Structure and Usage of robosim-solver.py

Mark A Post <mark.post@york.ac.uk>, 2020

The robosim-solver.py is a tool written in Python 3 with SymPy for importing XML from the RoboSim SDL schema and solving equations contained therein for physical components of a robot structure.

Structure

The robosim-solver.py code is written in three main parts, explained step by step below. Global variables are initialized to empty lists or zero on start.

XML Import

The tkinter library is used to select .model files to open. A loop “while filePath” at the beginning of the program continues to open and parse .model files and their equations into the workspace until the “Cancel” button is selected, after which the rest of the program is run.

Importing XML from .model files uses the ElementTree library for compatibility. Parsing the XML tags is done through list comprehension with a set of functions that return the operators associated with equations, e.g. “RCEquals()” that returns the ‘=’ character. The dictionary SymMap maps XML tags (e.g. “robochart:Equals”) to these characters, which then produce equations in textual format (e.g. a=2*b).

This is done through a tree traversal for each .model file opened. First, the searchElements() function is called for each tag name in ElementTagsToSearch (currently 'interactiondevices' and 'element'). The searchElements() function finds all sub-tags of these tags named 'inputs', 'outputs', 'locals', 'constants', and 'equations'. For each sub-tag in these sub-tags, the parseVariable() function simply adds the name of each variable for variables to the global lists inputVariableSymbols, outputVariableSymbols, localVariableSymbols, and constantVariableSymbols. For equations, the parseEquation() function performs a recursive traversal of equation trees and use the SymMap dictionary mappings to produce a text string of the equation, which is placed in the equationSymbols list of textual equations.

Equation Solution

The main step is to then solve the textual equations using the sympy library of algebraic functions and solvers. Export of equations for SymPy is done by generating text strings that are then defined in the workspace with exec(), as if they were written as text by hand in the Python console. The current procedure is as follows:

1. Constant variables are exported as simple variables by the exportSymbolCode() function that uses sympy symbols() on each variable name. This includes ‘C1’ and ‘C2’ that are often used in differential equation solutions (but not defined as constants by sympy).

2. Input variables, output variables, and local variables are assumed to be implicit functions of time and exported by exportFunctionCode() as functions of the variable ‘t’ using sympy Function(). This is necessary for solution of differential equations.
3. General solution of linear and differential equations in sympy often requires them to be defined as equalities. All equations are defined as equalities using the exportEqualityCode() function that splits equations into an lhs and rhs around their ‘=’ equality character and creates “equalityCode” strings that include sympy Eq(lhs,rhs) to define the equality.
    - If discretization is desired, the discretizedEqualityCode() function embeds the equalityCode string in SymPy as_finite_diff() which converts differential expressions of a single variable to fixed finite differences in a single variable.
    - These strings are then inserted into the global array “eqs” (as text strings, because SymPy seems to have no good way of embedding then using an Eq() object from an array).
    - For command-line convenience and further manipulation, the Eq() objects themselves are also exported to the workspace assigned to variables “eqs_1”, “eqs_2”, etc.
4. The list of variables to be solved for is set in the “symbolsToSolveFor” global list. Currently to achieve as complete a solution as possible this includes all time-dependent variables (outputVariableSymbols+localVariableSymbols+inputVariableSymbols). For each variable in this list, all equalities in the “eqs” list are checked to see if the current variable is contained in the equation. If the variable appears in the equation, then it is solved for that variable in one of the following manners:
    - If the string “Derivative” appears next to the variable, the equation is assumed to require a differential equation solution, and the solverDsolveCode() function is used to produce a string that runs SymPy dsolve() on the equality for the variable.
    - If not, the equation is assumed to require an algebraic solution, and the solverSolvesetCode() function is used to produce a string that runs SymPy solveset() on the equality for the variable.

    The solution is then checked for validity. If an EmptySet, ConditionSet, or Complement is produced, the solution is discarded as it is unlikely to be useful. If the solution is given as part of a FiniteSet it is extracted. The solution is then added to the dictionary “solutions” with the key as the variable that has been solved for. If other solutions to the variable exist, they are included in a list for that key variable. If the solution already exists in the list of solutions it is also discarded to prevent duplicates. The solutions for a given variable can then be found by indexing solutions with that variable, e.g. all solutions to variable ‘v’ in a list are produced by “solutions[‘v’]”.
5. Equations that isolate a single variable are then defined in the SymPy workspace. Each textual equation is checked in case a single variable appears on its right-hand side, if so the equation is reversed. Equations are then written to the workspace using Python exec(). This is to use the internal SymPy substitution capability to further substitute known variables into equations for chain solution (though at present making substitution work programmatically for XML export does not seem to work).

XML Export

Each variable with solutions in “solutions” is exported to XML as a tree of elements with the same schema as the input file. The textual expression for each solution variable is first parsed with the parseText() function that produces separate numeric elements and operators. The numerics and operators are then placed in order in an ElementTree structure using left, right, and expression tags.

Usage

To run this tool given a set of .model XML files:

1. Install Python 3.* and SymPy on Ubuntu 20.04 or something similarly-recent as 18.04 SymPy will not work (I think xml.etree and tkinter are usually built in to Python distributions)
2. Run Python3 robosim-solver.py - I use the Spyder IDE which is much like MATLAB and ideal for this
3. You will get a file selection box, select the .model files one at a time and when you are done press ESC or click Cancel (e.g. select Library.model, OK, then FootBot.model, OK, then Cancel and both of these will then be imported).
4. The console will show an expansion of the XML tree, show a log of the elements found, and then list the constants, variables and functions exported.
5. The equations will be converted to Equalities (named "eqs_0", "eqs_1"...) so that you can use them in the console and also in a list called "eqs". Note that I am currently exporting them to eqs as discretized equations using the "as_finite_diff" function in SymPy. They can be changed back to continuous form by commenting line 267 and uncommenting line 265 - maybe we make this a program option later.
6. The solver will then take each output variable and look for equations it appears in, then solving for the variable (using dsolve() if it is in a differential, and solveset() if not) and adding the result to a dictionary called "solutions". If you search the dictionary for an output's text string (e.g. '_as') you will get back a list of solutions for that variable.
7. Note that all the solution is done in the root workspace so after you run the programme in a Python console IDE like Spyder you can also experiment with solving the equations e.g. just enter "solveset(eqs_0,K)" to get a quadratic solution for the library motor model.
8. Solutions are parsed and exported in perhaps-compatible XML to the file 'solutions.model'