplit)
```

*- Inputs -*

**name** name of original variable

**entries** name of new entries, which is orientation sensitive.

**rowsplit** row distribution

**colsplit** column distribution

### Method CombineVariables (Subsystem Mode)

Combine multiple variables to a single variable to simplify the construction code. Not available in SimpleMode.

*Keywords*: Base Builder Variables Combine

*- Syntax -*

```
obj.CombineVariables(name, vars)
```

*- Inputs -*

**name** Name of the new variable

**vars** Cell array of strings. vars is orientation sensitive, meaning 'a', 'b', 'c' and 'a'; 'b'; 'c' will create different variables. Variables must have a matching block structure (see mat2cell).

### Method Build

Builds the current project, which means the derivative function interface is constructed. Afterwards the evaluation function is created. Additional settings for the evaluation function can be set.

*Keywords*: Base Builder Build

#### *- Syntax -*

```
handle = obj.Build()
handle = obj.Build('Name', Value)
```

#### *- Name Value -*

**`EvaluationProvider`** Parameter Value Pair. 'mex' generates a c++ mex file wrapper and mex function. 'none' will prevent the construction of the evaluation wrapper. Any other input invokes a MATLAB wrapper. (default = 'mex')

**`MultiThreading`** Flag to compile the model with multi-threading (default: false)

**`OutputFolder`** Folder to which the compiled/generated model is going to be saved (default: pwd).

#### *- Outputs -*

**`handle`** see get.Handle. In case 'none' was chosen for the evaluation provider, handle is empty []

# Index

# References

[1] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, Philadelphia, 2009.

[2] Christof Büskens. *Optimierungsmethoden und Sensitivitätsanalyse für optimale Steuerprozesse mit Steuer- und Zustands-Beschränkungen*. Dissertation, Westfälische Wilhelms-Universität, Münster, 1998.

[3] Matthias Gerdts. *Optimal control of ODEs and DAEs*. De Gruyter textbook. De Gruyter, Berlin and Boston, 2012.