Matlab Framework for nonlinear state space dynamics system models

For Users

Overview

Many techniques in control theory require a linear system description. That's why one often linearizes nonlinear models for the purpose of controller synthesis. However the controller in the end needs to be applied to the real world (and most of the time nonlinear) system. To do so one would like to simulate the nonlinear closed loop. While the builtin LTI framework in Matlab is really good, there is little to no support for nonlinear models. The purpose of this tool therefore is to simplify the work with nonlinear state space dynamic system models in Matlab. An attempt was made to provide similar functionality to what is already possible for linear state space models. The nonlinear models are considered to consist of a vector field f determining the dynamics and an output map h specifying the output of the system:

```
x_dot = f(x, u)
y     = h(x, u)
```

Handling of jacobians is implemented for most of the available operations, such that linearizing a nonlinear model at equilibria is possible, establishing the connection to the linear dynamic system framework already available in Matlab.

Implemented operations

- Constructing a `nlmodel` object from function handles f and h
- Constructing a `nlmodel` object from a Matlab ss object (results in a call to `ss2nlmodel`)
- Simulating the nonlinear system using `nlsim` (similar to `lsim`, but without plotting)
- Linearizing a nonlinear system at equilibrium using `nlmodel.linearize`
- Performing typical algebraic operations such as (jacobians are taken care of), just by using the same syntax used for ss-objects
    - addition (+, parallel interconnection)
    - multiplication (*, series interconnection)
    - horizontal ([,]) and vertical ([;]) concatenation
- Enhanced series interconnection (`series_nl`, not well tested!)
- And most important: upper LFT between a linear and a nonlinear model (`nl_upper_lft`)
- Assigning channel and state names
- Access to vector field f and output map h of the model at any time, e.g. for simulation
- Extensive consistency and error checking throughout all operations
- Detailed documentation, accessible by e.g. `help nlmodel`, `help nlmodel.plus`, ...

Examples

Two examples of a typical use case can be found in the files `examples/rocket_output_feedback_example.m` and `examples/rocket_output_feedback_example.m`. There a nonlinear 2D model of a rocket is used and a state feedback as well as an output feedback controller are being applied to the model. Then the simulation of the linear and nonlinear closed loop are being compared in a plot. The model details can be found in `rocket_model.pdf`. The model is taken from 'A Robust Control Approach for Rocket Landing' by Reuben Ferrante, 2017, University of Edinburgh.

Flaws of the non symbolic approach

Due to the fact that the program does not handle the dynamics symbolically, it cannot simplify expressions. This means that when combining `nlmodel` objects using the provided algebraic expressions, the resulting function handles determining the dynamic behaviour of the systems become increasingly complex and nested. There is no simple way to extract the mathematical expression defining a model after some interconnections have been done. This also results in increasing numerical errors when making extensive use of combining `nlmodel` objects.

Set-Up: Matlab Path

The required paths are "nlmodel" for the main functionality, "nlmodel/tests" for the tests and "nlmodel/examples" for the examples. These folders need to be added to the Matlab search path using `addpath '.../nlmodel'`, `addpath '.../nlmodel/tests'`, `addpath '.../nlmodel/examples'` and then `savepath` to make the changes permanent (might require administrator privileges).

Technical details

The code contains very detailed comments, so this section is only meant to give an overview regarding the implementation. For further information one should take a look at the code itself.

Object oriented approach

The implementation uses an object oriented approach to represent nonlinear models in Matlab. I tried to (at least at a superficial level) mimic the existing Matlab I/O model classes, which however are way more sophisticated and complex. The file `nlmodel.m` contains the class definition for an `nlmodel` object.

Consistency checking and error handling

To maintain consistency of models (regarding dimensions and channel names) we make heavy use of the concept of property validation, which is a more efficient and secure way to ensure data consistency than argument validation, although this concept is also used at some places. Errors and warnings are issued through the Matlab Exception Handling Framework and I tried to make them as helpful as possible.

Column vectors

Throughout all the code only column vectors are used. This is very important, because otherwise there would arise difficult to trace problems when dealing with systems having zero state, input and/or output dimension.

Nested functions for jacobians

Since the resulting jacobians are quite complicated and depend on many system properties when interconnecting systems, it makes the code much more readable to encompass them into separate functions.

That's why the jacobians are implemented as nested functions in most cases. This however leads to the fact that the scope of some variables increases into the nested function which causes Matlab to depict those variables in a blue font. It shall only be said that this doesn't cause any problems.

Testing

The code has been thouroughly tested. An automated approach has been used for testing, by creating random state space models (builtin Matlab function rss) and performing the same operations on the original Matlab ss objects and on nlmodel objects created from the ss objects. The Matlab ss framework can be assumed to be correct and therefore serves as reference. The resulting models are then compared using different measures: Comparing the channel names, the system matrices, sampling both f and h and comparing the state and output trajectories obtained by simulation. The errors remain negligible (less than 10e-11) in all cases except for the simulation trajectories of the upper_lft closed loops. This is due to numerical errors being massively amplified during the integration. The test script can be found at `tests/nlmodelTestRepeated.m` including usage details. In addition, identified errors have been reproduced using the Unit Test Framework from Matlab before being fixed and this way it can be ensured that those specific errors don't occur again. The unit tests can be found at `tests/nlmodelTest.m`. To compare a linear (`ss`) and a nonlinear (`nlmodel`) system, we use the function `compareToLinearSS`. This function contains the details of the comparison approach (norms etc.).

Mathematical derivation of operations

The mathematical details of the implemented operations can be found in the file `mathematical_details_of_implementation.pdf`.

Plotting function

A simple plotting function (`plot_sols`, or `plot_sols2` providing more options) is available as well, however it has not been tested extensively nor is it well documented. The function `add_sol` is used to create the needed data structure. See the examples for usage details.