



**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	
2.1.1 Control systems and their design tools	2	
2.1.2 PysimCoder: Control systems design and code generation tool	2	
2.2 Code generation path of pysimCoder	4	
2.2.1 Graphical User Interface of pysimCoder	4	
2.2.2 Source code of pysimCoder	5	
3 Vector support	8	
3.1 Vectors in Control Systems	8	
3.2 Basic implementation of vectors in pysimCoder	9	
3.2.1 Method of approach	9	
3.2.2 Changes in Python source and generated C-code	10	
3.2.3 New blocks for vector testing	11	
3.2.4 Demonstration of basic vector support (static dimension setting)	12	
3.3 Vectors with dynamic dimension-setting algorithm	12	
3.3.1 Method of approach	12	
3.3.2 Development of Algorithm	12	
3.3.3 Changes in source code	14	
3.3.4 Demonstration of automated dimension setting	15	
4 Extending support of LinuxOS/GNU targets in pysimCoder	16	
4.1 Zynq based board on pysimCoder	16	
4.1.1 Hardware requirements	16	
4.1.2 Existing Simulink project	16	
4.1.3 Translation to pysim-Coder block	16	
4.1.4 Demonstration of results	16	
4.2 RaspberryPi board on pysimCoder	16	
4.2.1 Hardware requirements	16	
4.2.2 Existing Simulink project	16	
4.2.3 Translation to pysim-Coder block	16	
4.2.4 Demonstration of results	16	
5 Testing and extending NuttX RTOS support in pysimCoder	17	
5.1 NuttX RTOS	17	
5.1.1 Introduction	17	
5.1.2 Existing support in pysimCoder	17	
5.2 ESP32C3 board with NuttX	17	
5.2.1 Hardware requirements	17	
5.2.2 Installing NuttX RTOS on board	17	
5.2.3 Testing pysimCoder blocks on board	17	
5.3 Demonstration of control using ESP32C3 with NuttX	17	
5.3.1 Hardware requirements	17	
5.3.2 Translation of RaspberryPi project	17	
5.3.3 Example of real usage	17	
6 Conclusion	18	
A Source Code	19	
B Glossary	20	
References	21	

/ 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
3.1	Vector represented in 3D space ..	8
3.2	Block diagram schematic for control system	9
3.3	Necessary code/files for pysimCoder block	10
3.4	Algorithm for automatic setting of dimensions of vector blocks	13



Chapter 1

Introduction

Chapter 2

PysimCoder

2.1 Introduction to PysimCoder tool

2.1.1 Control systems and their design tools

“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.

2.1.2 PysimCoder: Control systems design and code generation tool

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 graphi-

cal 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.

2.2.1 Graphical User Interface of pysimCoder

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.

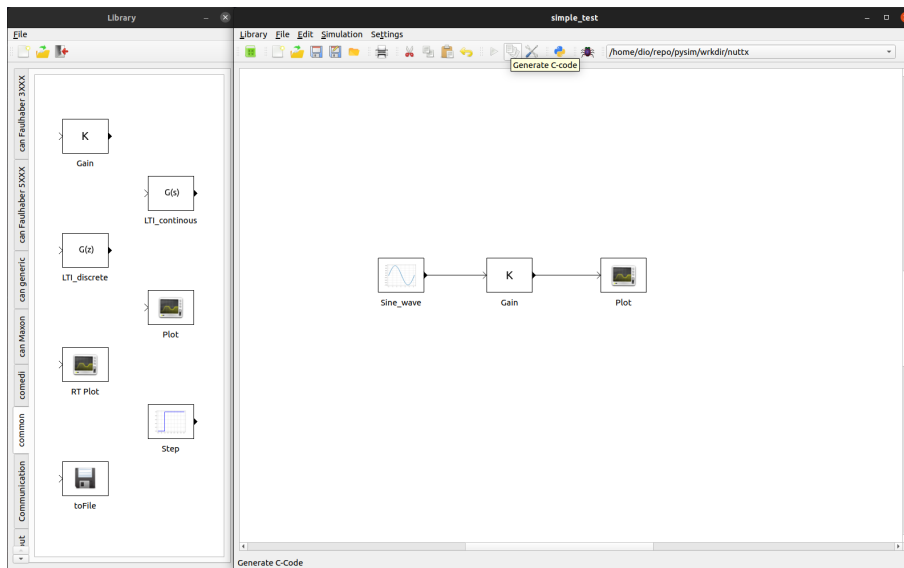


Figure 2.1. Simple preview of PysimCoder's graphical tool

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.

2.2.2 Source code of pysimCoder

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.

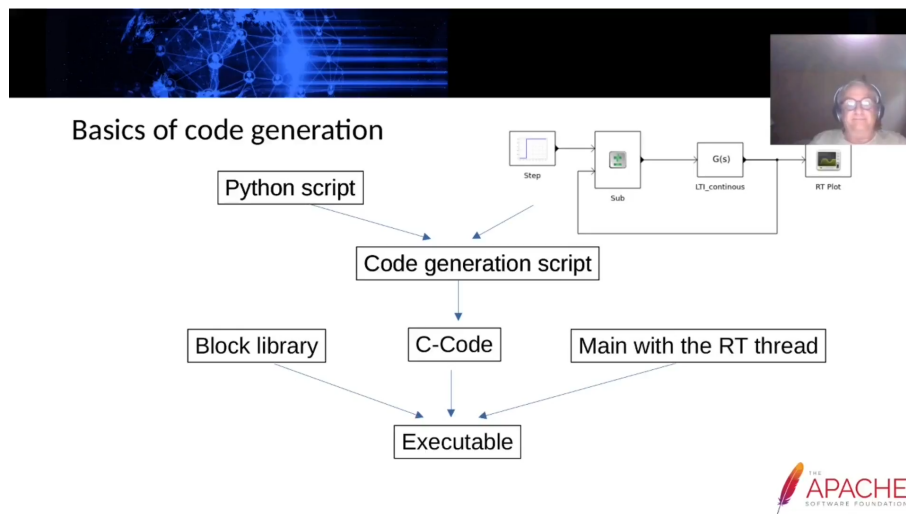


Figure 2.2. Code path from GUI to executable file

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.

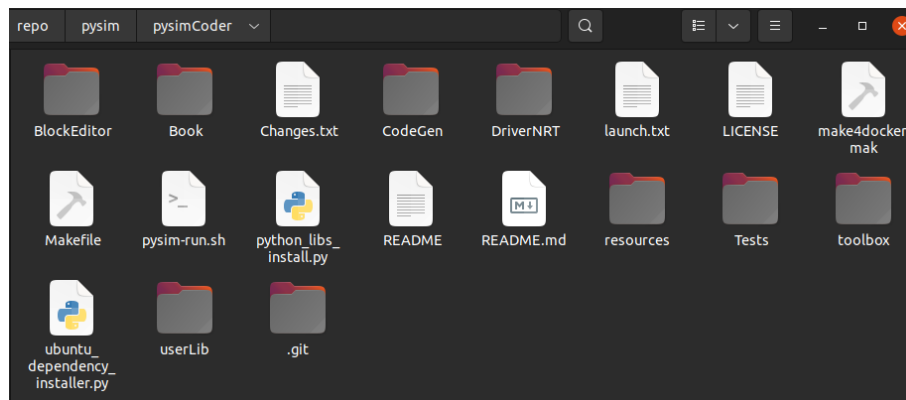


Figure 2.3. Source code organization in main repository of pysimCoder

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 (see Fig. 3.1). 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.

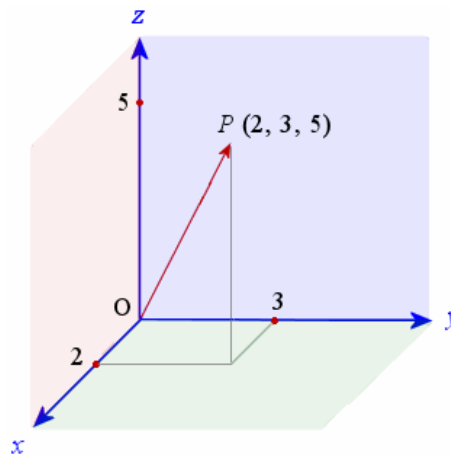


Figure 3.1. Visualization of a vector in a 3-dimensional space (x,y,z axes). The point P represents the direction of the vector, and contains the magnitude as well.

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 (see Fig. 3.2). The blocks are unidirectional and operate based on the transfer function of the variables to be controlled.[1, chapter 2.6]

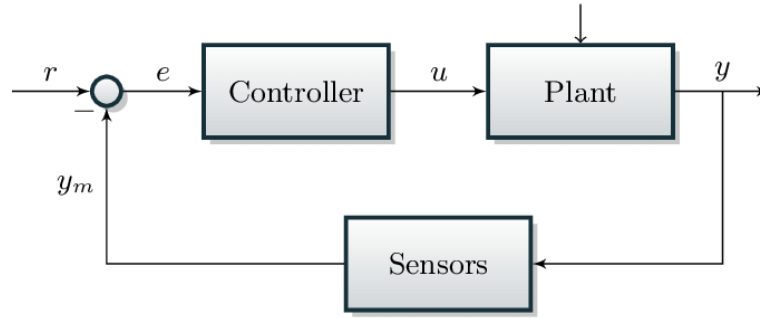


Figure 3.2. Example of usage of block diagram schematics for representation of a closed-loop feedback control system.

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. The approach to this will be covered in the next sections of this chapter.

3.2 Basic implementation of vectors in pysimCoder

3.2.1 Method of approach

Although pysimCoder did not have vector support before this implementation, it did have a framework for enabling the dimensions of a block. By creating new pysimCoder blocks which utilize this ability to carry input and output dimensions, I will be able to test the processing of vectors in the C-function level. In order to achieve this I will have to make changes in the source of the code generation tool, as well as

adding blocks in a new **Vectors** library. For initial simplicity, the dimensions will be set as a block parameter from the user. The automatic setting of dimensions will be covered later, in the next section of this chapter.

3.2.2 Changes in Python source and generated C-code

Before trying to create the new blocks for testing vector signals, first the source of the `pysimCoder` project must have the ability to enable vectors for these blocks. By making some minor changes to the block representation object, as well as the code generation process, the new blocks will then be able to read their own dimensions in order to properly process the input data.

To better understand where to make these changes, I will refer to the main elements of a `pysimCoder` block from chapter 2: the `.xblk` file, `.py` file, and `.c` function. In fact, these elements follow a strict hierarchy of representation of a block, as seen in the image below.

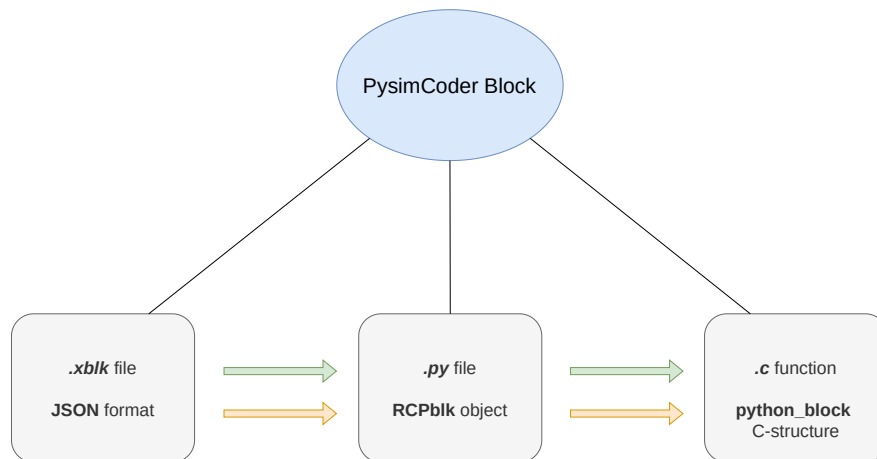


Figure 3.3. The supporting files for a `pysimCoder` block are shown. The green arrows represent which file contains the specification of the next file or function to be called. The orange arrow shows the switching of block representations. The general direction of both arrows show the hierarchy of the code dependencies of the block.

On the GUI level, the block is initially represented in the same format as the `.xblk` file, otherwise known as the JSON format (although there is no JavaScript involved). The corresponding Python function specific to the block is then called with all the information contained in the JSON, and will ultimately create a Python class called `RCPblk`. This class is the next level of representation, and it will be used while `pysimCoder` is generating the C-code. When Python is generating the code, it will use each `RCPblk` object to initialize the block on the C-code level. To do this it will pass all of the block information to a C-structure called `python_block`, which will enable all the written C-functions of the block to utilize its own properties for correct functioning of the code. This C-structure is the last level of representation, and is the only format needed for running of the code on the target. In other words, the other formats only serve to generate this final one.

So all these representations seem to be carrying the block information, and the last two already have some form of support for dimensions of input/output signals. They are seen in the image below, where both of them have the variables already made to carry dimension data. For `python_block` there is nothing to change, however in the `RCPblk` object, all of the dimension information seems to be defaulted to 1 for

all inputs and outputs. This is the first change to make, so that when the RCPBlk is called from the block's Python code, the dimension will pass from the user's input to the `python_block`. Otherwise it would just default to 1, making vectors impossible to use here.

After that is settled, the block can then take a dimension as an input parameter from the user, and pass it on to the other representations. This is key to the whole feature, since now I will be able to show how the source code can use this information in order to create actual memory space for the signal.

In pysimCoder, the main file which deals with code generation is `RCPgen.py`. In that file is where all the blocks of the simulation are gathered and interconnected, and then generated as C-code. The tool uses the `RCPblk` objects of each block in order to propagate the information to the `python_block` structure. The blocks are interconnected using `nodes`. A `node` in terms of pysimCoder is a connection between an output of a block, and the inputs of one or many other blocks. The node will carry the values from one block to another, however the source only generates one value per node. This is another barrier preventing the program from enabling vector support, and it is the next change to be made to prepare the blocks for using vector signals.

■ 3.2.3 New blocks for vector testing

After the changes made to the source as explained above, the new vector blocks will finally be able to pass vector signals and use their dimensions for the correct functioning of the C-code. The first vector block to create will be the **Constant** block, as seen in Simulink. This will be able to create a vector output signal based on the input array given from the user. Then to test if the vector signal is actually passing correctly, there is a need for a **Print** block which will be specific to printing vector signals as an array on the main terminal of pysimCoder. These two blocks will become the first experiment on vector signals.

The next blocks to be created will be related to mathematical operations on vectors. The **Gain** block takes each element of the input vector and multiplies it by the factor given by the user. In fact the **Gain** block already existed in pysimCoder, however it has never been used with vectors as its input. Then two very simple arithmetic operations will be implemented for vectors: summing and subtracting. Describing what they do is somewhat redundant, however it's worth noting that the input vectors of both blocks must be of equal dimension for correct operation. By refining these blocks, I will be able to prove that vectors can be passed, and processed in a mathematical way.

Finally there are two very important blocks which have a special role with vectors in a block diagram schematic. These are the **Multiplexer** and **Demultiplexer** blocks, otherwise known as **Mux** and **Demux**. **Mux** is responsible for generating a vector from many single inputs, while **Demux** breaks down the vector into many single outputs. This functionality will be especially useful in the future, when **normal** blocks want to interact with **vector-able** blocks.

These are all of the vector blocks that have been crafted for the purpose of testing vectors in pysimCoder. It is worth mentioning that some blocks are able to know their dimension even without any user input. If the dimensions can be found using some mathematical relation, then it can be passed automatically from the Python file which calls `RCPBlk`. Otherwise, the dimension will have to be given explicitly from the input parameters of the block, therefore it will pass first from the `.xblk` representation. In the next section I will explain my extended implementation of vectors, where all the dimensions are set automatically by an algorithm, and no user input is needed.

■ 3.2.4 Demonstration of basic vector support (static dimension setting)

■ 3.3 Vectors with dynamic dimension-setting algorithm

■ 3.3.1 Method of approach

In the initial functionality made for vector support, all the dimensions were either set manually, or set automatically from their internal properties. In this section I will cover the automatic setting of dimensions for all blocks which cannot know their dimensions by their own.

To do this, I will create an algorithm which will find each of these blocks, and set their dimensions based on the output of the preceding block. The detection and setting of dimensions must happen during the code generation process. By implementing this algorithm, the dimensions will not be passed as integer parameters anymore, but will use the appropriate variables for dimensions of input and output seen in the `RCPBlk` object class, and the `python_block` C structure.

It is worth to be noted that there are many approaches to setting dimensions automatically in a block diagram schematic which supports vectors. However, for the purpose of showcasing an experimental version which works with the previously working setup of the vector feature, I have chosen the above methodology. In the future I hope that pysimCoder developers including myself will use this proof as a motivation for a more general and organic solution for this feature.

■ 3.3.2 Development of Algorithm

As seen in the last section, some vector blocks seem to know their dimensions internally, such as `Constant`, `Mux`, and `Demux`. The first two of these also seem to be generating a vector output. For the purpose of the explanation of this dimension setting algorithm, I will denote these two blocks as `Source blocks`. It is important for this implementation that the `source blocks` always know their dimensions internally, since their dimensions will be passed on down the line.

Other blocks, which previously needed their dimensions to be specified, will then be split into two groups. The `middle blocks` are vector blocks with both input(s) and output(s), while the `destination blocks` are those with only vector input(s). Since the latter only have input signals, then they can simply inherit the dimension from the output of the preceding block. However for the `middle blocks`, there is also the setting of the output signal dimensions, which will need further processing.

To find a way to set the output dimensions of a `middle block`, each input/output relationship must first be investigated. In the best case scenario, the output signal would inherit the dimension from the corresponding input signal, assuming a 1:1 mapping of dimensions. In this case, all the middle vector blocks which were previously created (see section 2 of this chapter) have a 1:1 ratio of input/output dimensions, which will make the processing much easier. Nevertheless the chance of having a different ratio when working with vectors in the future is quite high, therefore I have ensured to leave a framework for making blocks which have a more complicated I/O relationship.

To describe the I/O relationship of a block on a mathematical level, I will use the notion of the `dimension ratio`. In other words, this is the ratio between the

input dimension and the output dimension. Just earlier it was mentioned that the current `middle` vector blocks all have a 1:1 ratio, which means that the value of the input dimension will be equal to the value of the output dimension. In other cases, the output dimension would have to be multiplied by some factor in order to achieve a natural number. This is the exact reason why I have decided to make a new parameter in the `RCPblk` class by the name of `dimRatio`. This parameter will hold the information of the I/O dimension ratio, ultimately guiding my new algorithm on how to set its output dimensions.

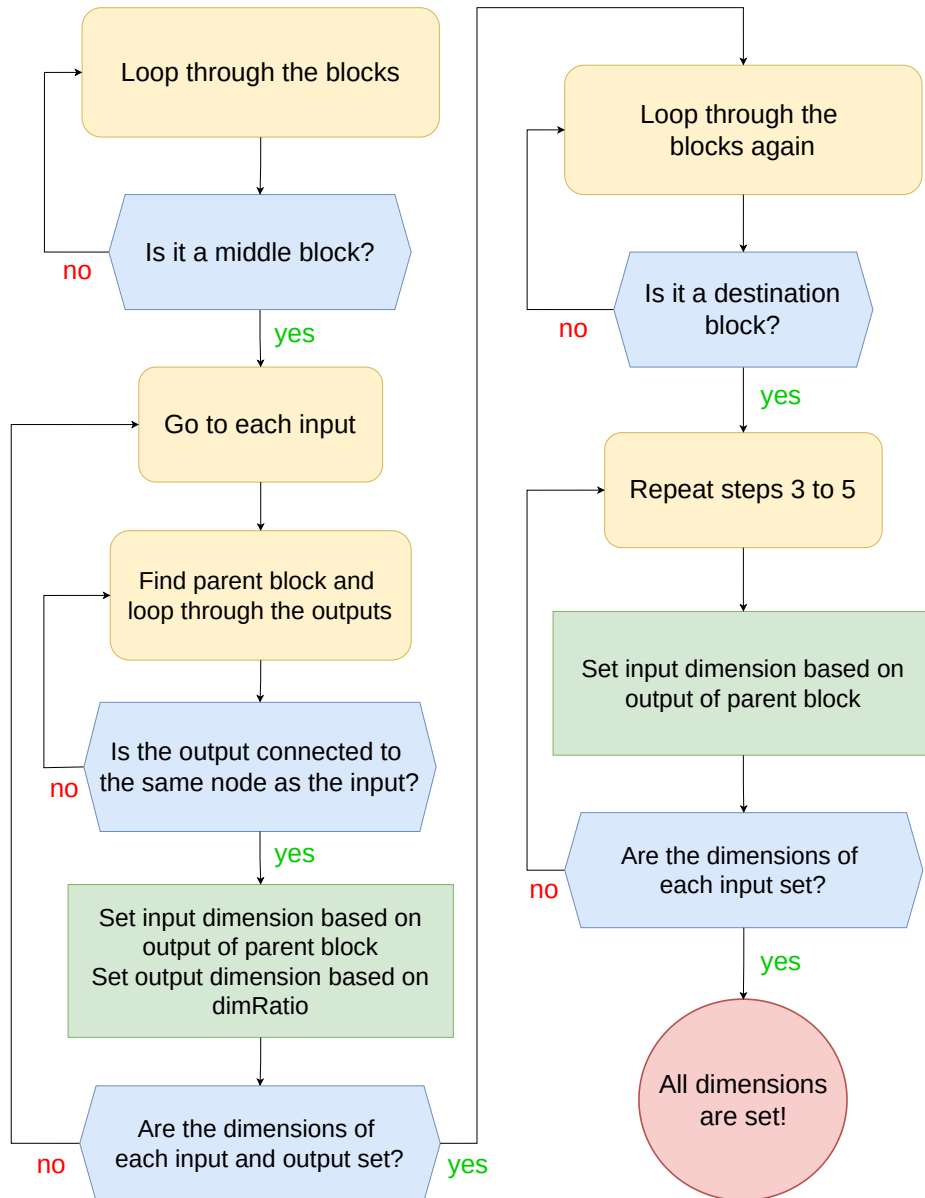


Figure 3.4. Flowchart explaining the path to setting dimensions automatically for vector blocks which previously needed input. Chart is divided in two parts: the left shows the setting of middle block dimensions, while the right is for the destination blocks. The first element of the flowchart is on the top left.

Now I will proceed to explain the inner workings of the dimension setting algorithm for vector blocks, using the flowchart given above (see Fig. 3.4). The first part of the algorithm will be dealing with the middle blocks, as seen in the left half of the diagram. Once that part is resolved, the algorithm can then continue to process the destination blocks. This has to happen in this order since the destination blocks usually get their dimensions from a middle block.

Initially the middle blocks must be found and accessed (Steps 1 and 2). Then the program must iterate through all the inputs of that block, and proceed to find the **parent** block which contains an output connected to the current input selected (Steps 3,4,5). This input will then inherit its dimension from the corresponding output of the parent block. The output signals of the middle block on the other hand, will be based on the dimension ratio parameter contained in the block (Step 6). If all the inputs have been dealt with, the algorithm will then move to the next phase, involving the destination blocks.

For the destination blocks, the same idea will be applied as for the middle blocks. The main difference here, will be that for destination blocks there are only inputs. This means there is no need for any dimension ratio to help select an output dimension, since there are no outputs. Therefore the inputs of the block will simply inherit their dimension from the parent block's output, and the algorithm will be done (Steps 8-13). Most of the steps are identical as for middle blocks, except for Step 9 and 11.

Although the flowchart seems a bit complex, the idea behind the algorithm is not so hard to understand. The goal is to propagate all the vector dimensions, from a **source block**, through the **middle blocks**, and onto the **destination blocks**. To do this the algorithm simply goes to each middle block, sets the dimensions based on the block preceding it, and then continues to do the same for all the destination blocks. Also, it is very important that there is a source block preceding all the other ones, and it must have static dimensions which were set internally.

■ 3.3.3 Changes in source code

After understanding the core logistics of the dimension setting algorithm, a decision must be made on where to apply the algorithm, and what other changes must be made. The main changes in the source code will be related to the code generation process, and also to the block definitions of the vector-enabled blocks. Before I explain my choice, it is worth noting that there are many places in the source where the algorithm can be implemented. The only condition is that it is done before the variables of the dimensions are initialized in the code generation process.

Most of the C-code writing that `pysimCoder` does is occurring in `RCPgen.py`, from the `supsisim` toolbox (see chapter 2.2.2). Therefore this file in a way would be the first place to put the algorithm, right before the variables of the dimensions and the nodes are initialized in the C-code. Although this placement works and is more direct in terms of implementation, other methods should be considered for the future. For example it can also be written as a function, and imported externally for this feature, in order to shorten the code in `RCPgen.py`. Another consideration would be to apply the algorithm on the GUI level. This could be handy for seeing all the dimensions before the code is even generated. Nevertheless, the first option has been chosen as a testing ground for the more advanced versions to come.

In order for the algorithm to function, it needs to be able to interact with a parameter used for the dimension ratio, which describes the I/O relationship of the vector block. This parameter will be called `dimRatio`, and it will be placed in the

`RCPblk` class. The reason for this placement in the block representation hierarchy, is the fact that the algorithm itself is placed in the `RCPgen.py` file, as mentioned above. The `dimRatio` will be a list with two elements, representing the ratio. The first element is the numerator, and the last is the denominator. In future implementations, the algorithm must be able to use this parameter for all types of ratios that could occur. However, for the current blocks which are contained in the `Vectors` library, there are only 1:1 ratios used (as seen in last subsection). For the **destination** blocks, who don't have any outputs at all, I will use the 0:0 ratio, so that the algorithm can know how to process it.

For the blocks to be able to adapt to these changes in `RCPblk` and `RCPgen`, all of the block file dependencies must be changed. First the input parameter must be removed from the `.xblk` definition, so that the user will not be able to input any dimension manually. Then the `dimRatio` variable must be enabled in the `.py` file of the block, and set to the corresponding ratio involved. Then the C function will also be changed, so that all the functions of the vector blocks will be using the `dimIn` and `dimOut` variables defined in the `python_block` structure. Previously the blocks were using the `intPar` to pass the dimensions in the C-code level, however after the next changes that will be mentioned below, all the blocks will be able to read their dimensions from the newly enabled variables.

I will finish this subsection by mentioning the other changes that were made in `RCPgen.py`. After the algorithm achieves its goal, the variables in the main C file which will hold all the dimension values will then be initialized. The nodes variables will also be initialized with the correct length of array (as seen in section 2 of this chapter), however this time it will simply read the dimension from the node output and directly use that value. In the first implementation of vectors, these values were buffered through a Python list. Last but not least, the `python_block` structures given for each block in the diagram will then take the values from the aforementioned dimension variables, and pass them onto the structure's internal dimension variables (`dimIn/dimOut`).

■ 3.3.4 Demonstration of automated dimension setting

Chapter 4

Extending support of LinuxOS/GNU targets in pysimCoder

4.1 Zynq based board on pysimCoder

- 4.1.1 Hardware requirements**
- 4.1.2 Existing Simulink project**
- 4.1.3 Translation to pysimCoder block**
- 4.1.4 Demonstration of results**

4.2 RaspberryPi board on pysimCoder

- 4.2.1 Hardware requirements**
- 4.2.2 Existing Simulink project**
- 4.2.3 Translation to pysimCoder block**
- 4.2.4 Demonstration of results**

Chapter 5

Testing and extending NuttX RTOS support in pysimCoder

5.1 NuttX RTOS

5.1.1 Introduction

5.1.2 Existing support in pysimCoder

5.2 ESP32C3 board with NuttX

5.2.1 Hardware requirements

5.2.2 Installing NuttX RTOS on board


5.2.3 Testing pysimCoder blocks on board

5.3 Demonstration of control using ESP32C3 with NuttX

5.3.1 Hardware requirements

5.3.2 Translation of RaspberryPi project

5.3.3 Example of real usage



Chapter 6

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.