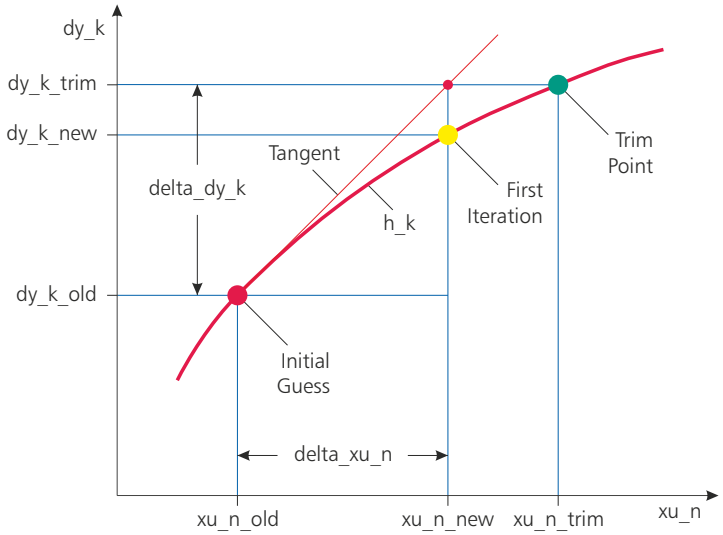

TrimMod 2.2

Jörg J. Buchholz

February 16, 2022

1 Manual

TrimMod[1] (figure 1.1) finds the trim point (equilibrium[2]) of a Simulink system.

The user can load a Simulink system, define certain trim requirements and ask TrimMod to calculate the corresponding trim variables that are necessary to satisfy the requirements. Finally, the Simulink system is initialized with the new trim point.

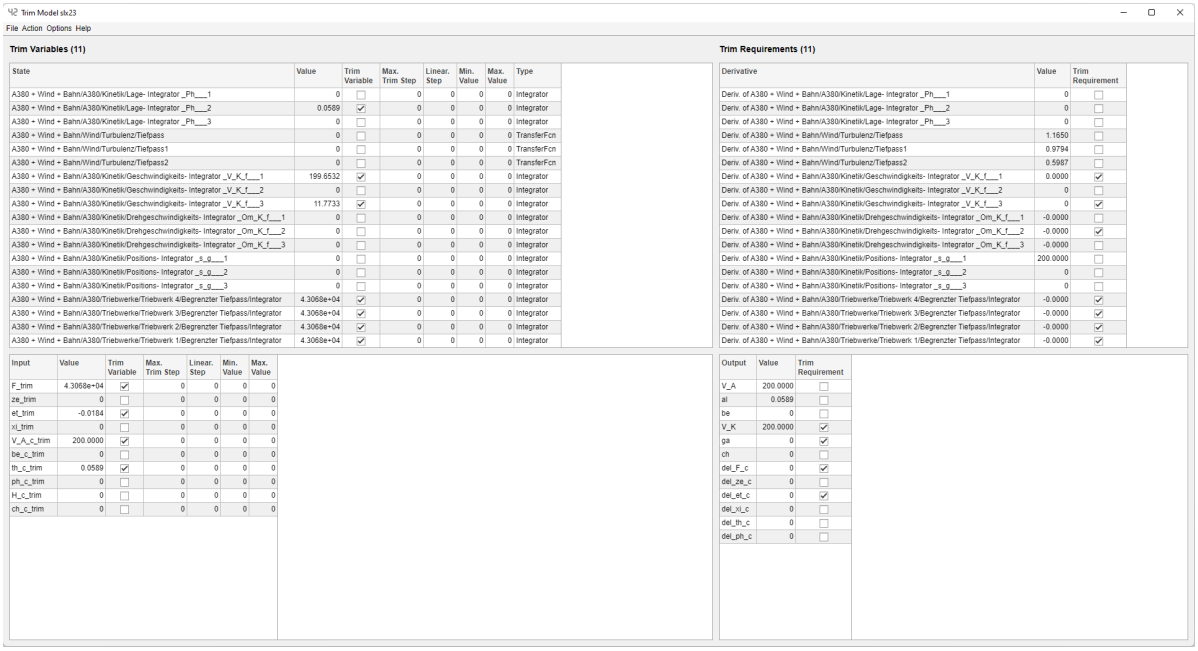

Figure 1.1: TrimMod 2.2

1.1 Block diagram and typical use case

As depicted in figure 1.2, the user opens the main app `trimmod.mlapp` (figure 1.1) and loads the Simulink model to be trimmed (e. g. `model.slx`) via the *File | Open Model* menu entry and – if already existing – a trim point file (e. g. `trimpoint.mat`) via *File | Load Trim Point*.

[1] TrimMod 2.2 has been developed and tested with Matlab 2021b.
[2] In fact, even non-equilibrium (instationary, accelerated, . . .) trim points can be found.

The user can then alter the trim requirements and the trim variables and define additional parameters (maximum number of trim iterations, cost value to be gained, ...) by calling `trimmod_parameters.mlapp` (chapter 3) via the *Options | Additional Parameters* menu entry.

If the trim point is formally well defined (i. e. the number of trim requirements equals the number of trim variables), the user can select the *Action | Trim* menu entry: `trimmod` calls `jj_trim.m` (chapter 5) that does the actual trimming and returns the new trim point. This trim point is displayed in TrimMod and automatically transferred to the *Modeling | Model Settings | Configuration Parameters | Data Import/Export | Load from workspace* menu entry of the Simulink system.

Finally, calling `trimmod_overview.mlapp` (chapter 4) via the *Options | Show Overview* menu entry displays the values of states, inputs, ... for every single iteration step of the trim process (figure 4.1).

The user can save the current trim point at any time by selecting the *File | Save Trim Point As* menu entry.

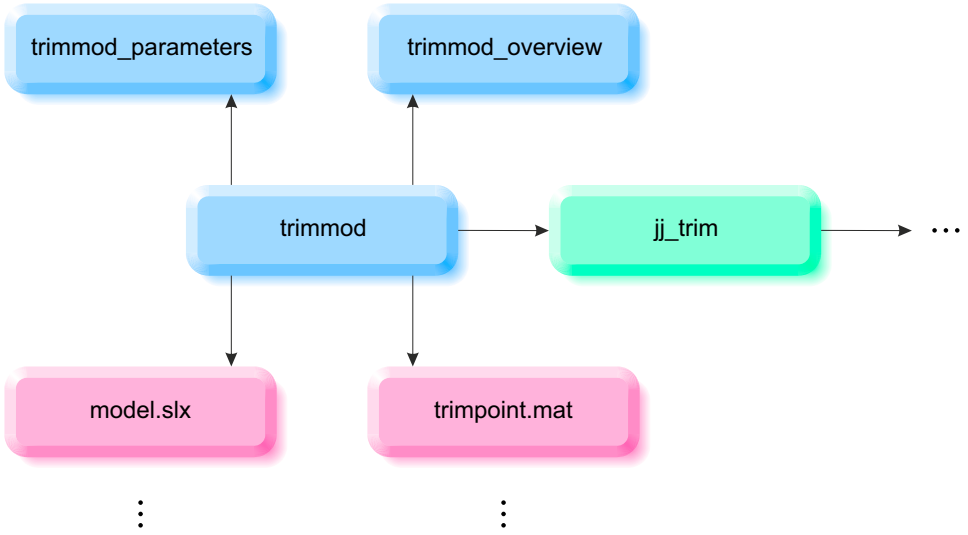

Figure 1.2: TrimMod block diagram

2 trimmod app

Unlike TrimMod 1.x, which was a classic GUIDE `fig`-file with a corresponding `m`-file (figure 2.1)

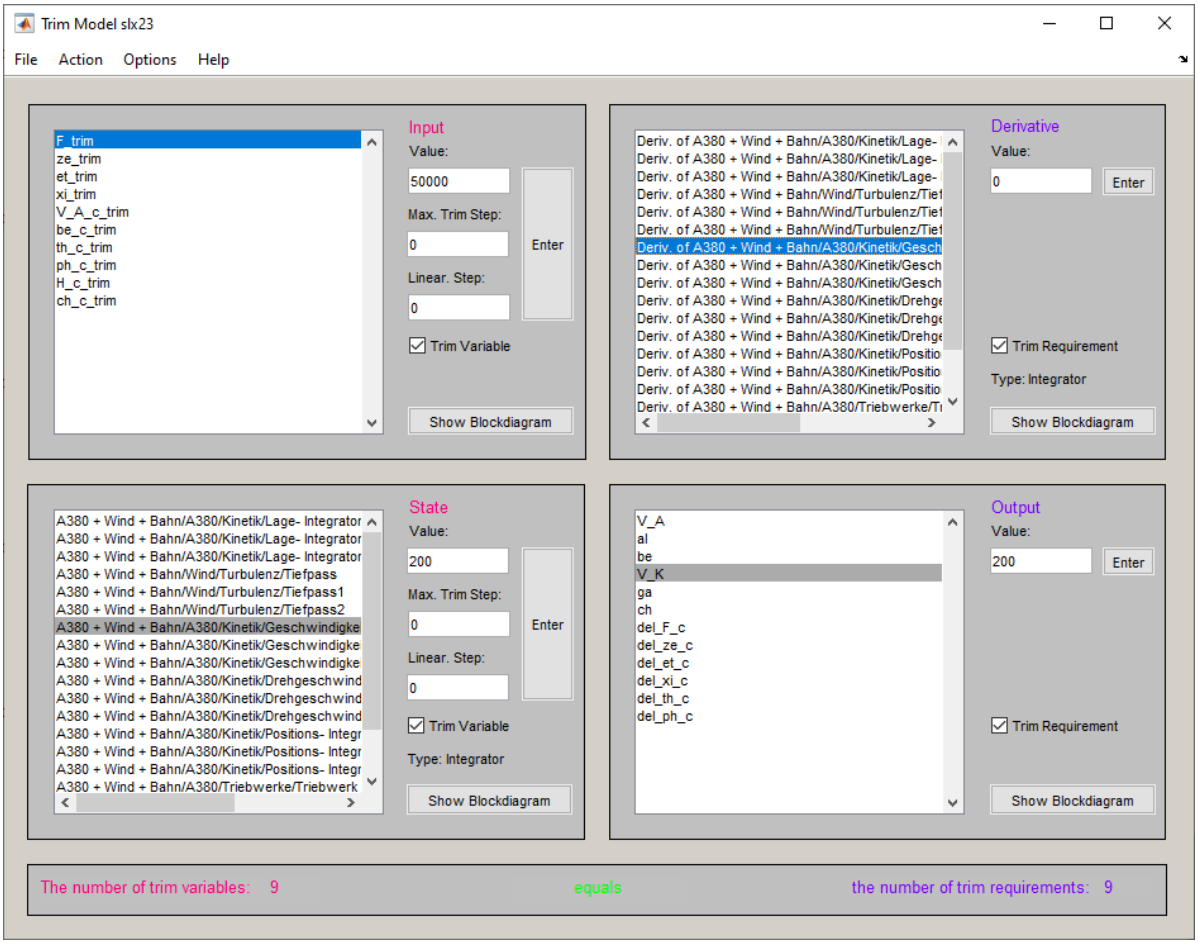

Figure 2.1: TrimMod 1.x

we designed TrimMod 2.2 as a contemporary Matlab App with the App Designer (figure 1.1).

Matlab Apps are classes that inherit methods (`delete`, `findobj`, `listener`, ...) from Matlab's App base class:

```
classdef trimmod < matlab.apps.AppBase
```

The App Designer automatically generates public properties for all the user interfaces (menus, tables, ...) we created in the Design View:

```
properties (Access = public)
    TrimModelUIFigure          matlab.ui.Figure
    FileMenu                   matlab.ui.container.Menu
    OpenModelMenu              matlab.ui.container.Menu
    LoadTrimPointMenu          matlab.ui.container.Menu
    SaveTrimPointMenu          matlab.ui.container.Menu
    SaveTrimPointasMenu        matlab.ui.container.Menu
    ExitTrimModMenu            matlab.ui.container.Menu
    ActionMenu                 matlab.ui.container.Menu
    TrimMenu                   matlab.ui.container.Menu
    UntrimMenu                 matlab.ui.container.Menu
    OptionsMenu                matlab.ui.container.Menu
    AdditionalParametersMenu   matlab.ui.container.Menu
    ShowOverviewMenu           matlab.ui.container.Menu
    HelpMenu                   matlab.ui.container.Menu
    HelponTrimModMenu          matlab.ui.container.Menu
    AboutTrimModMenu           matlab.ui.container.Menu
    GridLayout                 matlab.ui.container.GridLayout
    UITable_output             matlab.ui.control.Table
    UITable_derivative         matlab.ui.control.Table
    UITable_state              matlab.ui.control.Table
    TrimRequirementsLabel      matlab.ui.control.Label
    TrimVariablesLabel         matlab.ui.control.Label
    UITable_input              matlab.ui.control.Table
end
```

We declare a few private variables for the use in more than one method

```
properties (Access = private)
    parameters_app
    overview_app
    model_name
    model_path
    initial_conditions
    state_names_with_nl
    state_types
    state_names
    derivative_names
    inport_names
    n_inports
    outport_names
    n_outports
    pre_trim
    output_names
    input_names
```

```
end
```

and some public variables that can also be accessed by other apps (chapter 3 and chapter 4) too:

```
properties (Access = public)
  n_states
  n_outputs
  n_inputs
  parameters
  info_struct
  jaco_figure
  x
  u
  d
  y
  x_nam
  u_nam
  d_nam
  y_nam
  i_x
  i_u
  i_d
  i_y
end
```

Nearly all trimmod methods are callback functions and therefore automatically generated by the App Designer. The only method

```
methods (Access = private)
```

defined by ourselves is the function that saves the table data in a specific file (section 2.1).

2.1 save_trim_point

The private method save_trim_point saves the current table data – defining the current trim point – and the additional trim parameters in the file named via its parameter list:

```
function save_trim_point (app, file_name)
```

We create a save structure and add the current model name to the structure:

```
save_struct = struct ('model_name', app.model_name);
```

Additionally, we want to save the contents of all four tables (input, state, derivative, output)

```
save_struct.input_data = app.UITable_input.Data;
save_struct.state_data = app.UITable_state.Data;
save_struct.derivative_data = app.UITable_derivative.Data;
save_struct.output_data = app.UITable_output.Data;
```

and the additional trim parameters:

```
save_struct.parameters = app.parameters;
```

Finally, we save the structure in the file defined via the parameter list:

```
save (file_name, 'save_struct');
end
end
```

The following methods

```
methods (Access = private)
```

are callback functions auto-generated by the App Designer.

2.2 startupFcn

The startupFcn executes when the app starts up, but before the first user interaction:

```
function startupFcn(app)
```

We use it to initialize additional trim parameters with their default values and save them as public properties:

```
app.parameters.n_iter_max = 42;
app.parameters.cost_tbg = 1e-9;
app.parameters.CompileFlag = 1;
app.parameters.n_bisec_max = 10;
app.parameters.EnableMessages = true;
```

We will later directly access these parameters from section 3.1 and section 3.2.

Additionally, we add the current directory (the path to TrimMod) to Matlab's search path[1]:

```
addpath (cd);
```

[1] TrimMod calls other apps and functions in its own directory. Therefore, this directory has to be on Matlab's search path if we change the current directory to the model path later on (section 2.7).

App position and size

Furthermore, we might want to adjust the position of the app with respect to the current screen resolution. As a default, we chose a Full HD (1080p) app resolution (1920×1080) that most modern monitors can display. If this is not the case, we adjust the app resolution to the current monitor resolution.

We obtain a handle to the primary monitor

```
pri_mon = groot;
```

and the default[2] app position of `[1, 50, 1920, 980]`, including a safety margin[3] of 50 pixel at the bottom and at the top of the app:

```
win_pos = app.TrimModelUIFigure.Position;
```

If the app width is greater than the screen width

```
if win_pos(3) > pri_mon.ScreenSize(3)
```

we set the app width to the screen width:

```
    win_pos(3) = pri_mon.ScreenSize(3);
    app.TrimModelUIFigure.Position = win_pos;
end
```

If the app height is greater than the screen height (including the safety margins)

```
if win_pos(4) > pri_mon.ScreenSize(4) - 100
```

we set the app height to the (reduced) screen height:

```
    win_pos(4) = pri_mon.ScreenSize(4) - 100;
    app.TrimModelUIFigure.Position = win_pos;
end
```

2.3 ExitTrimModMenuSelected

The `ExitTrimModMenuSelected` method is called if the user selects the *File | Exit TrimMod* menu or if they press the red window close button:

```
function ExitTrimModMenuSelected(app, event)
```

[2] We defined the app's default `UIFigure` position of `[1, 50, 1920, 980]` in the Design View of the App Designer.

[3] Finding "safe" positions in a graphical Windows-based application is really a PITA. In Windows 10, the user could adjust the size of the taskbar, make it automatically disappear, or glue it to the top or even to the sides of the screen. Therefore, a safety margin of 50 pixel – which is the default height of the taskbar – at the bottom and at the top of the app should work in many standard Windows environments.

In this case, we want to give the user a last chance to save the current trim point. Therefore, we open a confirmation dialog box[4]

```
button_choice = uiconfirm ( ...
  app.TrimModelUIFigure, ...
  'Save trim point before exiting?', ...
  'Save trim point', ...
  'Options',{'Yes', 'No', 'Cancel'}, ...
  'DefaultOption', 1, ...
  'CancelOption', 3);
```

and take action depending on their choice:

```
switch button_choice
```

If they have pressed the `Cancel` button

```
  case 'Cancel'
```

we do not want to do anything and directly terminate this method:

```
    return
```

If they confirm that they want to save the current trim point,

```
  case 'Yes'
```

we call the appropriate method (section 2.10):

```
    SaveTrimPointAsMenuSelected (app, event)
end
```

If the user has **not** pressed the `Cancel` button, we close additional windows (apps and figures) that TrimMod might have opened

```
delete (app.parameters_app)
delete (app.overview_app)
delete (app.jaco_figure)
```

and finally close TrimMod itself:

```
delete (app)
end
```

2.4 AboutTrimModMenuSelected

The *Help | About TrimMod* menu method `AboutTrimModMenuSelected`

[4] Yes, the dialog box is also opened if the user has not changed anything. Maybe later ...

```
function AboutTrimModMenuSelected(app, event)
```

just displays a classic About dialog

```
helpdlg ({ ...
    'TrimMod 2.2'; ...
    'Joerg J. Buchholz'; ...
    'Hochschule Bremen'; ...
    'buchholz@hs-bremen.de'}, ...
    'About TrimMod');
end
```

with contact details of the author (figure 2.2).

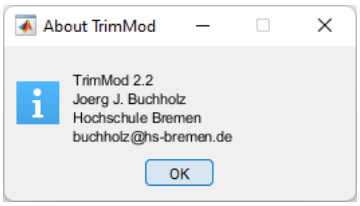

Figure 2.2: About TrimMod

2.5 HelponTrimModMenuSelected

As an additional help option, the *Help | Help on TrimMod* menu entry

```
function HelponTrimModMenuSelected(app, event)
```

opens this documentation:

```
open ('trimmod.pdf')
```

2.6 AdditionalParametersMenuSelected

As described in chapter 3, the *Options | Additional Parameters* menu entry

```
function AdditionalParametersMenuSelected(app, event)
```

opens another app by the name of `trimmod_parameters`:

```
app.parameters_app = trimmod_parameters(app);
end
```

2.7 OpenModelMenuSelected

The *File | Open Model* menu entry

```
function OpenModelMenuSelected(app, event)
```

opens an existing model and extracts all state, input, and output names of this model.

First, we ask the user via a file selection dialog box which model they want to open:

```
[file_name, file_path] = uigetfile ('*.slx;*.mdl', 'Open Model');
```

Unfortunately, we now have to bring back the app window to front after a file selection dialog box. This is a bug workaround that might be unnecessary in the future:

```
figure (app.TrimModelUIFigure);
```

If the user has pressed the Cancel button of the file selection dialog box

```
if ~file_name
```

we do not want to do anything and directly terminate this method:

```
  return
end
```

If the user has selected a valid model, we extract the model name without the file extension and save it in a property of this app:

```
app.model_name = strtok (file_name, '.');
```

We extract the model path and save it without the trailing backslash:

```
app.model_path = file_path(1 : (end - 1));
```

In order to inform the user about the model they have chosen, we insert the model name in the name of the app figure:

```
app.TrimModelUIFigure.Name = ['Trim Model ', app.model_name];
```

We actively set Matlab's working directory to the path of the current model[5]:

```
cd (app.model_path);
```

We delete all state, inport, derivative, and outport names. This is necessary if the previously opened model had more states than the current one[6]:

[5] In some cases this might not be the best idea especially since we never actively set the path back to where we came from. Nevertheless, in some cases the current model depends on other files in the same directory and since we already added the path to TrimMod to Matlab's search path (section 2.2), it seems safe to make the model directory the current one.

[6] Actually, this is only necessary for the derivatives, but well ...

```
app.state_names = {};
app.inport_names = {};
app.derivative_names = {};
app.outport_names = {};
```

For the next two commands, we first have to open the main system block diagram internally without bringing it to front:

```
load_system (app.model_name);
```

We uncheck (disable) the *Load from Workspace / Initial state* and *Input* checkboxes in the *Configuration Parameters* page of the current model:

```
set_param (app.model_name, 'LoadInitialState', 'off');
set_param (app.model_name, 'LoadExternalInput', 'off');
```

This prevents an error if the user has changed the number of states or inputs in the model since the last trim.

By calling the model, we receive the number of states, inputs, and outputs, the initial states, and the state names[7]:

```
[sizes, app.initial_conditions, app.state_names_with_nl] = ...
eval (app.model_name);
```

We extract the number of continuous states, inputs, and outputs

```
app.n_states = sizes(1);
app.n_outputs = sizes(3);
app.n_inputs = sizes(4);
```

find the block types of all states[8]

```
app.state_types = ...
get_param (app.state_names_with_nl, 'blocktype');
```

and substitute potential newline characters in the state names by blanks for better readability in the tables with the help of the external function `kill_nl`:

```
app.state_names = kill_nl (app.state_names_with_nl);
```

States

In a loop over all the states

```
for i_state = 1 : app.n_states
```

we strip the model name from the state names in order to make them easier to read in the tables

[7] Since Matlab Release 11 (Version 5.3), the state names might include newline characters.
[8] Valid state block types are `Integrator`, `StateSpace`, `TransferFcn`, `ZeroPole`, and `M-S-Function`.

```
  app.state_names{i_state} = ...
  app.state_names{i_state}(length (app.model_name) + 2 : end);
end
```

In another loop over all the states, we want to uniquefy multiple state names:

```
for i_state = 1 : app.n_states
```

First, we have to find the names of multiple states

```
  matches = find ...
    (strcmp (app.state_names(i_state), app.state_names));
```

and their numbers

```
  n_matches = length (matches);
```

If multiple state names have been found

```
  if n_matches > 1
```

we start a loop over all copies of a multiple state name

```
    for i_match = 1 : n_matches
```

and append ___1, ___2, ... to every copy of a multiple state name

```
      app.state_names{matches(i_match)} = ...
        [app.state_names{matches(i_match)}, '___', ...
        num2str(i_match)];
    end
  end
```

Derivatives get the expression Deriv. of in front of their names to make sure the user understands that these are the derivatives of the states:

```
  app.derivative_names{i_state} = ...
    ['Deriv. of ', app.state_names{i_state}];
end
```

Inports

In order to detect vector inports (that have more than one input), we have to compile[9] the model:

```
feval (app.model_name, [], [], [], 'compile');
```

We now want to distinguish between inports and inputs. We find and save all inports[10]

[9] This command "initiates" the simulation allowing us to access parameters that are only accessible in this state. On the other hand, later on, before we can actually start the simulation, we have to reset the model to the "uncompiled" state by feval (app.model_name, [], [], [], 'term');.

[10] The FollowLinks parameter finds inports in libraries, too.

```
app.inport_names = find_system (app.model_name, ...
  'FollowLinks', 'on', ...
  'searchdepth', 1, ...
  'Blocktype', 'Inport');
```

and the number of inports

```
app.n_inports = length (app.inport_names);
```

initialize the input names cell array

```
app.input_names = cell (app.n_inputs, 1);
```

and an input counter

```
i_input = 1;
```

and start a loop over all inports

```
for i_inport = 1 : app.n_inports
```

We determine the number of inputs of the current inport[11]

```
  n_inputs_of_port = get_param (app.inport_names{i_inport}, ...
    'CompiledPortWidths').Outport;
```

and strip the model name from the port names in order to make them more readable in the table

```
  app.inport_names{i_inport} = ...
    app.inport_names{i_inport} ...
    (length (app.model_name) + 2 : end);
```

If the current inport is a vector inport, it represents more than one input

```
    if n_inputs_of_port > 1
```

and we start a loop over all inputs of the current inport:

```
    for i_inputs_of_port = 1 : n_inputs_of_port
```

As with the states, we append ___1, ___2, ... to every copy of a multiple input name

```
      app.input_names{i_input} = ...
        [app.inport_names{i_inport}, '___', ...
        num2str(i_inputs_of_port)];
```

and increment the input index

```
      i_input = i_input + 1;
    end
```

[11] It sounds strange, but seems logical if you think about it: The inputs represented by a certain inport are the "outports" of this inport.

If the current inport is scalar

```
  else
```

we simply copy the inport name to the input name

```
    app.input_names{i_input} = app.inport_names{i_inport};
```

and increment the input index

```
    i_input = i_input + 1;
  end
end
```

Outports

Since the processing of the outports and outputs is conceptually identical to the steps described in the previous section, we do not comment every single command:

```
app.outport_names = find_system (app.model_name, ...
    'FollowLinks', 'on', ...
    'searchdepth', 1, ...
    'Blocktype', 'Outport');

app.n_outports = length (app.outport_names);

app.output_names = cell (app.n_outputs, 1);

i_output = 1;

for i_outport = 1 : app.n_outports
  n_outputs_of_port = get_param ...
    (app.outport_names{i_outport}, ...
    'CompiledPortWidths').Inport;

  app.outport_names{i_outport} = ...
    app.outport_names{i_outport} ...
    (length (app.model_name) + 2 : end);

  if n_outputs_of_port > 1
    for i_outputs_of_port = 1 : n_outputs_of_port
      app.output_names{i_output} = ...
        [app.outport_names{i_outport}, '___', ...
        num2str(i_outputs_of_port)];

      i_output = i_output + 1;
    end
  else
    app.output_names{i_output} = ...
```

```
      app.outport_names{i_outport};

    i_output = i_output + 1;
  end
end
```

As described in footnote (9), we have to revoke the simulation initialization (compilation) that we only needed to determine the number of inputs and outputs of the inports and outports:

```
feval (app.model_name, [], [], [], 'term');
```

Before we can write the model data into the corresponding tables, we initialize the tables

```
app.UITable_input.Data = num2cell (zeros (app.n_inputs, 7));
app.UITable_state.Data = num2cell (zeros (app.n_states, 7));
app.UITable_derivative.Data = num2cell (zeros (app.n_states, 3));
app.UITable_output.Data = num2cell (zeros (app.n_outputs, 3));
```

and specify the checkboxes:[12]

```
app.UITable_input.Data(:, 3) = ...
  num2cell (false (app.n_inputs, 1));
app.UITable_state.Data(:, 3) = ...
  num2cell (false (app.n_states, 1));
app.UITable_derivative.Data(:, 3) = ...
  num2cell (false (app.n_states, 1));
app.UITable_output.Data(:, 3) = ...
  num2cell (false (app.n_outputs, 1));
```

We are now ready to write the input, state, derivative, and output names into the corresponding tables:

```
app.UITable_input.Data(:, 1) = kill_nl (app.inport_names);
app.UITable_state.Data(:, 1) = app.state_names;
app.UITable_derivative.Data(:, 1) = app.derivative_names;
app.UITable_output.Data(:, 1) = kill_nl (app.outport_names);
```

Additionally, we write the state types into the last column (figure 2.3) of the state table:

```
app.UITable_state.Data(:, 8) = app.state_types;
```

[12] By assigning `false` to the third columns of all tables, we "declare" these columns as checkboxes.

State	Value	Trim Variable	Max. Trim Step	Linear. Step	Min. Value	Max. Value	Type
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _Ph___1	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _Ph___2	0	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _Ph___3	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass	0	☐	0	0	0	0	TransferFcn
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass1	0	☐	0	0	0	0	TransferFcn
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass2	0	☐	0	0	0	0	TransferFcn
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits- Integrator _V_K_f___1	200	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits- Integrator _V_K_f___2	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits- Integrator _V_K_f___3	0	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindigkeits- Integrator _Om_K_f___1	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindigkeits- Integrator _Om_K_f___2	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindigkeits- Integrator _Om_K_f___3	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrator _s_g___1	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrator _s_g___2	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrator _s_g___3	0	☐	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 4/Begrenzter Tiefpass/Integrator	0	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 3/Begrenzter Tiefpass/Integrator	0	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 2/Begrenzter Tiefpass/Integrator	0	☑	0	0	0	0	Integrator
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 1/Begrenzter Tiefpass/Integrator	0	☑	0	0	0	0	Integrator

Figure 2.3: State types in the last column of the state table

A call to the `update` method (section 2.8) displays the current number of trim variables and trim requirements in the app (bold headings in figure 1.1):

```
update (app, [])
```

Finally, we enable appropriate menu entries

```
app.LoadTrimPointMenu.Enable = true;
app.SaveTrimPointMenu.Enable = true;
app.SaveTrimPointAsMenu.Enable = true;
```

and append the model name to *Save Trim Point in ...* menu entry:

```
app.SaveTrimPointMenu.Text = ...
  ['&Save Trim Point in ', app.model_name, '.mat'];
end
```

2.8 update

The `update` method

```
function update(app, event)
```

determines the current number of trim variables and trim requirements, displays these numbers, and indicates a possible mismatch. It is called as a callback function whenever the user edits a cell in any of the tables.

We compute the number of trim variables as the sum of all checked inputs and states

```
n_trim_variables = ...
  sum (cell2mat (app.UITable_input.Data(:, 3))) + ...
  sum (cell2mat (app.UITable_state.Data(:, 3)));
```

and indicate this number in the corresponding label (bold heading in figure 1.1):

```
app.TrimVariablesLabel.Text = ...
  ['Trim Variables (', num2str(n_trim_variables), ')'];
```

The same is done for the number of trim requirements:

```
n_trim_requirements = ...
  sum (cell2mat (app.UITable_derivative.Data(:, 3))) + ...
  sum (cell2mat (app.UITable_output.Data(:, 3)));

app.TrimRequirementsLabel.Text = ...
  ['Trim Requirements (', num2str(n_trim_requirements), ')'];
```

If there is an equal number of trim variables and trim requirements

```
if n_trim_variables == n_trim_requirements
```

we color the corresponding labels black

```
  app.TrimVariablesLabel.FontColor = 'black';
  app.TrimRequirementsLabel.FontColor = 'black';
```

and enable the *Action* / *Trim* menu entry:

```
  app.TrimMenu.Enable = true;
```

If there is an unequal number of trim variables and trim requirements

```
else
```

we indicate the mismatch to the user by coloring the corresponding labels red

```
  app.TrimVariablesLabel.FontColor = 'red';
  app.TrimRequirementsLabel.FontColor = 'red';
```

and disable the *Action* / *Trim* menu entry:

```
  app.TrimMenu.Enable = false;
end
end
```

2.9 SaveTrimPointMenuSelected

If the user selects the *File* / *SaveTrimPoint* menu entry, the corresponding callback function is called:

```
function SaveTrimPointMenuSelected(app, event)
```

Its only task is to save the current trim point in the default file by calling the dedicated method (section 2.1) with the default file name:

```
save_trim_point (app, app.model_name)
end
```

2.10 SaveTrimPointAsMenuSelected

The callback function

```
function SaveTrimPointMenuSelected(app, event)
```

is called if the user selects the *File | Save Trim Point As* menu entry. It opens a file selection dialog box and lets the user decide in which file they want to save the trim point:

```
[file_name, file_path] = ...
  uiputfile ([app.model_name, '.mat'], 'Save Trim Point As');
```

Unfortunately, we now have to bring back the app window to front after a file selection dialog box. This is a bug workaround that might be unnecessary in the future:

```
figure (app.TrimModelUIFigure);
```

If the user has pressed the Cancel button of the file selection dialog box

```
if ~file_name
```

we do not want to do anything and directly terminate this method:

```
  return
end
```

If the user has selected a valid file name, we hand the user-defined file name over to the dedicated method (section 2.1):

```
save_trim_point (app, [file_path, file_name]);
end
```

2.11 LoadTrimPointMenuSelected

The *File | Load Trim Point* menu entry calls the corresponding function:

```
function LoadTrimPointMenuSelected(app, event)
```

In the function, we open a file selection dialog box and let the user decide which trim point to load:

```
[file_name, file_path] = uigetfile ('*.mat', 'Load Trim Point');
```

Unfortunately, we now have to bring back the app window to front after a file selection dialog box. This is a bug workaround that might be unnecessary in the future:

```
figure (app.TrimModelUIFigure);
```

If the user has pressed the Cancel button of the file selection dialog box

```
if ~file_name
```

we do not want to do anything and directly terminate this method:

```
  return
end
```

If the user has selected a valid file name, we load the `save_struct` defining the new trim point from the user-defined file:

```
load ([file_path, file_name], 'save_struct');
```

Parameters

We update the current parameters with the loaded data:

```
app.parameters = save_struct.parameters;
```

The next lines of code compare the current and the loaded trim point and define which entities have to be updated. In general, we only want to update those entities that are present in the current **and** in the loaded trim point. For that purpose, we use the `intersect` command that finds the intersection of two sets and returns the indices of the common elements in both sets.

Inputs

We find those rows with identical input names in the loaded data and the current table

```
[~, rows_loaded, rows_current] = intersect ...
  (save_struct.input_data(:, 1), app.UITable_input.Data(:, 1));
```

and update only those rows with identical names:

```
app.UITable_input.Data(rows_current, :) = ...
  save_struct.input_data(rows_loaded, :);
```

States

The same procedure updates only the common states, derivatives, and outputs:

```
[~, rows_loaded, rows_current] = intersect ...
  (save_struct.state_data(:, 1), app.UITable_state.Data(:, 1));

app.UITable_state.Data(rows_current, :) = ...
  save_struct.state_data(rows_loaded, :);
```

Derivatives

```
[~, rows_loaded, rows_current] = intersect ...
  (save_struct.derivative_data(:, 1), ...
  app.UITable_derivative.Data(:, 1));

app.UITable_derivative.Data(rows_current, :) = ...
  save_struct.derivative_data(rows_loaded, :);
```

Outputs

```
[~, rows_loaded, rows_current] = intersect ...
  (save_struct.output_data(:, 1), ...
  app.UITable_output.Data(:, 1));

app.UITable_output.Data(rows_current, :) = ...
  save_struct.output_data(rows_loaded, :);
```

We want to warn the user if there are entities in the loaded trim point that are not present in the current trim point and vice versa. Therefore, we create a cell array containing all input, state, and output names from the loaded trim point

```
loaded = union (union ( ...
save_struct.input_data(:, 1), ...
save_struct.state_data(:, 1)), ...
save_struct.output_data(:, 1));
```

and another cell array containing all input, state, and output names from the current model:

```
current = union (union ( ...
app.UITable_input.Data(:, 1), ...
app.UITable_state.Data(:, 1)),  ...
app.UITable_output.Data(:, 1));
```

The `setdiff` command finds those names that are present in the loaded data but not in the current model

```
loaded_without_current = setdiff (loaded, current);
```

If there are more names in the loaded data

```
if ~isempty (loaded_without_current)
```

we inform the user:

```
  warndlg ([ ...
    'The following names are present in the file', ...
    newline, ...
    file_name, ...
    newline, ...
    'but not in the model', ...
    newline, ...
    app.model_name, ...
    ':'; ...
    loaded_without_current; ...
    'I have just loaded the intersection of the sets.'], ...
    'Warning')
end
```

Informing the user about those names that are present in the current model but not in the loaded data is done in a similar way:

```
current_without_loaded = setdiff (current, loaded);

if ~isempty (current_without_loaded)
  warndlg ([ ...
    'The following names are present in the model', ...
    newline, ...
    app.model_name, ...
    newline, ...
    'but not in the file', ...
    newline, ...
    file_name, ...
    ':'; ...
    current_without_loaded; ...
    'I have just loaded the intersection of the sets.'], ...
    'Warning')
end
```

Finally, we update the number of trim variables and trim requirements:

```
update (app, [])
end
```

2.12 selection

If the user selects a name in the first column of a table, we want to open the corresponding block diagram and indicate the selected block. In

```
function selection(app, event)
```

we find the indices of the selected table element

```
indizes = event.Indices;
```

and check if the user selected a name (in the first column)

```
if indizes(2) == 1
```

If this is the case, we buffer[13] the name of the selected element:

```
  block_to_open = event.Source.Data{indizes(1), 1};
```

In order to get rid of potential ___1, ___2, ... of vector names, we split (those) at ___

```
  block_to_open = split (block_to_open, '___');
```

just take the first part and ignore the second part:

```
  block_to_open = block_to_open{1};
```

Elements in the Derivative table contain Deriv. of which is not part of the block name:

```
  block_to_open = erase (block_to_open, 'Deriv. of ');
```

Finally, we add the model name to the block to be opened:

```
  block_to_open = [app.model_name, '/', block_to_open];
```

Next, we want to open the block diagram containing the block to be opened. Therefore, we find the indices of all slashes in the name of the block to be opened

```
  slashes = strfind (block_to_open, '/');
```

and determine the index of the last slash:

```
  last_slash = slashes (end);
```

The system (block diagram) to be opened is the whole path to the block to be opened up to the last slash:

```
  system_to_open = block_to_open(1 : last_slash - 1);
```

[13] The following remodifications are necessary because we modified the block names in the tables for better readability; now we have to take those modifications back. As an alternative, we could have stored the original block names together with the modified names ...

The system has to be loaded before one of its blocks can be opened:

```
  load_system (app.model_name)
```

Finally, we can open the block diagram containing the block to be opened

```
  open_system (system_to_open);
```

We blink the selected block five times:

```
  for i_blink = 1 : 5
```

In a loop that toggles between on and off

```
    for selection = {'on', 'off'}
```

we cycle the selection of the block in question

```
      set_param (block_to_open, 'Selected', selection{1})
```

and wait 100 ms before we toggle back:

```
      pause (0.1)
    end
  end
end
end
```

2.13 TrimMenuSelected

After the user has loaded a model and defined a valid trim point, they can select the *Action* / *Trim* menu entry that calls the corresponding method:

```
function TrimMenuSelected(app, event)
```

For easier access, we copy the current values of the states, inputs, derivatives, and outputs to app properties with appropriate names:

```
app.x = cell2mat (app.UITable_state.Data(:, 2));
app.u = cell2mat (app.UITable_input.Data(:, 2));
app.d = cell2mat (app.UITable_derivative.Data(:, 2));
app.y = cell2mat (app.UITable_output.Data(:, 2));
```

The same is done with the indices of all trim variables and trim requirements

```
app.i_x = find (cell2mat (app.UITable_state.Data(:, 3)));
app.i_u = find (cell2mat (app.UITable_input.Data(:, 3)));
app.i_d = find (cell2mat (app.UITable_derivative.Data(:, 3)));
app.i_y = find (cell2mat (app.UITable_output.Data(:, 3)));
```

and the names of inputs, states, derivatives and outputs:

```
app.x_nam = app.UITable_state.Data(:, 1);
app.u_nam = app.UITable_input.Data(:, 1);
app.d_nam = app.UITable_derivative.Data(:, 1);
app.y_nam = app.UITable_output.Data(:, 1);
```

We read the *Maximum Trim Step* values from the states and inputs tables

```
del_x_max = cell2mat (app.UITable_state.Data(:, 4));
del_u_max = cell2mat (app.UITable_input.Data(:, 4));
```

and replace every zero-element with the default value (`1e42`):

```
del_x_max(~del_x_max) = 1e42;
del_u_max(~del_u_max) = 1e42;
```

The same is done with the *Linearization Step* values

```
del_x_lin = cell2mat (app.UITable_state.Data(:, 5));
del_u_lin = cell2mat (app.UITable_input.Data(:, 5));

del_x_lin(~del_x_lin) = 1e-6*(1 + abs (app.x(~del_x_lin)));
del_u_lin(~del_u_lin) = 1e-6*(1 + abs (app.u(~del_u_lin)));
```

the *Minimum Values*

```
x_min = cell2mat (app.UITable_state.Data(:, 6));
u_min = cell2mat (app.UITable_input.Data(:, 6));

x_min(~x_min) = -inf;
u_min(~u_min) = -inf;
```

and the *Maximum Values*:

```
x_max = cell2mat (app.UITable_state.Data(:, 7));
u_max = cell2mat (app.UITable_input.Data(:, 7));

x_max(~x_max) = inf;
u_max(~u_max) = inf;
```

We save the old, pre-trim values in a structure

```
app.pre_trim.state = app.UITable_state.Data(:, 2);
app.pre_trim.input = app.UITable_input.Data(:, 2);
app.pre_trim.derivative = app.UITable_derivative.Data(:, 2);
app.pre_trim.output = app.UITable_output.Data(:, 2);
```

and finally call the actual trim algorithm:

```
[x_tr, u_tr, d_tr, y_tr, ~, app.info_struct] = jj_trim ( ...
app.model_name, ...
app.x, app.u, app.d, app.y, ...
app.i_x, app.i_u, app.i_d, app.i_y, ...
app.x_nam, app.u_nam, app.d_nam, app.y_nam, ...
del_x_max, del_u_max, del_x_lin, del_u_lin, ...
app.parameters, ...
x_min, x_max, u_min, u_max);
```

If the trim algorithm returns a valid info structure containing the progressions of all states, inputs, derivatives, and outputs

```
if ~isempty (app.info_struct)
```

we enable the *Action | Untrim* and the *Options | Show Overview* menu entries:

```
  app.UntrimMenu.Enable = true;
  app.ShowOverviewMenu.Enable = true;
end
```

Next, we copy the trimmed states, inputs, derivatives, and outputs back into the corresponding tables:

```
app.UITable_state.Data(:, 2) = num2cell (x_tr);
app.UITable_input.Data(:, 2) = num2cell (u_tr);
app.UITable_derivative.Data(:, 2) = num2cell (d_tr);
app.UITable_output.Data(:, 2) = num2cell (y_tr);
```

Transfer trim point to model

Life gets a bit tricky here: We want to use the new trim point as the initial simulation values (states and inputs) of the current Simulink model. For that purpose, we can use the *Input* and *Initial state* edit texts in the *Modeling | Model Settings | Data Import/Export | Configuration Parameters | Load from workspace* menu entry of the model.

Unfortunately, Simulink claims[14] the right to alter the order of the states during the simulation. Therefore, we cannot simply use the trimmed state vector (`x_tr`) in the *Initial state* edit text, but we have to convert the vector to a structure that additionally features the corresponding state names. Additionally, in order to save the trim point together with the model, we have to save this trim point structure in the model workspace.

First, we add a zero as the *time span* trailing the input vector[15]:

```
u_tr_with_zero = [0, u_tr'];
```

[14] In [1], the Simulink documentation warns: "Avoid using an array for an initial state. If the order of the elements in the array does not match the order in which blocks initialize, the simulation can produce unexpected results."

[15] Simulink does not (yet) warn about using a vector as the initial input.

Next, we convert the state trim vector to a string with maximum precision:[16]

```
x_tr_string = mat2str (x_tr, 42);
```

and enable the checkboxes in the *Initial state* and *Input* menu entry of the current model

```
set_param (app.model_name, 'LoadInitialState', 'on');
set_param (app.model_name, 'LoadExternalInput', 'on');
```

The next three steps may seem a bit bold and may not survive every future Simulink version; but at least right now this seems to be the most elegant way to convert the initial state vector to its corresponding structure:

We transfer the state trim vector to the corresponding edit text

```
set_param (app.model_name, 'InitialState', x_tr_string);
```

set the `SaveFormat` to `Structure`

```
set_param (app.model_name, 'SaveFormat', 'Structure');
```

and reread[17] the initial state back as a structure into a new variable:

```
x_tr_structure = ...
Simulink.BlockDiagram.getInitialState (app.model_name);
```

We retrieve a handle to the model workspace

```
model_workspace = get_param (app.model_name, 'ModelWorkspace');
```

and write the initial state (as a structure) and the initial input (as a vector) to the model workspace:

```
assignin (model_workspace, ...
'initial_state_in_model_workspace', x_tr_structure)

assignin (model_workspace, ...
'initial_input_in_model_workspace', u_tr_with_zero)
```

Finally, we use these model workspace variables in the *Input* and *Initial state* edit texts:

```
set_param (app.model_name, ...
'ExternalInput', 'initial_input_in_model_workspace');

set_param (app.model_name, ...
'InitialState', 'initial_state_in_model_workspace');
```

[16] A postulated precision of 42 seems to give more significant(?) digits than the recommended conversion without any postulated precision.

[17] The `getInitialState` command reads the initial state vector from the *Initial state* edit text, automatically converts it into a suitable structure including the state names and returns the structure.

2.14 UntrimMenuSelected

If the user is not satisfied with the trim result, they can use the *Action* / *Untrim* menu entry that calls the corresponding method

```
function UntrimMenuSelected(app, event)
```

that tries to reestablish the pre-trim state. We disable the *Action* / *Untrim* menu entry because only **one** untrim history back-step is possible

```
app.UntrimMenu.Enable = false;
```

and copy the pre-trim states, inputs, derivatives, and outputs into the corresponding tables:

```
app.UITable_state.Data(:, 2) = app.pre_trim.state;
app.UITable_input.Data(:, 2) = app.pre_trim.input;
app.UITable_derivative.Data(:, 2) = app.pre_trim.derivative;
app.UITable_output.Data(:, 2) = app.pre_trim.output;
end
```

2.15 ShowOverviewMenuSelected

Selecting the *Options* / *Show Overview* menu entry calls the corresponding method

```
function ShowOverviewMenuSelected(app, event)
```

that deletes a potentially remaining overview app

```
delete (app.overview_app);
```

and opens a new overview table (chapter 4) that displays the progressions of all states, inputs, derivatives, and outputs:

```
app.overview_app = trimmod_overview (app);
end
end
```

Additionally, the table displays the cost function, the step type, the bisection counter, the step limitation, and the Jacobian matrix for every iteration step.

3 trimmod_parameters app

If the user selects the *Options | Additional Parameters* menu entry (section 2.6) in the `trimmod` app, a separate app

```
classdef trimmod_parameters < matlab.apps.AppBase
```

is opened that allows the definition of a few general trim parameters (figure 3.1)

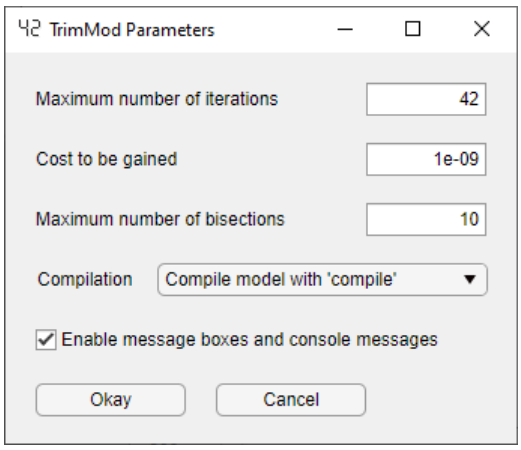

Figure 3.1: Default values of the additional parameters

In the app, we define a single private property that will allow us to access the calling app

```
properties (Access = private)
    trimmod
end
```

and three private methods (`startupFcn`, `OkayButtonPushed`, `CancelButtonPushed`):

```
methods (Access = private)
```

3.1 startupFcn

The `startupFcn` method of the `trimmod_parameters` app

```
function startupFcn(app, trimmod)
```

saves the name of the calling app (trimmod) in a property in order to allow the OkayButtonPushed method (section 3.2) to access the public properties of the trimmod app (chapter 2) directly:

```
app.trimmod = trimmod;
```

We move the lower left vertex of the parameters dialog figure to the lower left vertex of the TrimMod figure[1]

```
app.TrimModParametersUIFigure.Position(1 : 2) = ...
app.trimmod.TrimModelUIFigure.Position(1 : 2);
```

and populate the *EditFields*, the *DropDownList*, and the *CheckBox* with the current values from the calling trimmod app:

```
app.MaximumnumberofiterationsEditField.Value = ...
  trimmod.parameters.n_iter_max;
app.CosttobegainedEditField.Value = ...
  trimmod.parameters.cost_tbg;
app.CompilationDropDown.Value = ...
  trimmod.parameters.CompileFlag;
app.MaximumnumberofbisectionsEditField.Value = ...
  trimmod.parameters.n_bisec_max;
app.EnablemessageboxesandconsolemessagesCheckBox.Value = ...
  trimmod.parameters.EnableMessages;
end
```

3.2 OkayButtonPushed

If the user pushes the *Okay* button in the trimmod_parameters app

```
function OkayButtonPushed(app, event)
```

we write the values from the *EditFields*, the *DropDownList*, and the *CheckBox* back to the calling trimmod app

```
app.trimmod.parameters.n_iter_max = ...
  app.MaximumnumberofiterationsEditField.Value;
app.trimmod.parameters.cost_tbg = ...
  app.CosttobegainedEditField.Value;
app.trimmod.parameters.CompileFlag = ...
  app.CompilationDropDown.Value;
app.trimmod.parameters.n_bisec_max = ...
  app.MaximumnumberofbisectionsEditField.Value;
app.trimmod.parameters.EnableMessages = ...
  app.EnablemessageboxesandconsolemessagesCheckBox.Value;
```

[1] This might be useful if the user (has more than one monitor and) has moved the TrimMod figure (to a different monitor).

and close the app:

```
delete (app)
end
```

3.3 CancelButtonPushed

If the user pushes the *Cancel* button in the trimmod_parameters app

```
function CancelButtonPushed(app, event)
```

we just close the app:

```
delete (app)
end
end
```

4 `trimmod_overview` app

If the user selects the *Options* / *Show Overview* menu entry in the `trimmod` app, a separate app

```
classdef trimmod_overview < matlab.apps.AppBase
```

is opened that displays the intermediate trim results of every iteration step (figure 4.1).

States, Inputs, Derivatives, Outputs, Cost, ...	Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _…	0	0.0587	0.0589	0.0589	0.0589
A380 + Wind + Bahn/A380/Kinetik/Lage- Integrator _…	0	0	0	0	0
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass	0	0	0	0	0
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass1	0	0	0	0	0
A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass2	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits…	200	200	199.6532	199.6532	199.6532
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Geschwindigkeits…	0	11.7465	11.7733	11.7733	11.7733
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindig…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindig…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Drehgeschwindig…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrat…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrat…	0	0	0	0	0
A380 + Wind + Bahn/A380/Kinetik/Positions- Integrat…	0	0	0	0	0
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 4/B…	0	6.5302e+04	4.3067e+04	4.3068e+04	4.3068e+04
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 3/B…	0	6.5302e+04	4.3067e+04	4.3068e+04	4.3068e+04
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 2/B…	0	6.5302e+04	4.3067e+04	4.3068e+04	4.3068e+04
A380 + Wind + Bahn/A380/Triebwerke/Triebwerk 1/B…	0	6.5302e+04	4.3067e+04	4.3068e+04	4.3068e+04
F_trim	50000	6.5302e+04	4.3067e+04	4.3068e+04	4.3068e+04
ze_trim	0	0	0	0	0
et_trim	0	-0.0159	-0.0184	-0.0184	-0.0184
xi_trim	0	0	0	0	0
V_A_c_trim	200	200	200.0000	200.0000	200.0000
be_c_trim	0	0	0	0	0
th_c_trim	0	0.0587	0.0589	0.0589	0.0589
ph_c_trim	0	0	0	0	0
H_c_trim	0	0	0	0	0
ch_c_trim	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Lage- Int…	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Lage- Int…	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Lage- Int…	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/Wind/Turbulenz/Tiefpass	1.1650	1.1650	1.1650	1.1650	1.1650
Deriv. of A380 + Wind + Bahn/Wind/Turbulenz/Tiefpa…	0.9794	0.9794	0.9794	0.9794	0.9794
Deriv. of A380 + Wind + Bahn/Wind/Turbulenz/Tiefpa…	0.5987	0.5987	0.5987	0.5987	0.5987
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Geschwi…	-0.2803	0.2447	-4.0950e-06	9.0913e-12	1.1102e-16
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Geschwi…	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Geschwi…	5.4411	-0.0388	-8.5529e-05	-7.3133e-12	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Drehges…	0	0	0	-3.2193e-20	-3.2193e-20
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Drehges…	-0.0194	-1.8120e-05	-1.7904e-05	-1.9278e-12	-3.9705e-18
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Drehges…	0	0	0	-1.0164e-18	-1.0164e-18
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Positions…	200	200.3447	200.0000	200.0000	200.0000
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Positions…	0	0	0	0	0
Deriv. of A380 + Wind + Bahn/A380/Kinetik/Positions…	0	-0.0135	8.1049e-05	4.7073e-13	0
Deriv. of A380 + Wind + Bahn/A380/Triebwerke/Trieb…	1.6667e+04	-1.1488e+03	-0.0210	8.8665e-08	-2.4253e-12
Deriv. of A380 + Wind + Bahn/A380/Triebwerke/Trieb…	1.6667e+04	-1.1488e+03	-0.0210	8.8665e-08	-2.4253e-12
Deriv. of A380 + Wind + Bahn/A380/Triebwerke/Trieb…	1.6667e+04	-1.1488e+03	-0.0210	8.8665e-08	-2.4253e-12
Deriv. of A380 + Wind + Bahn/A380/Triebwerke/Trieb…	1.6667e+04	-1.1488e+03	-0.0210	8.8665e-08	-2.4253e-12
V_A	200	200.3447	200.0000	200.0000	200.0000
al	0	0.0587	0.0589	0.0589	0.0589
be	0	0	0	0	0
V_K	200	200.3447	200.0000	200.0000	200.0000
ga	0	6.7393e-05	-4.0524e-07	-2.3537e-15	0
ch	0	0	0	0	0
del_F_c	0	-3.4465e+03	-0.0555	-6.8212e-09	0
del_ze_c	0	0	0	0	0
del_et_c	0	1.9429e-16	0	0	0
del_xi_c	0	0	0	0	0
del_th_c	0	0	0	0	0
del_ph_c	0	0	0	0	0
Cost function	1.6667e+04	3.4465e+03	0.0555	8.8665e-08	2.4253e-12
Step type	0	1	1	1	1
Bisection counter	0	0	0	0	0
Step limitation	0	0	0	0	0
Jacobian	Click me!	Click me!	Click me!	Click me!	

Figure 4.1: Intermediate iteration step results

Just like in the `trimmod_parameters` app (chapter 3), we define a single private property that will allow us to access the calling app

```
properties (Access = private)
    trimmod
end
```

and two private methods (startupFcn, UITableCellSelection):

```
methods (Access = private)
```

4.1 startupFcn

The startupFcn method of the trimmod_overview app

```
function startupFcn(app, trimmod)
```

saves the name of the calling app (trimmod) in a property in order to allow the CellSelection method (section 4.2) to access the public properties of the trimmod app (chapter 2) directly:

```
app.trimmod = trimmod;
```

We position the overview figure "over" the TrimMod figure,

```
app.TrimModOverviewUIFigure.Position = ...
  app.trimmod.TrimModelUIFigure.Position;
```

fill the whole Overview figure with the Overview table

```
app.UITable.Position = [ ...
    1, ...
    1, ...
    app.TrimModOverviewUIFigure.Position(3), ...
    app.TrimModOverviewUIFigure.Position(4)];
```

and receive the number of trim iterations from the calling trimmod app:

```
n_iter = trimmod.info_struct.n_iter;
```

The remaining lines of code populate the table displayed in figure 4.1. The heading of the first table column is static:

```
app.UITable.ColumnName{1} = ...
  'States, Inputs, Derivatives, Outputs, Cost, ...';
```

For the remaining headings we start a loop over all iterations:

```
for i_iter = 0 : n_iter
```

and generate the heading of each data column dynamically:

```
  app.UITable.ColumnName{i_iter + 2} = ...
    ['Iteration ', num2str(i_iter)];
end
```

Each table row represents the iterated values of states, inputs derivatives, outputs, and other potentially interesting variables. For better interpretability, we prepare a column holding the variable names

```
names = [ ...
  trimmod.x_nam; ...
  trimmod.u_nam; ...
  trimmod.d_nam; ...
  trimmod.y_nam; ...
  'Cost function'; ...
  'Step type'; ...
  'Bisection counter'; ...
  'Step limitation'; ...
  'Jacobian'];
```

initialize the content of the table as a cell array of proper size

```
app.UITable.Data = cell (numel (names), n_iter + 2);
```

and use the names as the first table column:

```
app.UITable.Data(:, 1) = names;
```

The remaining columns hold the data. We prepare the data to be inserted into the table:

```
data = num2cell ([ ...
  trimmod.info_struct.x; ...
  trimmod.info_struct.u; ...
  trimmod.info_struct.d; ...
  trimmod.info_struct.y; ...
  trimmod.info_struct.cost; ...
  0, trimmod.info_struct.StepType; ...
  0, trimmod.info_struct.BisecCounter; ...
  0, trimmod.info_struct.LimitedStep]);
```

The last row of the table shall contain *Click me!* buttons (figure 4.1) since the Jacobians are matrices themselves that cannot be displayed inside the table. If there is no iteration because the initial Jacobian is singular

```
if n_iter == 0
```

we just need one button for the initial Jacobian:

```
  data = [data; {'Click me!'}];
```

If there are iterations

```
else
```

the last iteration does not return a Jacobian:

```
  data = [data; repmat({'Click me!'}, 1, n_iter), {''}];
end
```

Finally, we insert the prepared data into the table:

```
app.UITable.Data(:, 2 : end) = data;
end
```

4.2 UITableCellSelection

The cell selection callback method of the table

```
function UITableCellSelection(app, event)
```

is called if the user selects any cell of the table. In that case, we buffer the row and column indices of the currently selected cell

```
indices = event.Indices;
```

and check if the user has selected one of the *Click me!* buttons

```
if strcmp (app.UITable.Data{indices(1), indices(2)}, 'Click me!')
```

If this is the case, we delete a potentially open Jacobian figure

```
  delete (app.trimmod.jaco_figure)
```

and open a new figure to display the current Jacobian.

We move its lower left vertex 100 pixel up with respect to the lower left vertex of the Overview figure, in order to keep the *Click me!* row of the Overview app visible:

```
app.trimmod.jaco_figure = uifigure ('Position', [ ...
  app.TrimModOverviewUIFigure.Position(1), ...
  app.TrimModOverviewUIFigure.Position(2) + 100, ...
  app.TrimModOverviewUIFigure.Position(3), ...
  app.TrimModOverviewUIFigure.Position(4) - 100]);
```

We adjust the figure title

```
app.trimmod.jaco_figure.Name = ...
  ['Jacobian at iteration step ', num2str(indices(2)) - 2];
```

and create an empty table in the figure:

```
table_jaco = uitable (app.trimmod.jaco_figure, ...
  'Position', [1, 1, 1300, 700]);
```

We build the generalized input names vector by concatenating the state names and the input names

```
x_u_nam = [app.trimmod.x_nam; app.trimmod.u_nam];
```

and the generalized output names vector by concatenation of the derivative names and the output names:

```
d_y_nam = [app.trimmod.d_nam; app.trimmod.y_nam];
```

We find the indices of the trim variables in the generalized input names vector

```
index_trim_variables = ...
  [app.trimmod.i_x; length(app.trimmod.x) + app.trimmod.i_u];
```

and the indices of the trim requirements in the generalized output names vector

```
index_trim_requirements = ...
  [app.trimmod.i_d; length(app.trimmod.d) + app.trimmod.i_y];
```

and use the current Jacobian as the data of the table:

```
table_jaco.Data = num2cell ...
  (app.trimmod.info_struct.jaco{indices(2) - 1});
```

Finally, we use the names of the trim requirements as the row names of the table

```
table_jaco.RowName = d_y_nam (index_trim_requirements);
```

and the names of the trim variables as the column names of the table

```
table_jaco.ColumnName = x_u_nam (index_trim_variables);
```

and give all columns an equal width:

```
table_jaco.ColumnWidth = '1x';
end
end
end
```

5 jj_trim

This chapter does not explain every single line of code of `jj_trim`; we merely focus on the mathematical background of the trim algorithm:

A nonlinear time-invariant system can be described via its differential equation system

$$d = f(x, u) \tag{5.1}$$

and its output equation system

$$y = g(x, u) \tag{5.2}$$

where u is the input vector, x is the state vector, $d = \frac{dx}{dt}$ is the time derivative of the state vector, y is the output vector, and f and g are nonlinear vector functions, evaluated in every simulation time step. State vector x and input vector u are the independent variables on the right-hand side of the equations. Both vectors can be combined into a generalized input vector

$$xu = \begin{bmatrix} x \\ u \end{bmatrix}$$

Derivative vector d and output vector y are the left-hand side results of the function evaluations. They can be combined into the generalized output vector

$$dy = \begin{bmatrix} d \\ y \end{bmatrix}$$

Both equation systems can then be combined into

$$dy = h(xu) \tag{5.3}$$

where

$$h = \begin{bmatrix} f \\ g \end{bmatrix}$$

is the generalized system vector function.

To start a simulation, all elements of the generalized input vector xu (the complete x and u vectors) have to be known for the first evaluations of equation (5.3). Unfortunately, the trim point is often defined as a mixture of u, x, d, and y: The initial speed (x) of

a car might be known, but not the corresponding engine power or the accelerator angle (u) for no acceleration (d). The radius of the curve might be predefined, but not the corresponding turning wheel angle, ... Usually the user initially defines some elements of the generalized output vector dy that have to be satisfied, and some elements of the generalized input vector xu that are known a priori. The other (unknown) elements of the generalized input vector xu have to be found by the trim algorithm. The unknown elements of the generalized output vector dy can then easily be calculated via equation (5.3) if xu is completely known.

Both generalized vectors can therefore be split up into a known (subscript k) and an unknown (subscript n) part:

$$dy = \begin{bmatrix} dy_k \\ dy_n \end{bmatrix} \tag{5.4}$$

$$xu = \begin{bmatrix} xu_k \\ xu_n \end{bmatrix} \tag{5.5}$$

Accordingly, equation (5.3) too can be split up into two (vector) equations, one for the predefined elements of dy and one for the unknowns:

$$dy_k = h_k(xu) = h_k\left(\begin{bmatrix} xu_k \\ xu_n \end{bmatrix}\right) \tag{5.6}$$

$$dy_n = h_n(xu) = h_n\left(\begin{bmatrix} xu_k \\ xu_n \end{bmatrix}\right) \tag{5.7}$$

The trim algorithm now has to solve the nonlinear equation system (5.6) with respect to the unknown vector xu_n (called the trim variables vector), while the vector dy_k is called the trim requirements vector.

Trim requirements dy_k Those (known) elements of the generalized output vector dy that have to be satisfied

Trim variables xu_n Those (unknown) elements of the generalized input vector xu that the trim algorithm is free to vary

For a unique solution of equation (5.6), the number of (unknown) trim variables (length of xu_n) has to equal the number of equations, given by the number of trim requirements (length of dy_k).

If this prerequisite is fulfilled, TrimMod (the graphical user interface) calls `jj_trim` (the actual trim algorithm).

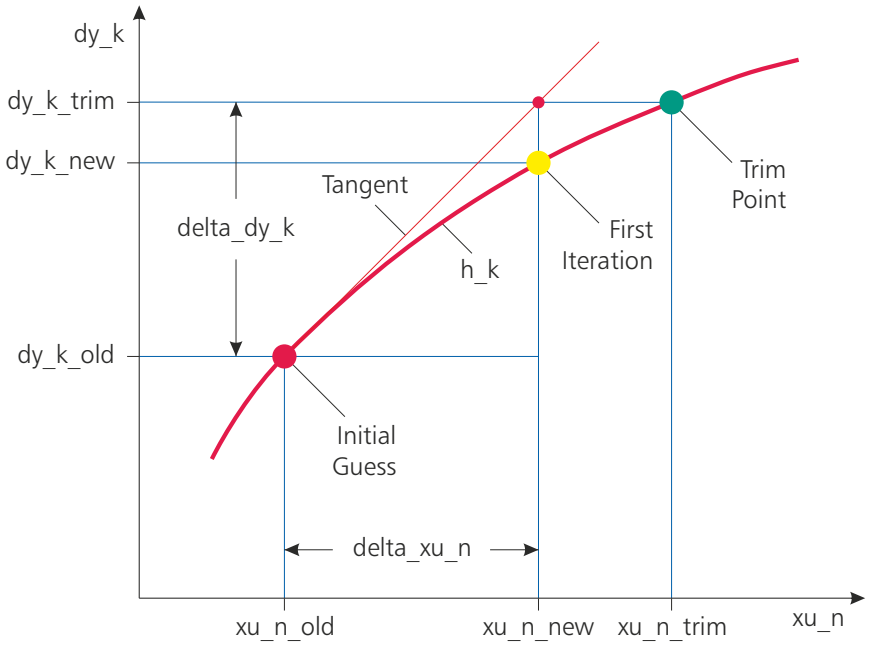

Figure 5.1: Newton-Raphson trim algorithm

As shown in figure 5.1, the first step of `jj_trim` is to insert the initial guess of the trim variable vector $xu_{n_{old}}$ on the right-hand side of equation (5.6) and to check whether the trim requirement vector $dy_{k_{trim}}$ is already met by $dy_{k_{old}}$. As this is usually not the case, a modified multidimensional Newton-Raphson algorithm is used to iteratively find new trim variable vectors $xu_{n_{new}}$ that – hopefully – finally approach the sought $xu_{n_{trim}}$.

Newton-Raphson relies on the local derivatives which can graphically be represented as a tangent hyperplane in the multidimensional case. The linearization routine `jj_lin` finds the gradients of this tangent hyperplane at $xu_{n_{old}}$ and returns a Jacobian sensitivity matrix $jaco$, which represents the linear relation

$$\Delta dy_k = jaco \cdot \Delta xu_n \tag{5.8}$$

of the trim requirement error

$$\Delta dy_k = dy_{k_{trim}} - dy_{k_{old}} \tag{5.9}$$

with respect to the trim variable correction

$$\Delta xu_n = xu_{n_{new}} - xu_{n_{old}} \tag{5.10}$$

A singular system decomposition (singular values and singular vectors) of the sensitivity matrix jaco is done in order to find trim variables that have no influence on any trim requirement, trim requirements that cannot be influenced by any trim variable, and linear

dependencies of trim variables or trim requirements. One or more singular values of zero indicate a wrong choice of trim requirements and/or trim variables. The corresponding singular vectors clearly show which trim requirements and trim variables are responsible for the rank deficiency. This detailed information can then be used to chose those trim requirements and trim variables that correctly describe the desired trim state.

If the sensitivity matrix *jaco* has full rank (is non-singular), the linear equation system (5.8) can be solved via Matlab's backslash operator:

$$\Delta xu_n = jaco \backslash \Delta dy_k$$

and equation (5.10) can be used to find the next solution vector:

$$xu_{n_{new}} = xu_{n_{old}} + \Delta xu_n$$

Bibliography

[1] Mathworks, "Documentation: Initial state," 2022. [Online]. Available: https://de.mathworks.com/help/simulink/gui/initial-state.html