



**CZECH TECHNICAL
UNIVERSITY
IN PRAGUE**

F3

**Faculty of Electrical Engineering
Department of Control Engineering**

Bachelor's Thesis

Open Rapid Control Prototyping, Education and Design Tools

Dion Beqiri

beqirdio@gmail.com

May 2022

Supervisor: Ing. Pavel Píša, Ph.D.

Acknowledgement / Declaration

Thanks to...

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague 20. 5. 2022

.....

Abstrakt / Abstract

AbstractCZ

AbstractEN

/ Contents

1 Introduction	1
2 PysimCoder	2
2.1 Introduction to Pysim- Coder tool	2
2.2 Code generation path of pysimCoder	3
3 Vector support	7
3.1 Vectors in Control Systems . . .	7
3.2 Stable implementation of vectors in pysimCoder	8
3.3 Vectors with dynamic dimension-setting algorithm . . .	8
4 Support for Zynq based board in pysimCoder	9
5 RISC-V board with NuttX RTOS	10
6 Demonstration of Real Time Control System using PysimCoder	11
7 Conclusion	12
A Source Code	13
B Glossary	14
References	15

/ **Figures**

2.1	PysimCoder's front-end GUI	4
2.2	PysimCoder's Code Generation path.....	5
2.3	PysimCoder's Main Repository structure	6



Chapter 1

Introduction

Chapter 2

PysimCoder

2.1 Introduction to PysimCoder tool

“A control system is an interconnection of components forming a system configuration that will provide a desired system response.”[1, chapter 1]

We as a species have thrived upon the idea of controlling the processes around us. Of course, there are many natural occurrences that we cannot prevent nor avoid (yet), however we still maintain an increasing ability to control and automate all types of processes that can be used to our benefit. This concept of automated control systems has become the engine behind the rapid development of human technology, and concurrently has changed the way humans live and function as a society.[1] Therefore, as our technology advances further and faster, the necessity for automation solutions and engineers will only grow with it.

Due to the wide range of industries and fields which rely heavily on automation, there are various solutions and tools which offer the ability to design, simulate, and control real time systems. The software which is used all depends on the application and hardware which is being targeted. In an industrial setting, one would probably use TIA Portal or CODESYS to program the controllers of some production line in a factory. For designing and simulating complex electrical circuits, AutoDesk Fusion360 would be a perfect environment for high accuracy and modern solutions, while LTspice is more useful for simpler analog circuits.

For many fields of engineering, there is a corresponding software which is used for control design of a certain type of system, however for the purpose of this thesis I will focus on solutions which are meant for mathematical simulations, and code generation for microcontroller units. These include programs such as MATLAB and Simulink, which when used together make up a very elaborate tool for designing a system controlled by a microprocessor unit, and for visualizing the data coming from the physical system in real-time. In essence, Simulink is a graphical tool for creating block diagram schematics which represent a particular system using mathematics. Furthermore, users of Simulink can create their own blocks, not only for mathematical calculations, but even for control of physical hardware. Using this feature, one can combine the mathematical blocks with hardware-related blocks in order to generate binary code for execution on a MCU, which can then control different types of hardware connected to it.

Another such tool is PysimCoder, which has been the focal point of my diploma work. Started and developed by professor Roberto Bucher from the University of Applied Sciences and Arts of Southern Switzerland, PysimCoder is an open-source graphical tool used for real-time code generation, typically for microcontrollers and PCs which are controlling some physical peripherals. Additionally, it can also be used as a simulation tool for simple control schemes. Target operating systems include GNU/Linux with or without preemptive-rt kernel, and NuttX RTOS usually for constrained MCU based systems. At its core, it is a Rapid Prototyping Control application, similar in style to

programs like Simulink and XCos.[2] This application is still under development, and it has nowhere near the amount of investment or resources that its alternatives use. Nevertheless, this application has some advantages which should be considered for a control system tool.

Firstly, it is open-source and freely accessible, which allows complete introspection of the generated code, as well as the source code of the application itself. This way, if there is an error or an issue during usage, anyone with a good knowledge of programming can go inside the source and fix the bugs as needed. Another great advantage is that it takes much less memory space, which results in faster installation and overall usage. At the current moment, the whole project takes up about 100MB of space on my hard drive, which is negligible compared to the many gigabytes of space which Simulink requires. This is immediately noticed when comparing loading times between the two.

Probably the best thing about this program is that it uses Python as a programming language for most of the core functions. Using many Python libraries which are specialized in control systems mathematics, the application is able to design a schematic on a source code level. The program then translates this design into C programming language, which will then be deployed to a MCU which is capable of running C-code (most of them support it). The combination of Python and C languages is perfect for easy deployment in most environments, both for the host running PysimCoder, and for the target running the generated code.

PysimCoder is originally set up for installation in a Linux OS environment, however due to the many virtualization and containerization techniques used in computers today, the program can even be run in Windows OS, or Mac OSX. The program is so light and accessible that you can really run it from any consumer-level operating system, with barely any strain on the CPU. This, in combination with the free and open-source nature of the program makes it perfect for the software requirements of the future.

It is obvious by now that the paid and closed-source alternatives will have more features, stronger software stability, higher precision, and overall better performance. Nevertheless, pysimCoder still has a chance to become just as good, and still remain free and open-source. This will depend on students and researchers such as myself who will choose to develop further features and abilities to the program. One feature which pysimCoder would highly benefit from is being able to generate and use vector signals. This is something that Simulink already has full support for, therefore having it here will be one step further to a professional-level software, with virtually no financial cost.

In the next chapter, I will be explaining my journey to the extension of pysimCoder for support of vector signals and blocks with vector operations. Before that however, I must explain how the program works internally, and show the code-generation path from Python to C-code. This explanation will help to better understand the logic behind my implementation of the vector feature, as well as the other features added based on the requirements of this thesis.

2.2 Code generation path of pysimCoder

In order to fully understand the implementation of the features which I have added in the pysimCoder project, I must initially explain how the source code is organized and how it functions. When working on an open-source project, the greatest advantage is that you have complete access to the whole source of the application, however this comes with great risk as well. Any small change in the core functionality of the program can lead to unexpected results and errors. Therefore, the developer must always carefully

analyze and understand the source code before they start making changes and adding new functions.

From the front-end perspective, pysimCoder is a graphical tool used for designing block diagram schematics for the purpose of simulation and real-time code generation. Knowing this, it is only natural that I give a small preview of the GUI of the application. As seen in the image below, the pysimCoder tool is very similar to its larger and more costly counterparts such as Simulink or xCos as far as the graphical editor is concerned. In fact, the editor itself is based on the PySimEd project and the qtnodes-develop project, in combination with a lot of common blocks from PyEdit.

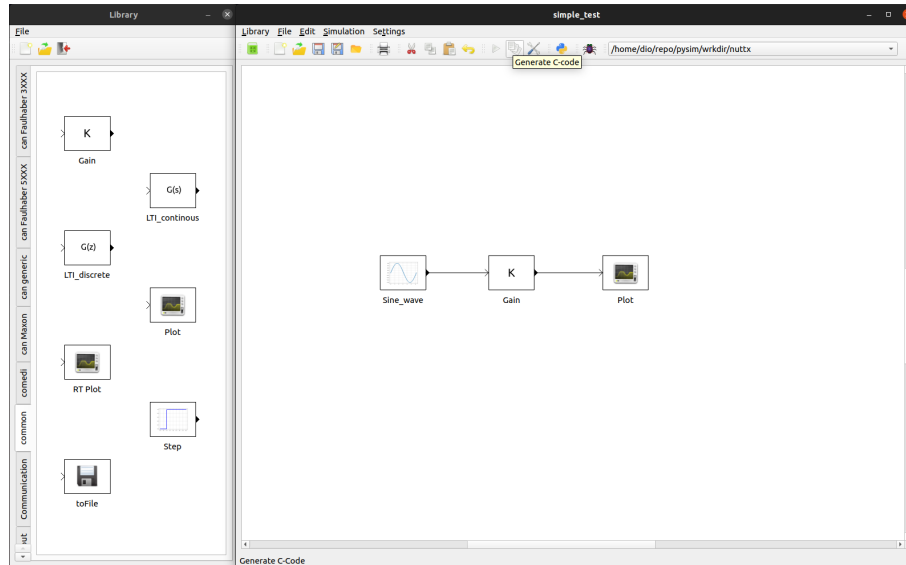


Figure 2.1. Simple preview of PysimCoder’s graphical tool

The application is divided in two windows: the library and the diagram canvas. On the left, we have the library window which contains all the blocks which are supported on pysimCoder. From here the user will be able to drag and drop different types of blocks onto the canvas, depending on their control needs. Then on the right, there is the canvas window where the user will be able to interconnect all the blocks that were added there from the library. On the top of this window there is a toolbar and a menu bar from which the user is able to perform a handful of operations necessary for their design. The most note-worthy buttons used are: **Generate C-code**, **Simulate**, and **Settings**. These operations are essential to the code-generation and simulation functions of the application.

Regardless of the complexity of the diagram, the work flow (set of operations) will always be the same getting from design to code generation. The blocks are dragged to the canvas from the library, then interconnected based on the user’s design. Once the blocks are nicely organized and connected, the user will then have to tune any block parameters that must be changed for their particular system. After verifying the correctness of the diagram, the user will have to choose a target-specific template Makefile which will tell the program which compiler to use for the current project. Finally, the user must save the project, and the **Generate C-code** button will be pressed in order to build a binary executable file. In case of a simple simulation running on the host computer, the **Simulate** button can also be pressed in order to execute the file in the same terminal where the pysimCoder application is running.

Every instruction given above will be related to the next part of this section, where I will explain in simple terms how the source code is reacting to the different

operations we are performing in the graphical tool, and what is the final result of the code generation performed by the application.

The core functionality of the pysimCoder project is enabled by three main languages: Python, C basic, and Makefile. Python is the engine behind the graphical tool, and the back-end processes related the operation of the tool, and also handles the C-code generation. This generated code written in C-basic is constructed by many .c files related to the blocks being used. Makefile is used for organizing the compilation the .c files of the blocks, installing all modules necessary for use of pysimCoder, as well as for producing the final executable file which is the end result of the design made in the GUI. The image below gives a more simplistic view of the code generation path, from GUI to the executable generation.

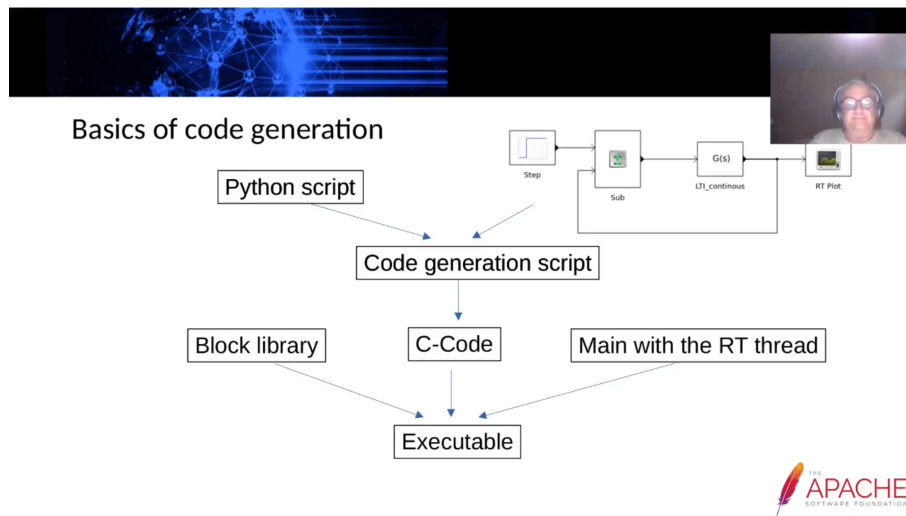


Figure 2.2. Code path from GUI to executable file

Each block from the Library window has three main components which complete its functionality in the code. Firstly, before any code generation takes place, there is the .xblk file which holds all the info of the block which the graphical interface can interpret. These include necessary information such as number of inputs and outputs, assignment of editable parameters along with their default values, name of the block, and also name of the .py file which will be essential to the further processing of the block.

The reference to the .py file specified in the .xblk file is the next step once we want to move towards code generation. This .py file is responsible for calling a python class named RCPblk, whose purpose is to create an object from the current state of the block to be used in the main code connecting it to the other components of the schematic.

Last but not least, the block needs a C-code function that will be used in the main C code generated by Python. This function will contain the actual code necessary for achieving the desired output of the block.

Now I will break down the path of the source code based on the work-flow presented earlier in the GUI. In the initial phase, the design of the schematic is taking place. This includes everything from dragging and dropping blocks onto the editor, making connections between them, setting their parameters and choosing the simulation settings. During the design phase, various Python tools from the path /toolbox/supsisim/src are being utilized to run the graphical tool, and to ultimately save the state of the current project into a file with .dgm extension. This .dgm file will contain everything known about the design before code generation takes place, which mainly includes the

edited `.xblk` of each block, the connections between the blocks, and the simulation settings.

Once the design of the schematic is completed and saved into a `.dgm` file, the code generation can finally take place. In this phase, the same tools from the path `/toolbox/supsisim/src` will be used. When the **Generate C-code** button is pressed in the toolbar, the supsisim toolbox will first create a file called `tmp.py` and store it in the working directory of the project. This `.py` file will be responsible for collecting all the C-functions of each block, and using them to generate the main C-code. Then it will create a folder in the same path, and it will store there the main C-code that was generated, and also a Makefile. The last step of the `tmp.py` file will be to call the Makefile, which will finally compile the generated C-code into a binary executable. This binary file is the end-product which is obtained from the simulation design, and is the only file necessary for the execution of the simulation on the target-hardware.

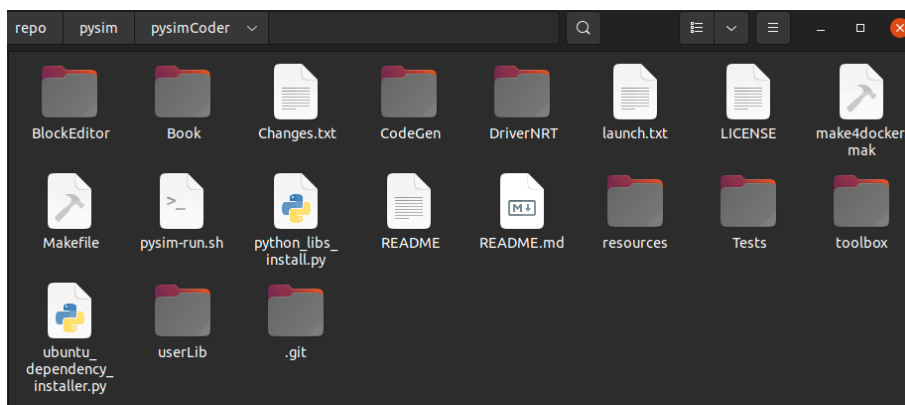


Figure 2.3. Source code organization in main repository of pysimCoder

Above we can see the root filesystem of the `pysimCoder` project. Most of the core work being done by the program is happening at the `toolbox` folder. In the `resources` folder there is all the `.xblk` and `.py` files of each block, as well as all the images used for the blocks' thumbnails. For all code generation purposes of the program, the folder `CodeGen` will be used. Here lies all the C files containing the functions specific to each block (including their dependencies), as well as all the template Makefiles used for the various targets that are supported. These three directories are essential to the `pysimCoder` application, and work closely together during the path from the GUI to the binary executable generation.

Although there are a few parts of the source code I have not explained, there is already enough information in this chapter to be able to clearly understand the different implementations and changes that will be made in the forthcoming chapters.

Chapter 3

Vector support

3.1 Vectors in Control Systems

As remembered from any elementary mathematics or physics class, a **vector** is a directed line segment. Usually a vector can be visualized on a line, a plane, or a space, otherwise known as the three spatial dimensions. In other words, a vector is a set of numbers, which represent a line with a direction in an N-dimensional space. Although vectors can have a finite or infinite number of dimensions, they have already played a significant role in mathematics and physics even with only the three spatial dimensions that we are closely familiar with. The uses of vector mathematics reach wide across many fields of science, such as classical mechanics, astrophysics, electronics, and many others.[3] Included here is also control engineering, which uses vectors for many of its mathematical operations.

In order to design and analyze control systems, quantitative mathematical models of these systems must be obtained. This condition is true independently of the type of dynamic system in question. The mathematical model of the dynamic system is initially expressed as a set of differential equations, which are then linearized for further use with the Laplace transform. The Laplace mathematical tool is then able to obtain a solution of these linearized differential equations, and ultimately describe the operation of the system.[1, chapter 2.1]

Once the mathematical model is simplified with the use of transfer functions (obtained from Laplace), the input-output relationship can be described. This function is especially useful when wanting to represent the system as a block diagram schematic. The blocks are unidirectional and operate based on the transfer function of the variables to be controlled.[1, chapter 2.6]

Nowadays all the graphical tools for control systems are based from this concept, and many of them use vector signals for operation of their blocks. PysimCoder also uses vectors (and matrices) for many of the mathematical operations necessary for the simulation of a control system, however before my implementation covered in the next sections of this chapter, it did not support the use of vector signals as inputs or outputs of a block. This is a feature that would greatly improve the application, since it can simplify the diagram appearance, and it could better utilize many of the mathematical tools needed for controlling a dynamic system.

Simulink already supports vector signals both in the graphical editor and in the C-code generation tool (SimulinkCoder). The **Constant** block has the ability to produce a single number, but also a vector or matrix. This block can be used to generate a pure vector signal which is constant on its output. There is also a **Mux** block which can combine input signals into a vector, and also its inverse **Demux** which extracts each element from the vector signal and outputs them individually. These blocks can be used in combination with other blocks which may have vectors as an input or output of their operation. Simulink only has a handful of **pure** vector blocks, including **Mux** and

Demux. Most of the other blocks support single-data signals, as well as vectors on top of that, if the input is given as such.

Many of the same blocks mentioned above that are used in Simulink can be emulated in `pysimCoder`. Therefore, for the purpose of testing and experimenting with vector signals for the first time in `pysimCoder`, I have used the same concept of these blocks. I started with a `Constant Vector` block, which could create a pure vector signal, together with a `Print Vector` block to print the signal on the terminal. After that I proceeded with creating the `Mux` and `Demux` blocks, since they will be able to convert vector signals into its individual elements, and back. This can create an interface between the vector blocks and the `normal` blocks.

3.2 Stable implementation of vectors in `pysimCoder`

3.3 Vectors with dynamic dimension-setting algorithm



Chapter 4

Support for Zynq based board in pysimCoder




Chapter 5

RISC-V board with NuttX RTOS



Chapter 6

Demonstration of Real Time Control System using PysimCoder



Chapter 7

Conclusion



Appendix A

Source Code



Appendix B

Glossary

MCU ■ Micro Controller Unit



References

- [1] DORF, Richard C.. *Modern Control Systems*. 6 ed. Menlo Park: Addison-Wesley, 1992. ISBN 0-201-60701-8.
- [2] BUCHER, Roberto. *Python for control purposes*. Available from <https://robertobucher.dti.supsi.ch/wp-content/uploads/2017/03/BookPythonForControl.pdf>.
- [3] SHAFAREVICH I., Remizov A.. *Linear Algebra and Geometry*. 1 ed. Berlin: Springer, 2013. ISBN 978-3-642-30993-9.