ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEVELOPMENT OF A DOMAIN-SPECIFIC LANGUAGE FOR DESIGNING AND ANALYSING CONTROL SYSTEMS

SENIOR DESIGN PROJECT

Celaleddin HİDAYETOĞLU

Control and Automation Engineering

Thesis Advisor: Prof. Dr. Leyla GÖREN

JUNE 2019

ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEVELOPMENT OF A DOMAIN-SPECIFIC LANGUAGE FOR DESIGNING AND ANALYSING CONTROL SYSTEMS

SENIOR DESIGN PROJECT

Celaleddin HİDAYETOĞLU
(040130332)

Control and Automation Engineering

Thesis Advisor: Prof. Dr. Leyla GÖREN

JUNE 2019

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK ELEKTRONİK FAKÜLTESİ

KONTROL SİSTEMLERİ TASARLAMAK VE ANALİZ ETMEK İÇİN BİR ALANA ÖZGÜ DİL GELİŞTİRİLMESİ

BİTİRME TASARIM PROJESİ

Celaleddin HİDAYETOĞLU
(040130332)

Kontrol ve Otomasyon Mühendisliği

Tez Danışmanı: Prof. Dr. Leyla GÖREN

HAZİRAN 2019

To flowers and birds

FOREWORD

I thank my family and my friends for their love, support and encouraging attitude.

I thank my thesis advisor Professor Leyla Gören for her inspiring lectures, constructive feedback and positive communication throughout the project.

I thank Alan Kay, Peter Norvig, Richard P. Gabriel, Paul Graham, Douglas N. Adams, Peter Seibel, Panicz Godek, and numerous other people for sharing their ideas related to computation and programming languages. Their ideas influenced my perspective on these topics very much, even though I've never met them personally.

June 2019 Celaleddin HİDAYETOĞLU

TABLE OF CONTENTS

Page

FOREWORD ..... **vii**
TABLE OF CONTENTS ..... **ix**
ABBREVIATIONS ..... **xi**
LIST OF TABLES ..... **xiii**
LIST OF FIGURES ..... **xv**
SUMMARY ..... **xvii**
ÖZET ..... **xix**
1. INTRODUCTION **1**
1.1 Organization of the Document 1
1.2 The Idea of Lisp 2
2. THE LANGUAGE **5**
2.1 Usage 5
2.1.1 Naming values 5
2.1.2 Expressing transfer functions 6
2.1.3 Substituting free symbols of transfer functions with values 8
2.1.4 Expressing input signals 8
2.1.5 Evaluating the response of a system 9
2.1.6 Plotting the response of a system 10
2.1.7 Other plotting operations 12
2.1.7.1 Pole-zero plot 12
2.1.7.2 Bode plot 13
2.1.7.3 Nyquist plot 13
2.1.7.4 Root locus plot 14
2.1.8 Expressing block diagrams 15
3. IMPLEMENTATION DETAILS **17**
3.1 Choice of implementation language 17
3.2 Operators of the control systems DSL 17
4. RESULTS AND DISCUSSION **21**
4.1 Results 21
4.2 Future possibilities 21
REFERENCES **23**
CURRICULUM VITAE **25**

ABBREVIATIONS

DSL : Domain-specific language

LIST OF TABLES

Page

Table 2.1 : Input signal expressions and their optional parameters 9

LIST OF FIGURES

Page

Figure 2.1 : Step response of a system and unit step function plotted using `plot-together` 11
Figure 2.2 : Step, ramp and sine input responses of a system plotted using `plot-separately` 11
Figure 2.3 : Step, ramp and sine input responses of a system and their respective input functions plotted using `plot-separately` and `together` 12
Figure 2.4 : `pole-zero-plot` of a system 13
Figure 2.5 : `bode-plot` of a system 13
Figure 2.6 : `nyquist-plot` of a system 14
Figure 2.7 : `root-locus-plot` of a system 14
Figure 2.8 : A block diagram representing a control system 15

DEVELOPMENT OF A DOMAIN-SPECIFIC LANGUAGE FOR DESIGNING AND ANALYSING CONTROL SYSTEMS

SUMMARY

The aim of the project is to develop a language closer to the domain of control systems than general-purpose programming languages. Such a language may enable a person to express the notions of control systems in a more compatible way with the mental model of a control systems engineer. As a result, people working with control systems may communicate their intentions more clearly with each other through programs. Additionally, they may understand computer programs written with such a language more easily and modify them in a more comfortable and precise way.

A domain-specific language is a computer programming language with a limited expressiveness focused on a particular domain. A language, which the ideas expressed using it can be executed on a computer, with specialized expressions and a focus on control systems would be called a control systems domain-specific language.

Development of the control systems DSL consists of three parts. Firstly, the essential elements of control systems development is determined by analyzing a prominent book of the field. Secondly, syntactic rules are designed to express these elements in a clear way. Lastly, these rules are implemented to form a language which its expressions are executable on the computer.

The control systems DSL is implemented using the programming language Hy, which is a dialect of Lisp embedded in Python. The packages python-control, SymPy and matplotlib from the Python ecosystem are used for control systems related operations, symbolic mathematics and data visualization respectively. Lisp macros are employed to build syntactic abstractions over the underlying subsystems developed using these packages. New expressions and operators tailored to control systems development are defined using syntactic transformations powered by macros.

The project is developed as an open-source project to make it possible for people interested in to read or modify the source code of the project and contribute to it. Furthermore, the project is registered on the Python package index, so it is easy for users to get the language and start using it.

KONTROL SİSTEMLERİ TASARLAMAK VE ANALİZ ETMEK İÇİN BİR ALANA ÖZGÜ DİL GELİŞTİRİLMESİ

ÖZET

Projenin amacı, kontrol sistemleri alanına genel amaçlı programlama dillerinden daha yakın bir dil geliştirmektir. Böyle bir dil sayesinde dili kullanan kişinin, kontrol sistemleri ile ilgili fikirlerini bir kontrol mühendisinin zihinsel modeline daha uygun bir şekilde ifade etmesi sağlanabilir. Bu durum kontrol sistemleri üzerinde çalışan insanların maksatlarını programlar üzerinden birbirlerine daha net şekilde anlatabilmelerini sağlar. Ek olarak, alanda çalışan insanların böyle bir dil kullanılarak yazılmış programları daha kolay anlayabilmesi ve bu programlar üzerinde daha rahat ve hatasız bir şekilde değişiklikler yapabilmesi mümkündür.

Bir alana özgü dil, yalnızca belirli bir alana odaklanarak o alan ile ilgili fikirlerin ifade edilmesini sağlayan bir programlama dilidir. Bir kontrol sistemleri alana özgü dili ise kontrol sistemleri odağında özelleşmiş ifadeler barındıran ve bu dille ifade edilen fikirlerin bilgisayarlar tarafından çalıştırılabilir olduğu bir dildir.

Kontrol sistemleri alana özgü dilinin geliştirilmesi üç aşamada gerçekleştirildi. İlk aşama olarak alanın önde gelen bir kitabı incelenerek kontrol sistemleri geliştirme sürecinin temel unsurları belirlendi. İkinci aşamada bu unsurların net bir şekilde ifade edilebileceği sözdizimsel kurallar oluşturuldu. Üçüncü ve son aşama olarak bu kurallar, ifadeleri bir bilgisayar üzerinde çalıştırılabilecek bir dili oluşturmak üzere gerçeklendiler.

Kontrol sistemleri alana özgü dili, Python programlama dilinin içine gömülü olarak geliştirilmiş bir Lisp lehçesi olan Hy programlama dili kullanılarak gerçeklendi. Python ekosisteminden python-control, SymPy ve matplotlib paketleri sırasıyla kontrol sistemleri ile ilgili işlemler, sembolik matematik ve veri görselleştirme süreçlerinin gerçekleştirilmesi için kullanıldılar. Bu paketlerden faydalanılarak geliştirilen alt sistemlerin üzerine bir sözdizimsel soyutlama olarak değerlendirilebilecek alana özgü dilin geliştirilmesi için Lisp makroları kullanıldı. Kontrol sistemleri geliştirmek için özel olarak tasarlanan ifade ve operatörler, makrolar ile ifade edilen sözdizimsel dönüşümler kullanılarak tanımlandı.

İlgilenen herkesin proje kaynak kodunu inceleyebilmesi, düzenleyebilmesi ve projeye katkı verebilmesi için proje bir açık-kaynak proje olarak geliştirildi. Ayrıca alana özgü dil çalıştırılabilir hale getirilerek Python paket dizinine eklendi, böylece dilin kullanıcılarının dile ulaşmasının ve kullanmaya başlamasının kolay olması sağlandı.

1. INTRODUCTION

The objective of the project is to develop an expressive programming language for the domain of control systems. Being a programming language, the ideas expressed using it must be executable on a computer. But beyond that, being a language, it must allow its users to clearly express their thoughts related to the domain. The users of this language would be the people working or studying in the field of control systems.

A domain-specific language is a computer programming language with a limited expressiveness focused on a particular domain [1]. Such a language described in the first paragraph can be called a control systems domain-specific language. A control systems DSL provides expressions tailored to the domain in a way that people in the field would put their intentions on programs more compatibly with their mental model. Likewise, they can understand programs written using the DSL more effortlessly and modify them in a more precise way.

The development of the control systems DSL is realized in three steps. First of all, to identify the essential concepts of control systems development, the book *Modern Control Engineering* by Katsuhiko Ogata is analyzed. Usage patterns and frequencies of the concepts are taken into consideration. Secondly, based on the usage patterns, syntactic rules are proposed for expressing the identified concepts. While proposing the rules, constraints of the implementation tools are considered. Lastly, the syntactic rules found to be the clearest are implemented to form the control systems DSL.

1.1 Organization of the Document

This document is organized into four chapters.

First and current chapter is "Introduction" where the rationale of the project, methods used during development and basic ideas for providing a basis for the following chapters are explained.

Second chapter, "The Language", presents the control systems domain-specific language developed within the scope of the project. In this chapter, usage of the language is illustrated with examples.

In the third chapter, "Implementation Details", the internals of the control systems DSL are explained in outline. Since the source code of the language is public [2], one can read this chapter along with the source code to better understand the implementation.

In the last chapter, "Results and Discussion", the results of the project are shared and future possibilities are discussed.

1.2 The Idea of Lisp

Since the control systems DSL is implemented using and inside a dialect of Lisp, it is sensible to present the basic ideas around it. Because programs written in Lisp may look strange to someone unfamiliar with Lisp.

The core building blocks of Lisp are lists. Actually, Lisp is the abbreviation of *list processing* [3]. Lists have an important role in programs written using Lisp, but the source code of a Lisp program also consists of lists.

The syntax of Lisp is very simple. The expressions meant to be evaluated are called *forms* [4]. There are three kinds of forms: symbols, self evaluating objects and compound forms. Symbols are objects to be used for naming other objects. Self evaluating objects are the objects that have values naturally, like numbers `5` and `10` or string objects such as `"lisp"`. Compound forms are non-empty lists of other forms.

For instance, the expression `(+ 1 2 3)` is an addition form. The symbol + names the addition function. The rest are numbers which are self evaluating objects. Compound forms (which are lists) share the general structure of `(operator argument-1 argument-2 ...)`, meaning the first element of a compound form is an operator and the rest are the arguments to be passed to the operator.

Some convenient consequences arise from having the source code as lists. Since Lisp is good at manipulating lists, and its source code consists of lists, it can manipulate its source code. This makes it possible for Lisp to define operators called *macros* which can take code as their arguments. Macros are functions that operate on source code and they can write code using the code supplied to them. This functionality is called

syntactic transformation [5] and it can be used to define domain-specific languages [1].

2. THE LANGUAGE

The control systems DSL is written using the programming language Hy. But more than that, the DSL is written inside Hy. Thus, the programs expressed using the control systems DSL are actually Hy programs. Thanks to Lisp macros, there are no extra layers between the DSL created for control systems and the core language of Hy. Therefore, Hy code can be used in conjunction with the control systems DSL.

2.1 Usage

This section describes the expressions of the control systems domain-specific language and their usage from the functionality perspective.

In the program samples, the character group "`=> `" is the prompt of the language, which indicates that the computer is waiting for user input. This prompt is also the one users of the language see when they installed the language on their computers.

2.1.1 Naming values

A name can be given to a value using `define`. This enables the programmer to refer to this value using the given name further in the program.

In the below expression, the value of `42` is named with the symbol `answer`.

```
=> (define answer 42)
```

Using a name for a value is beneficial because of various reasons. In a simple sense, a name describes what a value is. A plain `5` in a fragment of program may not make sense to the programmer, but the meaning of a name `resistance`, with the value of `5`, is obvious.

Furthermore, naming provides a way to store the result of a calculation and use it without calculating it again. As a result, there will be one source of information for a

specific value and refactoring the program would be easier than using a nameless value repeatedly more than one places.

```
=> (define the-result (expensive-operation)
     "The result of an expensive operation")
=> (* the-result
      (+ the-result 5))
```

`define` also accepts an optional documentation string for the value being named as the last element of its expression. It can be accessed using the `documentation` operator. This is especially convenient while interacting with the program.

```
=> (define answer 42 "The number forty-two")
=> (documentation answer)
The number forty-two
```

2.1.2 Expressing transfer functions

There are two ways to express a transfer function: using the `define-transfer-function` operator and the `#tf` shorthand expression.

`define-transfer-function` allows the programmer to define a symbolic transfer function with a name. Symbolic transfer function means that the transfer function may contain free symbols without a value. The free symbols will be kept as is and it is possible to give them values when required.

Below expression defines a transfer function named `second-order-system`. The documentation string after the name is optional. The arguments which their first elements are `numerator` and `denominator` are required, and they represent the numerator and denominator of the transfer function, naturally.

```
=> (define-transfer-function second-order-system
     "A transfer function representing a second-order
      system"
     (numerator ωn^2)
     (denominator s^2 + 2*ζ*ωn*s + ωn^2))

          ωn^2
  ---------------------
  s^2 + 2*s*ζ*ωn + ωn^2
```

Numerator and denominator of a transfer function can be accessed using `numerator` and `denominator` operators respectively. `documentation` operator returns the documentation string as expected.

```
=> (numerator second-order-system)
ωn^2
=> (denominator second-order-system)
s^2 + 2⋆s⋆ζ⋆ωn + ωn^2
=> (documentation second-order-system)
A transfer function representing a second-order system
```

Another optional argument accepted by `define-transfer-function` is in the form of `(sampling-period dt)` where `dt` is the sampling period of a discrete transfer function. Sampling period of a discrete transfer function can be accessed using the `sampling-period` operator. Below is a discrete transfer function definition.

```
=> (define-transfer-function a-system
     "A discrete transfer function"
     (numerator k)
     (denominator 2⋆z + a)
     (sampling-period 0.1))

     k
  -------
  a + 2⋆z

  dt = 0.1

=> (sampling-period a-system)
0.1
```

The shorthand expression `#tf` provides a way to express simple and non-symbolic transfer functions. `#tf` is particularly useful for one-off uses where the transfer function conceptually does not need a name. For instance, a constant gain in a block diagram can be expressed as `#tf(5)`.

Actually, it is possible to name a transfer function expressed with `#tf` using `define`.

```
=> #tf(1/s)

  1
  -
  s

=> (define integrator #tf(1/s))
```

2.1.3 Substituting free symbols of transfer functions with values

The transfer function `second-order-system` defined in the previous subchapter contains two free symbols: ζ and ωn. Numerical values can be given to these free symbols using the `substitute` operator. `substitute` expressions are in the form of `(substitute transfer-function symbol-value-list)`.

```
=> (substitute second-order-system (ζ 0.8
                                    ωn 1))

            1
  ---------------------
  1.0*s^2 + 1.6*s + 1.0
```

If a variable is `define`d using a symbol which also exists as a free symbol in transfer function, `substitute` uses that definition to evaluate the transfer function.

```
=> (define ζ 0.8)
=> (substitute second-order-system (ωn 1))

            1
  ---------------------
  1.0*s^2 + 1.6*s + 1.0
```

The values provided to the `substitute` directly (using the `symbol-value-list` argument) have higher priority than the variables defined globally.

2.1.4 Expressing input signals

Input signals are useful for system response related operations and plotting. A system response operator may need an input signal as its parameter to evaluate the system's

Table 2.1 : Input signal expressions and their optional parameters

Input signal	Optional parameters
`#step`	`(start-time end-time amplitude)`
`#ramp`	`(start-time end-time slope)`
`#parabola`	`(start-time end-time)`
`#sine`	`(start-time end-time)`

response under that input signal. For the plotting case, while comparing the input signal and the system output, it is convenient to see both of these in a visual way.

There are four kinds of predefined input signals. These are `#step()`, `#ramp()`, `#parabola()` and `#sine()`. All these input signals are self-explanatory with their names and use the shorthand expression form.

They take the optional parameters described in Table 2.1 to enable their behaviors to be altered. A step input rising after one second and ending after five seconds with an amplitude of ten units can be expressed as `#step(1 5 10)`.

2.1.5 Evaluating the response of a system

To evaluate the response of a system represented by a transfer function, the transfer function must not contain any free symbols. The name `substituted-system` is used to refer to the transfer function which its free symbols are substituted with appropriate values.

```
=> (define substituted-system
           (substitute second-order-system (ωn 8
                                            ζ 0.8)))
=> substituted-system

             64
  ----------------------
  1.0*s^2 + 12.8*s + 64.0
```

There are two operators to obtain the response of a system: `step-response` and `input-response`. `step-response` operator applies unit step function to the system. It is in the form of `(step-response transfer-function)`. `input-response` is used to simulate the response of a system given an input. It is in the form of `(input-response input transfer-function)`.

Functionally, the two lines of expressions below are the same. The second is more flexible though, because it is possible to use the optional parameters of `#step()` input.

```
=> (step-response substituted-system)
=> (input-response #step() substituted-system)
```

These operators return the data which includes points in time and corresponding instantaneous system response values. This data may be useful as is, but usually a visualization of it is more useful. Plotting the data returned by these operations on a graph is explained in the next subsection.

2.1.6 Plotting the response of a system

There are two plotting operations available. Given one or more than one plottable data: The first plotting option, `plot-together` operator, plots all the given data on the same graph. The second one, `plot-separately`, plots each of the given data on their own graph.

So, `plot-together` plots things together and `plot-separately` plots things separately.

There are some system response and input signal visualizations below, created using these operators.

```
=> (plot-together
     (step-response substituted-system)
     #step())
=> (plot-separately
     (input-response #step() substituted-system)
     (input-response #ramp() substituted-system)
     (input-response #sine() substituted-system))
```

`together` operator may be used to merge two different plottable data into a single plottable data. The time series part of both data must be the same for this operation to succeed.

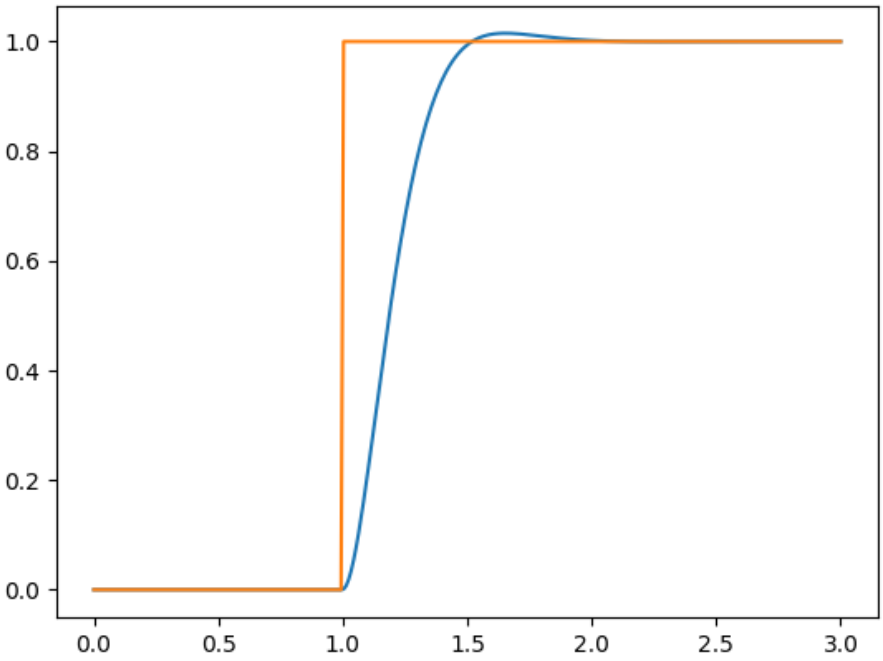

Figure 2.1 : Step response of a system and unit step function plotted using `plot-together`

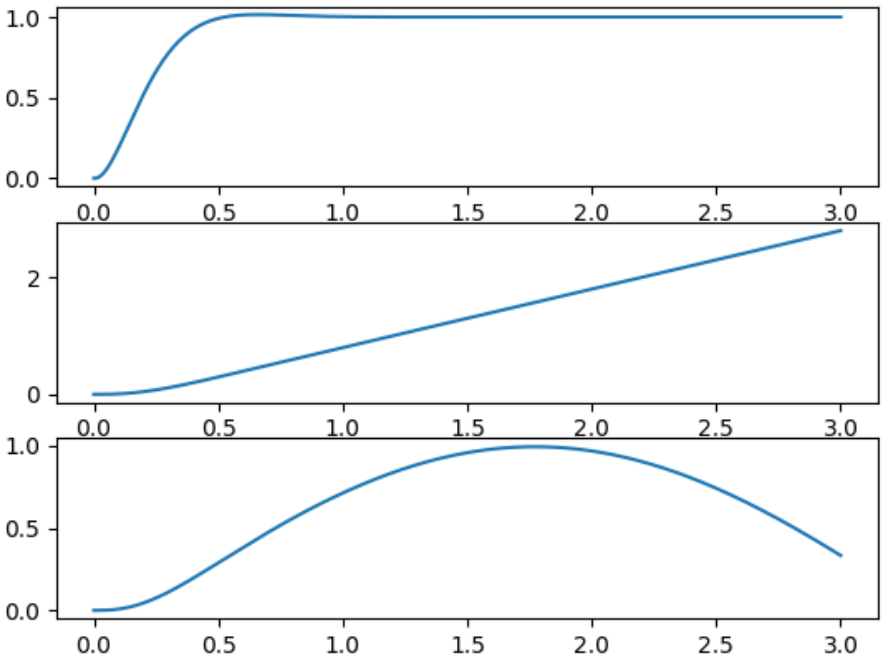

Figure 2.2 : Step, ramp and sine input responses of a system plotted using `plot-separately`

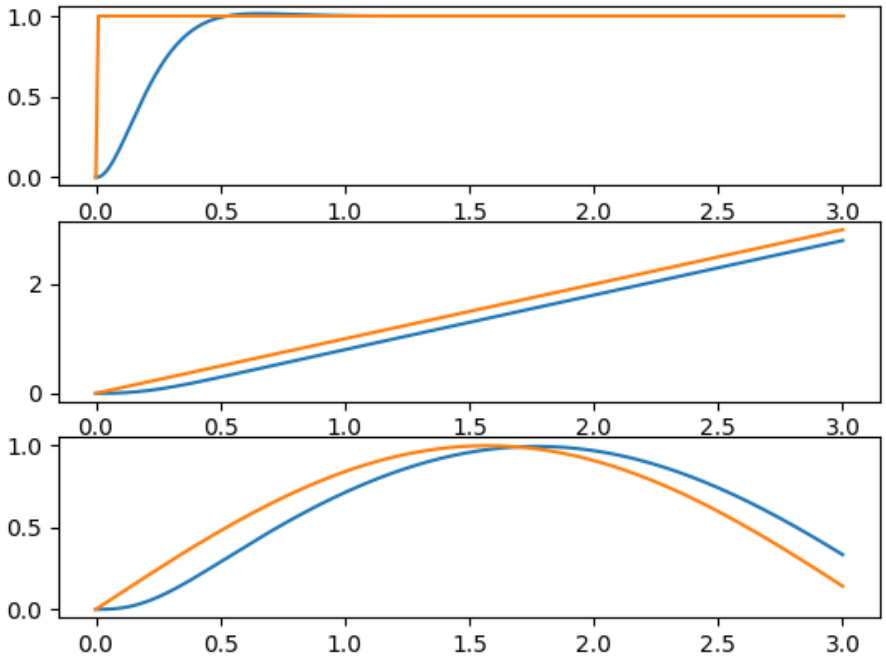

Figure 2.3 : Step, ramp and sine input responses of a system and their respective input functions plotted using `plot-separately` and `together`

```
=> (plot-separately
     (together
       (input-response #step() substituted-system)
       #step())
     (together
       (input-response #ramp() substituted-system)
       #ramp())
     (together
       (input-response #sine() substituted-system)
       #sine()))
```

2.1.7 Other plotting operations

All of the plots in this section share the same form of `(plot-name system-name)`. The usage and result of each operation is demonstrated in their following respective subsection.

2.1.7.1 Pole-zero plot

```
=> (pole-zero-plot substituted-system)
```

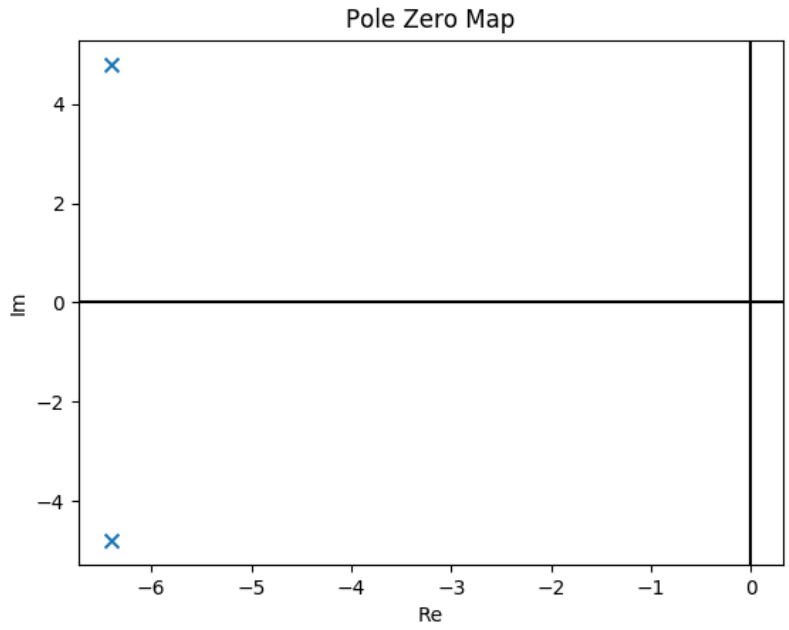

Figure 2.4 : `pole-zero-plot` of a system

2.1.7.2 Bode plot

```
=> (bode-plot substituted-system)
```

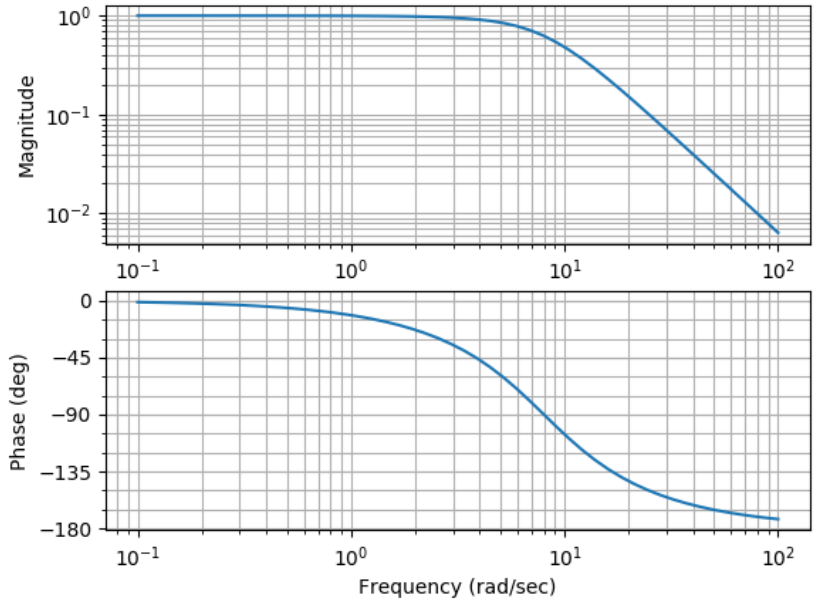

Figure 2.5 : `bode-plot` of a system

2.1.7.3 Nyquist plot

```
=> (nyquist-plot substituted-system)
```

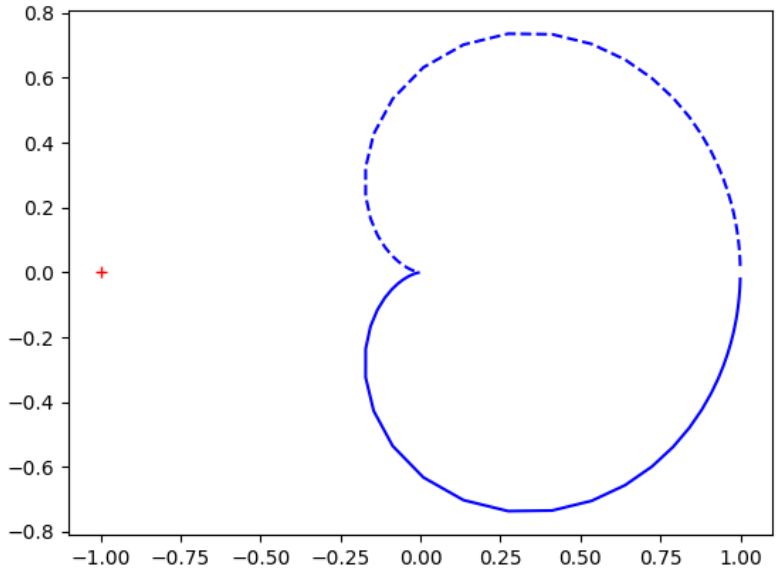

Figure 2.6 : `nyquist-plot` of a system

2.1.7.4 Root locus plot

```
=> (root-locus-plot substituted-system)
```

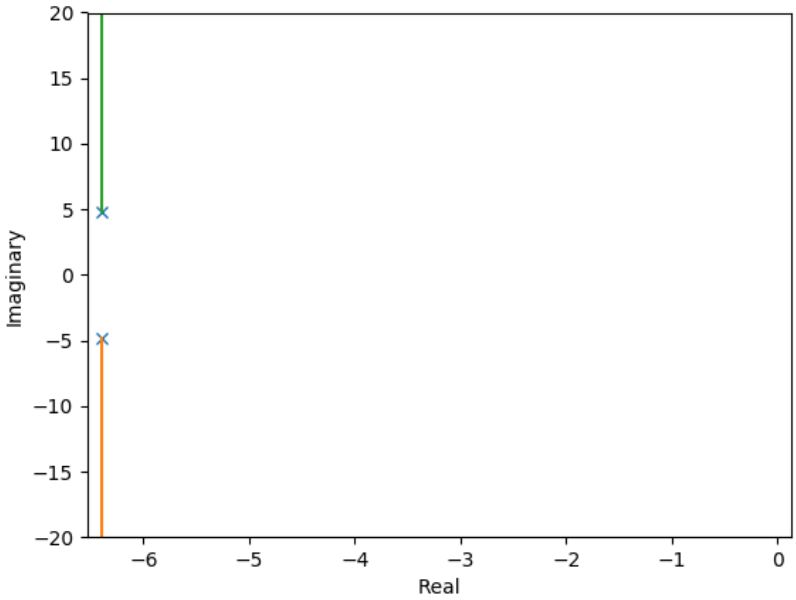

Figure 2.7 : `root-locus-plot` of a system

2.1.8 Expressing block diagrams

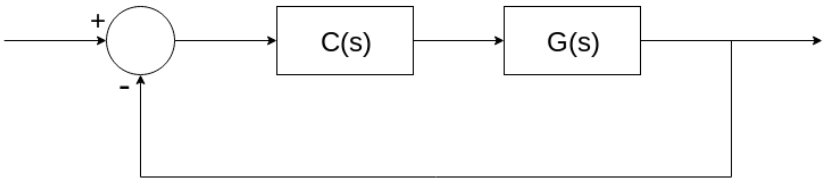

Figure 2.8 : A block diagram representing a control system

Block diagrams can be expressed using the `connect` operator. `connect` operator also has an alias: `o`. The reason behind the alias name is that the letter "o" represents the sum node of block diagrams. After the sum node, there may be more than one forward path and one feedback path. The below program fragment shows the control systems DSL representation of the block diagram in Figure 2.8.

```
=> (define p-controller #tf(10))
=> (o
     (> p-controller substituted-system)
     (^ #tf(1)))
```

The symbol > represents the *forward* path, and the symbol `^` represents the *feedback* path. One of them carries a signal forward from the sum node, the other one feeds a signal back to it. The feedback sign is negative by default, but the sign can be explicitly expressed using the symbols `-^` and `+^` to mean negative and positive feedback respectively.

The `o` operator returns a transfer function object, thus this expressions can be used nestedly to express more complex system interconnections.

3. IMPLEMENTATION DETAILS

3.1 Choice of implementation language

The domain-specific language is implemented using the Hy programming language. Hy is a dialect of Lisp which is embedded in Python [6]. Being a Lisp and its interoperability with Python are the reasons for choosing Hy as the implementation language.

Interoperability with Python is desired because of the extensive ecosystem of Python. The control systems DSL uses Sympy package to handle symbolic mathematics and python-control package to handle control systems related operations. matplotlib package is used for plotting and data visualization.

Being a Lisp is desired because of Lisp's simple and powerful syntax. The simplicity of Lisp's syntax makes it possible for a program to treat its code as data and manipulate it. This feature enables macros to take code as their input, transform it to another form, add to or take from it, and return new code to be executed. The control systems DSL uses macros extensively to implement syntactic abstractions.

3.2 Operators of the control systems DSL

This section roughly describes the implementation details of operators provided by the control systems DSL. The source code, so the concrete implementations of the operators can be found in the open-source repository of the language [2].

`(define symbol value "Documentation string")` binds the value to the symbol using Hy's `setv` and attaches the documentation string to the value if the documentation string exists. `define` is a macro.

`(documentation symbol)` returns the documentation string attached to the value named with symbol if there exists a documentation string attached. `documentation` is a macro.

`define-transfer-function` operator parses its arguments to create a transfer function object, attachs the documentation string to the object if it is available, and names the object with the provided symbol. `define-transfer-function` is a macro.

`numerator`, `denominator` and `sampling-period` operators are simple wrappers around methods of transfer function objects. They are functions.

`substitute` operator is a macro. It merges the provided name-value pairs and defined values in the local scope to supply them to the provided transfer function object.

`#tf` is a tag macro. It basically turns the provided symbols to a transfer function by converting them to a string which is evaluated in an environment where `s` is a transfer function.

`connect` (also usable as `o`) operator is a macro. It parses its body and turns it into function calls to system interconnection functions `parallel`, `series` and `feedback` of python-control package.

Input signal expressions `#step()`, `#ramp()`, `#parabola()` and `#sine()` are implemented using Hy's tag macros. They take some optional parameters. The expressions are parsed and transformed into a call to the internal `define-input-signal` function which returns the data representing input signals.

`(step-response system-name)` form is transformed into a `input-response` form with `#step()` as its input. `step-response` is a macro.

`(input-response input system-name)` form is transformed into a call to python-control package's `forced-response` function. The system denoted with system-name is converted to python-control's transfer function object from the control systems DSL's transfer function object. `input-response` is a macro.

In the cases of `bode-plot`, `nyquist-plot`, `pole-zero-plot` and `root-locus-plot`, the forms of these operations are transformed into calls to python-control package's `bode_plot`, `nyquist_plot`, `pzmap` and `rlocus` functions, respectively. These operators are macros.

`plot-together` and `plot-separately` are functions that use the plotting facilities of matplotlib package to visualize their arguments. `together` function merges its arguments' value parts by keeping the time series parts of them as a single common ground.

4. RESULTS AND DISCUSSION

4.1 Results

The control systems DSL developed throughout the project is bundled into a Python package and registered on the Python package index. Thus, the DSL is currently accessible and easily installable to any computer with Python installed.

The DSL is not suitable for any commercial use in its current condition. A field where the project may play a role is education. Since the DSL is free from unnecessary code and details, it can be used for educational purposes. The compatibility between the concepts of control systems and the domain-specific language would make it easier for students to express their ideas to computers.

4.2 Future possibilities

The control systems DSL currently supports transfer function models only. However, the python-control package used for control systems related operations also support state-space models [7], so it is possible to design and implement state-space related expressions to integrate state-space models into the DSL.

The project is developed in an open-source environment where everybody interested in can contribute to the project [2]. A development community may be formed around the project if the project can arouse interest and make some people enthusiastic about it.

It is also possible to write or integrate more packages for adding functionality beyond the python-control package. Dynamic equation expressions, where the transfer function or state-space models are automatically generated, would be good to experiment with.

REFERENCES

[1] **Fowler, M.** (2010). *Domain-Specific Languages*, Addison-Wesley Professional.

[2] **Hidayetoğlu, C.**, `https://github.com/celaleddin/gently`, A tool for designing and analysing control systems.

[3] **Seibel, P.** (2005). *Practical Common Lisp*, Apress.

[4] **Pitman, K.**, `http://clhs.lisp.se/Body/26_glo_f.htm`, Common Lisp HyperSpec.

[5] **Dybvig, R.K.**, **Hieb, R. and Bruggeman, C.** (1992). Syntactic Abstraction in Scheme, *Lisp and Symbolic Computation*.

[6] **Hy Contributors**, `http://docs.hylang.org/en/stable/`, Hy Documentation.

[7] **Python Control Systems Library Contributors**, `https://python-control.readthedocs.io/en/0.8.2/`, Python Control Systems Library Documentation.