An Embedded DSL for Simple Causal LTI Systems

T. Mitchell Roddenberry

25 March 2024

Abstract

We present a simple domain-specific language (DSL) embedded in `Hy`, a LISP dialect that is itself embedded in `Python`. This DSL is designed for describing and analyzing rational causal LTI systems, with capabilities for symbolic expressions, variables, and plotting. The goal of this DSL is to provide an environment in which students can learn about basic properties of LTI systems experimentally, without needing to either (i) learn an entire programming language syntax, or (ii) install heavy, proprietary, and overly-featureful simulation software.

The language strives to have syntax that is as simple as possible. It is of course not perfect in this regard. However, the hope in implementing this is to gradually improve the system to the point where users can operate it with what amounts to little more than colloquial language, much like the early LISP programs for expert systems.

This manual begins with a tutorial, written in "Q&A" style,[1] illustrating how the language works by example. Notably, the language is bundled with a chatbot that provides a "natural language" interface. For greater detail, the tutorial is followed by a technical reference on language features. See the appendices for information on installation.

[1]Inspired heavily by Daniel P. Friedman and Matthias Felleisen. *The Little Schemer.* MIT Press, 1995.

Contents

1 Tutorial

What is `s`?	`s` is a *symbol.*
Are there any other symbols?	Yes, some of them include `a` , `b` , `c` , `d` , `e` , `w` , `x` , `y` , `z`.
Is that all of them?	No, symbols are also given by capital letters and the Greek alphabet. Examples include `A` , `B` , `C` , `alpha` , `beta` , `gamma`.
Which symbol is the most important?	All symbols are equal, but some are more equal than others. `s` is the most equal symbol.
Why is `s` a special symbol?	Because we use it to represent the argument for the Laplace transform.
Are there any other special symbols?	Yes, `i` and `j` represent $\sqrt{-1}$.

What is `'(system (rat 1 [] []))`?	It is a causal LTI system with a rational transfer function.
Is it?	Ok, it is a *representation* of a causal LTI system with a rational transfer function. See: *The Treachery of Images*. We are not surrealists, nor are we French, so we will not distinguish between a system and its representation unless we have to.
Does `'(system (rat 1 [] []))` have any zeros?	No.
Does `'(system (rat 1 [] []))` have any poles?	No.
What is the gain of `'(system (rat 1 [] []))`?	The gain is 1.

So the transfer function of `'(system (rat 1 [] []))` is $H(s) = 1$?

Yes, that is correct.

I am getting tired of writing down `'(system (rat 1 [] []))`, is there a shorthand that I could use for this?

Yes, there are a few. Since it is a causal LTI system with no zeros or poles, it is determined only by its gain. The shorthand, then, is `gain 1`, since it has a gain of 1.

What is `(gain 2)` short for?

`(gain 2)` is short for `'(system (rat 2 [] []))`.

`gain` creates causal LTI systems that scale their input by a constant factor

---

Are there any other types of causal LTI systems with rational transfer functions?

Yes, of course there are.

What is `'(system (rat 1 [0] []))`?

It is a causal LTI system with a gain of 1, a zero at 0, and no poles.

What is the transfer function of `'(system (rat 1 [0] []))`?

It is $H(s) = s$.

Woah, $H(s)$ looks like the symbol s̲, right?

Good observation, I should have said that

```
(transfer-function
  '(system
     (rat 1 [0] [])))
```

is s̲.

What does the system `'(system (rat 1 [0] []))` do?

It is a system that takes the derivative of its input.

Is there a shorthand for `'(system (rat 1 [0] []))`?

Yes, `(derivative)` is shorthand for `'(system (rat 1 [0] []))`.

What is the opposite of differentiation?	Integration.
What is the gain of a causal LTI system that integrates its input?	The gain of such a system is 1.
Does the transfer function of a causal LTI system that integrates its input have any zeros?	No.
Does the transfer function of a causal LTI system that integrates its input have any poles?	Yes, it has a pole at 0.
Is `'(system (rat 1 [] [0]))` a causal LTI system that integrates its input?	Yes; a piece of shorthand for it is `(integrate)`.

`derivative` , `integrate` create causal LTI systems

What happens if I say `(declare S gain 2)`?	It creates `(gain 2)`, but we have given it the name S.
After that, what happens if I type in S?	You will get `(gain 2)` back.
How would I create a differentiator denoted by D?	`(declare D derivative)`
Does `(declare I integrate)` create an integrator named I?	Yes, it does.
But S , D , I are symbols, are they not?	That is true, but `declare` renames them.
Can I `(declare s gain 1)`?	You could, but I would not suggest it!

`declare` is a prefix that names systems

What if I want to put systems together?

Sure, what do you have in mind?

I would like to make a system that takes a derivative, then scales by a factor of 2.

Ok, so you would like to put a (derivative) and a (gain 2) in series?

Yes! How do I do that?

(compose (derivative)(gain 2)).

What LTI system is that?

```
['compose '(system
              (rat 1 [0] []))
           '(system
              (rat 2 [] []))]
```

Can I give that system a name?

Sure, just use declare.

Like this?

```
(declare diffscale compose
                     (derivative)
                     (gain 2))
```

Excellent.

Are there other ways to combine systems?

Yes, you can put systems in parallel using parallel or sum, and you can create feedback systems using feedback.

Explain parallel to me.

(parallel S1 S2) creates an LTI system that applies S1\lstinline and \lstinlineS2! to the input, then sums the respective output.

Tell me the transfer function of

```
(parallel (derivative) (integrate)
   )
```

s + 1/s

Does sum behave any differently?

No.

How many systems can I put in parallel using parallel?

As many as you want: (parallel S1 S2 S3) is valid syntax.

Some systems have feedback loops, can I create those?

Yes, use (feedback S1 S2).

Tell me the transfer function of

```
(feedback (gain 2) (integrate))
```

2/(1+2/s)

How many systems can I apply feedback to?

Only two: but those systems can each consist of *composed, parallel,* or *feedback* systems themselves!

compose , parallel , feedback combine LTI systems

Typing some of these things in is a bit tedious: show me how to make a complicated system out of systems that I have already named.

Use the names as shorthand:

```
(declare S1 gain 2)
(declare S2 derivative)
(declare S3 integrate)
(declare S4 feedback (compose S1
   S2) S3)
```

What is the transfer function of this system?

You can find it like this:

```
(transfer-function S4)
```

is 2*s/3.

Is there a one-liner for defining `S4` like above, while also naming the systems `S1,S2,S3`?

Like this?

```
(declare S4 feedback
  (compose (declare S1 gain 2)
           (declare S2 derivative)
              )
  (declare S3 integrate))
```

Use `declare` statements to name things along the way

All of the systems we have built so far only have poles and zeros at 0.

That is correct.

What is the gain of `'(system (rat 2 [] [3]))`?

2

What are the zeros of `'(system (rat 2 [] [3]))`?

There are none.

What are the poles of `'(system (rat 2 [] [3]))`?

There is a single pole at 3.

What is `(rational 2 [] [3])`?

`'(system (rat 2 [] [3]))`

What are the zeros of `'(system (rat 3 [1+1j 1-1j] []))`?

There is are two zeros at $1 \pm j$, where $j = \sqrt{-1}$.

What is `(rational 3 [1+1j 1-1j] [])`?

`'(system (rat 3 [1+1j 1-1j] []))`

Neat, so `rational` creates causal LTI systems with rational transfer functions from their gain, zeros, and poles.

You got it!

What if I know the numerator and denominator of the transfer function, but not the gain, poles, and zeros?

Use `rational-polynomial`.

What is `(rational-polynomial "2*s**2-2*s+2 ""s+1")`?

`'(system (rat-poly 2*s**2-2*s+2 s+1))`.

What is the transfer function of `'(system (rat-poly 2*s**2-2*s+2 s+1))`?

The transfer function can be computed as follows.

```
(transfer-function '(system
                      (rat-poly
                        2*s**2-2*
                          s+2
                        s+1)))
```

is `(2*s**2-2*s+2)/(s+1)`.

`rational` , `rational-polynomial` create LTI systems

What does `(declare S1 rational 1 [1] [-1 ])` do?

Names `S1` as an LTI system with gain 1, a zero at 1, and a pole at -1.

Is `S1` stable?

Good question.

```
(?is S1 stable)
```

yields `True`, so yes.

Why is it stable?

Because its region of convergence contains the imaginary axis.

Are you sure?

Yes.

```
(region-of-convergence S1)
```

yields `re(s)>-1`.

Can you check the convergence of the Laplace transform at $1 + j$?

Sure.

```
(?is S1 convergent-at 1+1j)
```

yields `True`.

What are the zeros of `S1`?

Try the command

```
(zeros S1)
```

which yields `[1]`.

What are the poles of `S1`?

Similarly,

```
(poles S1)
```

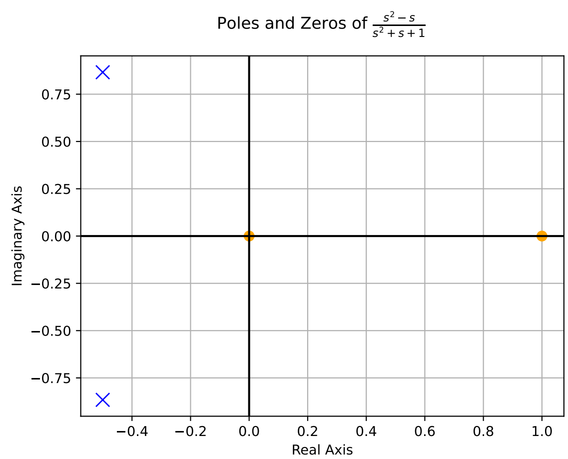
yields `[-1]`.

Ask questions with `?is`

What is the pole-zero plot of the system declared below?

```
(declare S1 feedback
            (rational 1 [1] [-2])
            (compose (integrate)
                     (gain -1)))
```

`(pole-zero-plot S1)` displays the figure below:



Is `S1` stable?

Yes: `(?is S1 stable)` yields `True`.

Does this make sense?

Yes, the poles are to the left of the imaginary axis.

```
(region-of-convergence S1)
```

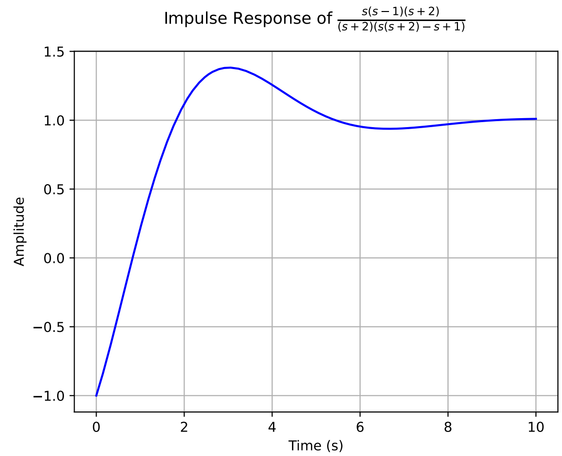
yields `re(s) > -1/2`.

Is the computer smart enough to cancel out poles and zeros that overlap?

Yes! But it may be subject to numerical error, so be careful.

What is the impulse response of `S1`?

You can plot it with `(impulse-plot S1)`

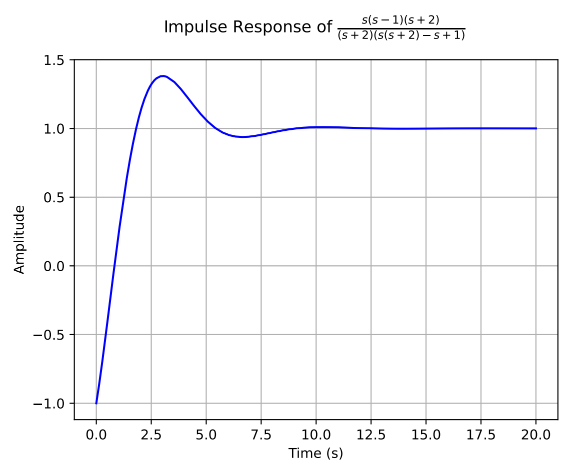


What if the impulse response isn't well-defined?

Then you should try `(step-plot S1)` or `(ramp-plot S1)`.

Can I see the impulse response for 20 seconds?

Yes, specify with `(impulse-plot S1 :upper-limit 20)`.



Visualize with `pole-zero-plot` , `impulse-plot` , `step-plot` , `ramp-plot`

What is the transfer function of `(rational 1 [] [a])`?

`(transfer-function (rational 1 [] [a]))` is `1/(s-a)`.

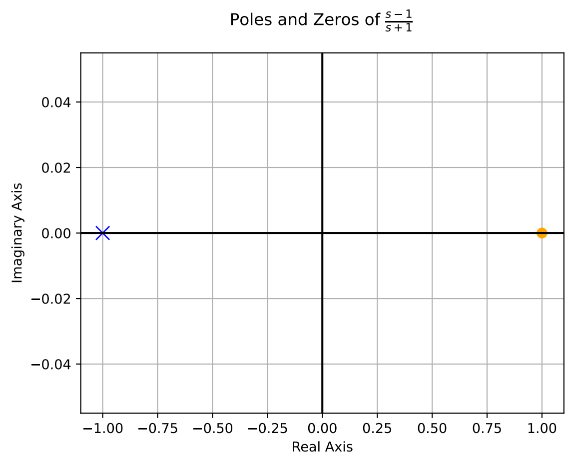
What is the pole-zero plot of the system declared below?

```
(declare S1 rational 1 [a] [b])
```

There is not enough information to tell.

What is the pole-zero plot when $a = 1$ and $b = -1$?

Let us take a look. `(pole-zero-plot S1 {a 1 b -1})` yields



What are the zeros of `S1`?

`(zeros S1)` yields `[a]`.

What are the zeros of `S1` when $b = 1$?

`(zeros S1 {b 1})` yields `[a]`.

What are the poles of `S1` when $b = -1$?

`(poles S1 {b -1})` yields `[-1]`.

What is the region of convergence of `S1`?

`(region-of-convergence S1)` yields `re(s)>re(b)`.

Is the Laplace transform of `S1` convergent at $s = b + 1$?

Yes: `(?is S1 convergent-at (+ b 1))` yields `True`.

When there are degrees of freedom, bind symbols with `{...}`

What are the poles of the system declared below?

`(declare S1 rational 1 [] [a])`

`(poles S1)` yields a.

`(?is S1 stable)`

`re(a) < 0`

What is `(region-of-convergence S1)`?

`re(s) > re(a)`

Is `S1` stable when $a = 1$?

How do you think we should see?

Like this maybe? `(?is S1 {a 1} stable)`

That won't work. You need to use special syntax when using `?is`.

`(?is S1 stable :bind {a 1})`

Ahh, ok, `:bind` tells us what `{a 1}` does.

Right, you need to be sure to use `:bind` whenever you combine `?is` and symbolic expressions.

When using `?is`, bind symbols with `:bind {...}`

What are the poles of the system declared below?

`(declare S1 rational 1 [a] [a b c])`

`(poles S1)` yields `[b c]`.

What is the region of convergence of `S1`?

`(region-of-convergence S1)` yields `re(s) > Max(re(b), re(c))`.

`(?is S1 stable :bind {c (+ b 1)})`

`re(b) + 1 < 0`

1.1 Exercise

Problem: Construct a feedback system using a single-pole filter and constant-gain feedback.

1. Determine the transfer function.
2. Determine the poles and zeros in terms of the gain of the filter, the pole of the filter, and the gain of the feedback system.
   (a) Determine a set of parameters such that the system is stable, then plot the poles and zeros.
   (b) Determine a set of paramaters such that the system is unstable, then plot the poles and zeros.
3. Is the system stable when the filter has gain 1 and a pole at 1, and the feedback has a gain of 2?
   (a) If so, plot the impulse response.
   (b) If not, plot the step response.
4. Is the system stable when the pole of the filter is equal to the product of the gain of the filter and the gain of the feedback? What about when it is equal to the product of the gains minus one?

Solution:

exercise1.lti

```
(import systems_sandbox *)
(require systems_sandbox *)

(declare S1 feedback (rational b [] [a]) (gain k))

(print "1.␣Transfer␣function␣is:␣"
       (transfer-function S1))

(print "2.␣Poles:␣"
       (poles S1))
(print "2.␣Zeros:␣"
       (zeros S1))

(print "RoC:␣"
       (region-of-convergence S1))

;; stable case
;; a = 1 b = 1 k = 3
(print "2a.␣Is␣{a=1,␣b=1,␣k=3}␣stable?␣"
       (stable S1 {a 1 b 1 k 3}))
(pole-zero-plot S1 {a 1 b 1 k 3})

;; unstable case
;; a = 1 b = 1 k = -2
(print "2b.␣Is␣{a=1,␣b=1,␣k=-2}␣stable?␣"
       (stable S1 {a 1 b 1 k -2}))
(pole-zero-plot S1 {a 1 b 1 k -2})

;; a = 1 b = 1 k = 2
(if (stable S1 {a 1 b 1 k 2})
  (do (print "3.␣{a=1,␣b=1,␣k=2}␣is␣stable.")
      (impulse-plot S1 {a 1 b 1 k 2}))
  (do (print "3.␣{a=1,␣b=1,␣k=2}␣is␣not␣stable.")
      (step-plot S1 {a 1 b 1 k 2})))

;; equality condition: a = bk
(print "4.␣Is␣{a=b*k}␣stable?␣"
       (stable S1 {a (* b k)}))
(print "4.␣Is␣{a=b*k-1}␣stable?␣"
       (stable S1 {a (- (* b k) 1)}))
```

Program output:

```
1. Transfer function is:  TransferFunction(b, -a + b*k + s, s)
2. Poles:  [a - b*k]
2. Zeros:  []
RoC:  re{s} > re(a) - re(b*k)
2a. Is {a=1, b=1, k=3} stable?  True
2b. Is {a=1, b=1, k=-2} stable?  False
3. {a=1, b=1, k=2} is stable.
4. Is {a=b*k} stable?  False
4. Is {a=b*k-1} stable?  True
```

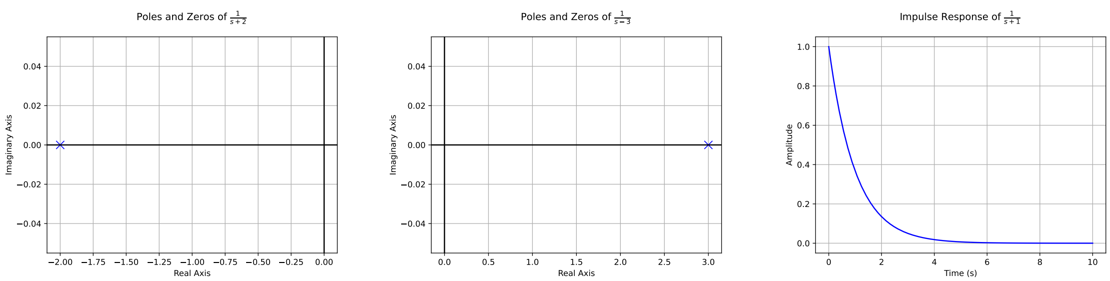


Figure 1: Plots for first exercise. Parts 2a, 2b, and 3 from left to right, respectively.

1.2 LTIza: A Helpful Chatbot

ELIZA is a famous chatbot[2] written by Joseph Weizenbaum in the 1960s, capable of simulating Rogerian psychobabble via simple pattern matching rules. For those who are averse to learning programming language syntax, such as the one demonstrated earlier, a simple chatbot named LTIza is provided.

To start a conversation with LTIza, you need to be in an interactive session and have a system declared with some name, say S1. Then, run the command (ask-ltiza S1) to enter a REPL.[3] Within the LTIza REPL, you can give a variety of loosely-structured queries, which the program will then attempt to parse into a command. If the command is a "verb," so-to-speak, then it will print the command and run it. If the command defines a "noun," in particular if it binds symbols to values, then it will show the set of bound symbols in the prompt. See below for an example session.

```
=> (declare S1 rational 1 [a] [-2 -1 b])
=> (ask-ltiza S1)
LTIza {} => what are the poles
  (poles S {})
  [-2, -1, b]
LTIza {} => what are the zeros
  (zeros S {})
  [a]
LTIza {} => bind a equal to -3
LTIza {a: -3} => bind b to 1
LTIza {a: -3, b: 1} => poles
  (poles S {a -3 b 1})
  [-3, -2, -1]
LTIza {a: -3, b: 1} => is the system stable
  (stable S {a -3 b 1})
  True
LTIza {a: -3 b: 1} => unbind a
LTIza {b: 1} => what is the region of convergence
  (region-of-convergence S {b 1})
  re(s) > Max(-2, re(a))
LTIza {b: 1} => is the system stable
  (stable S {b 1})
  re(a) < 0
LTIza {b: 1} => quit
=>
```

You can type help at the LTIza {} => prompt to get more information about available functionality. Be encouraged by this tool: use the code output to learn how to write your own programs in more formal language!

[2]Originally called "chatterbots."

[3]Read-evaluate-print loop: the usual interactive interface for shell-like programs.

2 Reference

2.1 Language Core

The core language features can be found in `systems_sandbox.hy`.

2.1.1 Defining Systems

`(rational [[gain 1]] [zeros []] [poles []])`

Returns a representation of a causal LTI system with rational transfer function of the form

$$H(s) = \frac{\text{gain} \times \prod_{z \in \text{zeros}} (s - z)}{\prod_{p \in \text{poles}} (s - p)}.$$

`gain` is a numeric value or a symbol, `zeros` is a list of numeric values or symbols, and `poles` is a list of numeric values or symbols.

`(rational-polynomial [[numerator 1] [denominator 1]])`

Returns a representation of a causal LTI system with transfer function of the form

$$H(s) = \frac{\text{numerator}}{\text{denominator}}.$$

`numerator,denominator` are strings coinciding with a polynomial of numerics and symbols.

`(gain [[gain 1]])`

Alias for `(rational gain)`.

`(derivative [])`

Alias for `(rational 1 [0] [])`.

`(integrate [])`

Alias for `(rational 1 [] [0])`.

`(compose [#* l])`

Composes multiple systems in series. Takes a variable number of arguments.

`(feedback [S T])`

Creates a feedback system from two systems.

`(parallel [#* l])`

Combines multiple systems in parallel. Takes a variable number of arguments.

`(sum [#* l])`

Same as `parallel`.

`(declare [S #* H])`

Macro to assign whatever expression is given by `H` to the name `S`. Note: `H` should not be enclosed by parentheses, *e.g.*, `(declare S gain 2)`.

2.1.2 System Analysis

`(transfer-function [S [bind {}]])`

Displays the transfer function of the system `S` after binding symbols according to the dictionary `bind`.

`(zeros [S [bind {}]])`

Returns the zeros of the system `S` after binding symbols according to the dictionary `bind`.

`(poles [S [bind {}]])`

Returns the poles of the system `S` after binding symbols according to the dictionary `bind`.

`(region-of-convergence [S [bind {}]])`

Returns inequality expressing values of symbol `s` for which the transfer function of the system `S` is convergent, after binding symbols according to the dictionary `bind`.

`(stable [S [bind {}]])`

After binding symbols according to the dictionary `bind`, either returns Boolean (True/False) value indicating if the system `S` is stable, or returns inequality indicating conditions underwhich `S` is stable if there are remaining degrees of freedom.

2.1.3 Plotting

`(pole-zero-plot [S [bind {}]] #** kwargs)`

Plots the pole-zero plot of the system `S` after binding symbols according to the dictionary `bind`. Passes `kwargs` to plotting function.

`(impulse-plot [S [bind {}]] #** kwargs)`

Plots the impulse response of the system `S` after binding symbols according to the dictionary `bind`. Passes `kwargs` to plotting function.

`(step-plot [S [bind {}]] #** kwargs)`

Plots the step response of the system `S` after binding symbols according to the dictionary `bind`. Passes `kwargs` to plotting function.

`(ramp-plot [S [bind {}]] #** kwargs)`

Plots the ramp response of the system `S` after binding symbols according to the dictionary `bind`. Passes `kwargs` to plotting function.

`(frequency-plot [S [bind {}] [sigma0 0]] #** kwargs)`

After binding symbols according to the dictionary `bind`, plots the magnitude of the transfer function of the system `S` at $H(\sigma_0 + j\omega)$ for varying values of ω. Passes `kwargs` to plotting function.

A Installation & Getting Started

At this point, I haven't packaged this software in a slick way – it relies on the user having a working Python 3 installation and being able to install the required packages via `pip` or some other means. For now, you should just run the included script `main.py` from within the project directory. If I get around to it, I'll figure out a good way to package everything.

A.1 Prerequisites

This program was developed with `Python 3.11.8`, but it will probably work fine on earlier version of `Python 3.11`, and even on `Python 3.9` or `Python 3.10`. No promises, though.

You will also need a `pip` installation alongside your `Python` install. Chances are that `pip` was distributed with `Python`.

The rest of the installation tutorial will use "virtual environments," another feature that comes with `Python`.

A.2 Installation

The left column of the instructions describes what you are doing, and the right column shows it in a terminal[4] session. The `$` at the beginning of each line refers to the terminal prompt. You should not type it in yourself.

Navigate to the directory where you have downloaded the source code. For me, that directory is called `elec242`, and I obtain the source code via `git`. You may wish to get the source code some other way; that's fine.

```
$ git clone https://github.com/tmrod/
    systems-sandbox ~/elec242
$ cd ~/elec242
```

Create a virtual environment in that directory, then activate it.

```
$ python -m venv .venv
$ . ./.venv/bin/activate
```

With the virtual environment activated, you may or may not see a change in your prompt. Either way, continue on to install the necessary packages.

```
(venv)$ pip install -r requirements.txt
```

Hopefully that worked. Congratulations: you have now installed everything you need!

[4]In particular, a shell like `bash` or `sh`. I don't know anything about using Windows.

A.3 Getting Started

I'll assume that you followed the instructions from the previous section. Navigate to the directory where you installed everything and activate the virtual environment.

```
$ cd ~/elec242
$ . ./venv/bin/activate
(venv) $
```

To start up a REPL, run the script `main.py` with no arguments. Be sure that you run this *from the install directory*. Once again, I haven't bothered to package the program files in a way that is convenient. This will drop you into a REPL. The startup banner might vary depending on your particular system.

```
(venv) $ python main.py
Hy 0.27.0 using CPython(main) ...
=>
```

The prompt `=>` corresponds to the REPL in which you can type in commands to describe and study systems. This is a good time to start working through Section 1. For instance, you could declare a system, then ask the chatbot about it.

```
=> (declare S1 rational 1 [a] [-2 -1 b])
=> (ask-ltiza S1)
LTIza {} => what are the poles
  (poles S {})
  [-2, -1, b]
LTIza {} => quit
```

And to exit the REPL, `<Ctrl+d>` works.

```
=> <Ctrl+d>
(venv) $
```

In Section 1.1, we put the solution to a question in a file and ran it all at once, rather than typing each command manually at the REPL. Suppose there is some file containg commands, say, at `manual/exercise1.lti`. It can be run like this:

```
(venv) $ python main.py manual/exercise1
    .lti
1. Transfer function is:
    TransferFunction(b, -a + b*k + s, s)
2. Poles:  [a - b*k]
2. Zeros:  []
RoC:  re(s) > re(a) - re(b*k)
2a. Is {a=1, b=1, k=3} stable?  True
2b. Is {a=1, b=1, k=-2} stable?  False
3. {a=1, b=1, k=2} is stable.
4. Is {a=b*k} stable?  False
4. Is {a=b*k-1} stable?  True
```

Notice: at the top of the script there are two important lines. When running a file as a script like this, you need to include those to make everything work. I'll figure out how to remove the need for those once I package this software properly.

```
(import systems_sandbox *)
(require systems_sandbox *)
```