Demystifying the Syntax and Semantics of Control Theory

Developing course material using Domain-Specific Languages

Filip Nylander Jakob Fihlman Christian Josefson Elin Ohlman
Tommy Räjert Simon Hägglund

Spring - 2020

Table of Contents

1 Introduction

This is a supplementary learning material for the course "Control theory" (ERE103) at Chalmers school of technology. The purpose of this material is to give computer science students a more familiar introduction to the subject of control theory. While aimed towards students taking the course ERE103, this learning material is suitable for everyone with an interest in learning more about control theory and is familiar with haskell. If you want to learn haskell just to be able to follow this learning material or just brush up your skills we strongly reccomend learnyouahaskell by Miran Lipovača.

The method we use in this material is to create domain-specific languages (DSLs) for the course. A DSL is a programming language specialised for a domain, be it math, physics, astronomy, etc. In our case, we will introduce a new DSL for each section, each specialised for the content of said section. For students who already attended the course "DSLsofMath" this approach should be nothing new. For students who have not, additional details are provided throughout the first chapter; "Complex Numbers".

All DSLs in this material are written in Haskell, so the syntax should be familiar from previous courses. If you are not familiar with Haskell or just want to brush up your skills, we recommend "Learn you a Haskell for Great Good!" by Miran Lipovača (http://learnyouahaskell.com/).

We hope you find this reading useful. Best of luck!

2 Prerequisites

This section is dedicated to refreshing your understanding of key concepts that are not technically part of control theory, but are nonetheless necessary in order to understand it. In addition, this section includes some basic instructions of how DSLs work in practice.

These subjects are only touched upon on a very basic level, thus if you are already feeling confident about them, this section can be safely skipped.

2.1 Complex numbers

In short, the complex numbers are an extension of the real numbers in the same way that the real numbers are an extension of the rational numbers. With the reals, we saw the addition of a number of constants(e, π, etc.). This time around, we add imaginary numbers.

First, some technicalities. The complex numbers are built on the real numbers, but the real numbers are unfortunately not very easily implementable in computers. In reality we have to approximate. In order to avoid this problem, we're going to introduce a type synonym:

```
1 type R = Double
```

Now, whenever we want something to be a real number, we can use the type `R`. Type synonyms are just that — synonyms — so whenever you see `R` throughout the text you could exchange it with `Double` and get the exact same result. We hope you see the reason in using `R`, however.

2.1.1 The Imaginary Number Concept

So what are imaginary numbers? In short, imaginary numbers are a way to solve some equations that are otherwise impossible to solve. For example, if we wanted to find a root to -1 we'd need to use imaginary numbers.

Let's start with the definition:

$$i = \sqrt{-1},$$

or put otherwise,

$$i^2 = -1.$$

What we've got so far is just one number. So what? Well, let's look at what we can do with them.

Imaginary numbers have a tendency to show up alongside real numbers. As such, mathematicians have standardized complex numbers as being read "a + bi." Some examples of complex numbers written this way are $2 + 3i$, $\pi + ei$ and $1 + 0i$. We'll get back to that soon, but first we want to introduce our own representation.

```
data Complex = Complex (R, R)
    deriving (Show)
```

This is the first step to creating a domain-specific language: establishing how our 'words' (or in this case numbers) are written. We create a data type called `complex`, which holds a pair of values.

Now you might ask yourself: why did we choose to put our doubles in parenthesis? The answer lies in the complex plane and how complex numbers can be seen as coordinates in a diagram. For example, see the following figure:

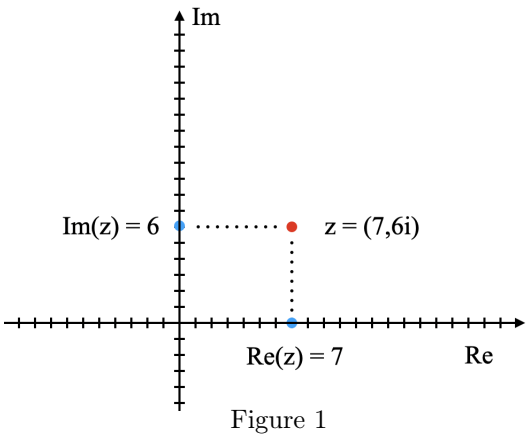


Figure 1

In the picture you see a complex plane with the complex number $(7, 6i)$ marked. Note the similarity to coordinate planes. TODO: write a little more about how real works like x and im as y.

2.1.2 Basic Operators

For our first operators, let's just create a pair of functions that extract the real and imaginary parts of the Complex number. As can be seen in the picture above, the real numbers exist along the x-axis, and the imaginary ones exist along the y-axis. As such:

```
whatsReal :: Complex -> R
whatsReal (Complex (x,y)) = x
```

Exercise 1. Implement the other function, `whatsImaginary :: Complex -> Real`.

Well, that was easy enough. Note that had we chosen a different representation of complex numbers, say the polar one, these would be a little harder.

First, let's define the most fundamental of all math operators: addition. Please look at the picture below.

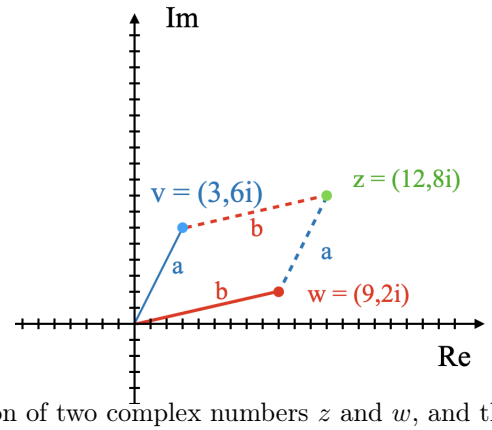


Figure 2: Addition of two complex numbers z and w, and the resulting number v.

In the picture we see the complex numbers z, w and v. We want adding them to behave like adding two points in a graph: the result of going from the origin to one of the points and then following the other from there. For example, in 2, we see that $v = z + w$, because we can go from the origin to $(9, 2i)$ and then follow $(3, 6i)$ from there.

When working in rectangular form we can implement addition similar to how it works with Cartesian coordinates: add reals to reals and imaginaries to imaginaries. Since we represent our complex numbers as a pair, we only have to add the components of the pair to each other:

```
add :: Complex -> Complex -> Complex
add (Complex (r1,i1)) (Complex (r2,i2)) = (Complex (r1+r2,i1+i2))
```

Note that addition was easy to implement because of our choice of coordinates.

Next, let's add subtraction. This works according to the same principle as addition, where real affects real and imaginary affects imaginary.

```
sub :: Complex -> Complex -> Complex
sub (Complex (r1,i1)) (Complex (r2,i2)) = Complex (r1-r2,i1-i2)
```

2.1.3 Imaginary Geometry

Now, let's move on to the main reason why we chose to mimic coordinates when defining our datatype. No, it wasn't to simplify addition and subtraction. That was just a bonus.

The real reason is that it allows us to use complex numbers to express geometry. To do this, we'll need two core functions. First off, the absolute value.

Note that, since complex numbers exist on a plane rather than a line, we can't just turn them positive and call it a day. Instead, we'll have to call upon our good old friend Pythagoras:

```
absolute :: Complex -> R
absolute (Complex (real,imaginary)) = sqrt (real^2 + imaginary^2)
```

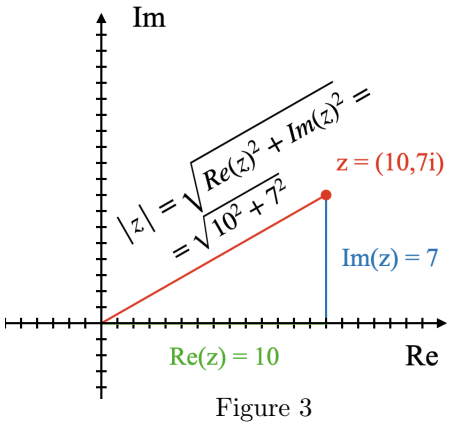


Figure 3

Since squaring always result in positive values, there's no need to worry about whether the values started out positive or negative.
Let's move on to arguments, a.k.a. angles.

Pythagoras' theorem is not the only geometric rule that can be combined with complex diagrams/-coordinates. In fact, all of them can. Let's do some tangents. Except when working with complex numbers, the resulting value is called an argument rather than an angle.

Unlike regular angles, however, the standard way to express a complex argument, also known as the "Principal Argument" does not include a complete 360degree circle from positive real to positive real. Instead, the principal argument uses two scales from the positive reals to the negative reals. One covers positive imaginary values and gives arguments ranging from 0 to π. The other covers negative imaginaries and gives arguments ranging from 0 to $-\pi$. See picture below for clarity.

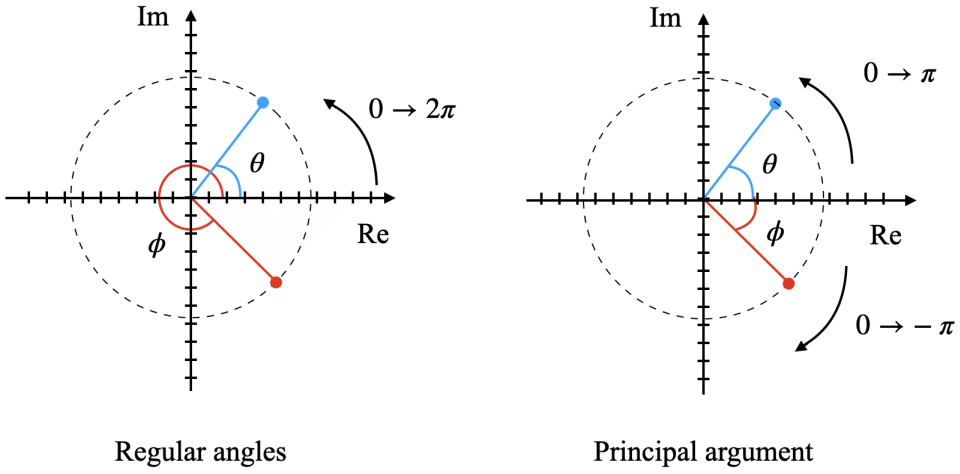


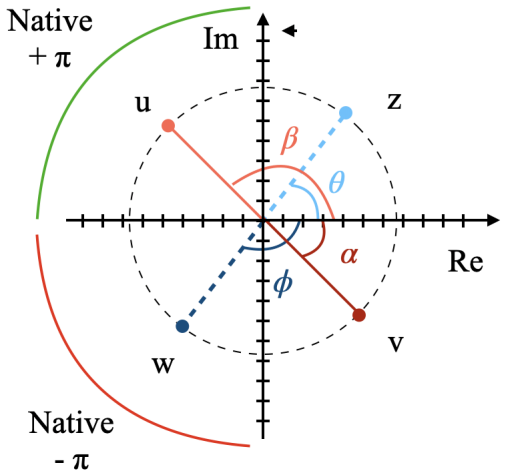
Figure 4

The good news is that the arcus tangent(arctan) operator supports this split natively. The bad news is that the arcusTangent operator input only describes the depression of the line; the koefficient.

As illustrated just below, what we'll have to do is program the operator to determine which quadrant the complex number is in, and "spin" the resulting argument accordingly.

$$tan(\theta) = \frac{Im(z)}{Re(z)} = tan(\phi) = \frac{Im(w)}{Re(z)} = k$$

$$tan(\alpha) = \frac{Im(u)}{Re(u)} = tan(\beta) = \frac{Im(v)}{Re(v)} = l$$



Where k is the slope of the "blue" dashed line and l is the slope of the "red" solid line

$arctan(k) = \theta \qquad arctan(l) = \alpha$

$arg(z) = arctan(k) = \theta$

$arg(w) = arctan(k) - \pi = \phi$

$arg(v) = arctan(l) = \alpha$

$arg(u) = arctan(l) + \pi = \beta$

Figure 5

```haskell
argument :: Complex -> Double
argument (Complex (real,imaginary))
  | real < 0 && imaginary > 0  = (atan (imaginary/real)) + pi
  | real < 0                   = (atan (imaginary/real)) - pi
  | otherwise                  = atan (imaginary/real)
```

Please note that our function is currently undefined whenever the real component is zero, due to division by zero. We could solve this with a further three lines describing specific cases for +90°, -90°, and true zero. In this case we chose not to in order to keep our code as clean as possible. Alternatively, we could've just used Haskells builtin version of the operator above, known as `atan2`. If we were lazy, that is. Which we are not.

The Absolute Value and the Principal Argument, with their powers combined, form an alternate way to express the "position" of a Complex number; by expressing the direction and distance from the origin. The later sections will have a field day exploring the possibilities of this, but for now let's move on.

2.1.4 Advanced Operators

There is one final function that awaits us; one final operator:
multiplication!
..
Ok, might not sound like much, but there's a reason why we're tackling it last.
Can you tell us how to multiply two Complex numbers, based on what we've been talking about so far?
..
The simple truth of the matter is, we can't. Each Domain-specific Language is, as the name suggests, only meant to express one domain. Our chosen domain was geometry, and we chose to express our Complex numbers as coordinates to aid with that.

That does not mean we can't build a function to multiply two Complex numbers, but it does mean we'll have to move outside the bounds of our DSL to accomplish it.

To wit; in regular math, complex numbers are normally written on the form "a + bi", I.E. reals + imaginaries. With that language in mind, basic algebra become a lot more intuitive. 'multiply Complex (r1,i1) Complex (r2,i2)' in our DSL becomes (r1 + i1)(r2 + i2). This can be solved the same way we solve any multiplication of additions; by multiplying each combination and adding the results:

$$(1+2i) * (3+i) =$$
$$1*3 \quad 1*i \quad 2i*3 \quad 2i*i$$
$$= 3 + i + 6i + 2i*i =$$
$$3 + 7i - 2 = 1 + 7i$$

Figure 6

As such, by cleverly switching languages, we can find solutions to problems that our current DSL struggles with.

We can now define an operator in our own DSL using the information provided by traditional algebra to reach a correct answer. There is still one annoying complication remaining, however; we've got one term containing i^2. How will we make that work with how we've defined our datatype? We never added a way to write powers!

Ach, if only there was a convenient comment regarding the definition of i that we brought up at the beginning of the section and fully expect you to have forgotten about.

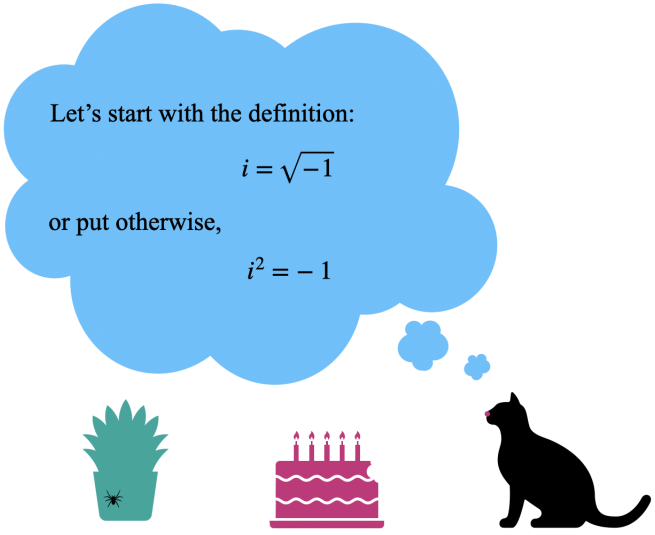


Figure 7

Oh wait, there is just such a thing! i^2 is the same as -1! We can now complete our final function and also operator.

```
multiply :: Complex -> Complex -> Complex
multiply (Complex (r1,i1)) (Complex (r2,i2))
  = Complex (r1*r2 - i1*i2, r1*i2 + r2*i1)
```

This code may look a bit overwhelming, but it's quite simple, really. We're just inserting the

rule from basic algebra above, except with the final i^2 expression replaced with a negation, in accordance with the definition of i.

TODO: Mention that this was a fairly shallow dsl. Write a little about how using polar coordinates would make some stuff easier and some harder, show how by giving alternative definitions of (some of?) the functions. Maybe implement it as a Num instance?

As noted at the very top, complex numbers are an entire field of study. If we were to cover all there is to know about them, this section would take up a small library worth of text and require an actual budget to write. The above should be just enough to give a basic understanding of the concept, along with all operators used in the sections below. Just remember that the coordinate comparison, while undeniably useful, does not apply to every possible operation and you'll do fine.

2.2 Integrals

Say we've got a graph. This graph describes a line. This line separates the graph in two sections; one above and one below the line.

How much space exists below the line?

2.2.1 Setting up the Language

A problem like the one described above is normally solved with an integral. To solve an integral, we'll first need a language that can describe a line.

TODO: Graph Pic Here!

Unlike complex numbers, there is no one uniform expression that can describe any line. Thus, this time, we'll have to invent something a little bit more flexible.

```
data Funk = X
    | Constant Integer
    | Add Funk Funk
    | Multiply Funk Funk
deriving (Show)
```

We'll do something a little more standard this time. The above DSL mimics the standard syntax for functions, or at least as closely as we can make it. Our "words" are the variable X, a constant, addition, and multiplication.

These are theoretically used in the exact same way that they would be on paper; the first two are values, while the second two are operators. For simplicity's sake, we'll be using an integer to describe our constants. X is, as you'd expect, an undefined variable.

Our operators; addition and multiplication, are a little more complicated. Rather than being simple values, they are statements that take other "words" and tie them together. Perhaps this is best explained by an example. Say you want the function "$5 + x^2$". In our DSL, we'd write it as

```
Funk test = Add (Constant 5) (Multiply X X)
```

The Multiply word, in this case, ties two X's together, while the Add word ties the result to the constant. Thus, we've created a "sentence" of words(or a function, according to math).

This is still a very limited language. At some point, we'll probably want more words to describe any other operations we might want to be part of our calculations; powers, divisions, roots, etc.

The list is effectively endless, and so to keep things comprehensive, let's stick to this simplified version for now.

HejHopp We're still not ready for integrals, however. First, we'll also need a function that takes the sentences we just created plus a value for x and produces a result. In practice, this is simplicity itself; we'll simply replace each word with their algebraic equivalent and let Haskell roleplay as a calculator for a moment. To wit:

```
Calculate :: Funk -> Integer -> Integer
Calculate
  | X value                    = value
  | (Constant int) value       = int
  | Add funk1 funk2 value      = (Calculate funk1 value) + (Calculate funk2 value)
  | Multiply funk1 funk2 value = (Calculate funk1 value) * (Calculate funk2 value)
```

Exercise 2. Expand the DSL with a word for subtraction, and full functionality in the 'Calculate' operator.

2.2.2 Brute Force Mathematics

Right, with our basic language defined, it's time to take a first stab at that graph.

The traditional first step to solving an integral is to, instead of look for an entirely accurate answer, instead look for an estimated answer.

To wit, let's use our test function from above ($5 + x^2$), and say we want to know how much space exists beneath the line and between x=2 and x=5. To do this, we could take the average of these numbers, (2+5)/2=3.5, and see how much space would exist between the line if this average was true throughout the function.

```
BFIntegral :: Funk -> Integer -> Integer -> Integer
BFIntegral funk start stop = (Calculate funk ((start+stop)/2)) * stop-start
```

There, quick and easy. Sadly not accurate. Our main problem is that our test function grows exponentially. This means that taking a value "in the middle" of the graph and expecting the values on either side to balance each other out is foolish.

Rather than despair, however, let's try to make our integral a little more accurate. The way we go about this is that rather than use an average for the entire graph, we cut the problem equally sized into chunks, calculate each chunk individually, and add the results together. TODO: Picture of basic integral.

```
BFIntegral' :: Funk -> Integer -> Integer -> Integer
BFIntegral' funk end end    = 0
BFIntegral' funk start stop = (Calculate funk start) + (BFIntegral' funk start+1 stop)
```

We made each chunk precisely 1 wide for simplicity, using a simple recursive function. The result should be considerably more accurate now, though still not perfect.

We also learnt an important lesson. See, the first example used chunks as well. Or rather, a single chunk. By cutting the chunk into smaller chunks, we increased accuracy. This is something we can repeat. In theory, we can repeat this infinitely; cutting our problem into an infinite number of infinitely small chunks. If each size reduction on the part of the chunks brings us closer to the true

answer, and we reduce the size infinitely, then we will become infinitely close to the true answer. I.E. We will have the true answer.

In practice, no computer can actually calculate infinity, but they can get close enough for most purposes. So let's give that a shot and see what happens; let's make one chunk for every 1/100.

```
BFIntegral'' :: Funk -> Double -> Double -> Double
BFIntegral'' funk end end    = 0
BFIntegral'' funk start stop = (Calculate funk start)*0,01 + (BFIntegral'' funk start+0,01 stop)
```

If this is still not enough.. well, there's still one subsection left to read. Beware though, it might be math heavy.

2.2.3 An Elegant yet Primitive Solution

TODO convert to primitive function solution goes here

Exercise 3. Extend support for subtraction to this form of integral.

Exercise 4. Expand the DSL with a word for a complex constant, plus associated functionality for both types of integral.

3 Basic control theory

Control theory studies continuous systems and the control of these. A easy example of a control system, with human input, is when you take a shower. Your body sends signals to the brain when its to hot or cold and you adjust the temperature with your hand. In this case the input is the signal your body sends to your brain and the output is the water temperature. This is a controls system with human input. In control theory we want to automate systems. A example of a common automated control system is the temperature system in your house. A thermostat measures the outside and inside temperature and control the heat elements in you house. This is a control system with two inputs.

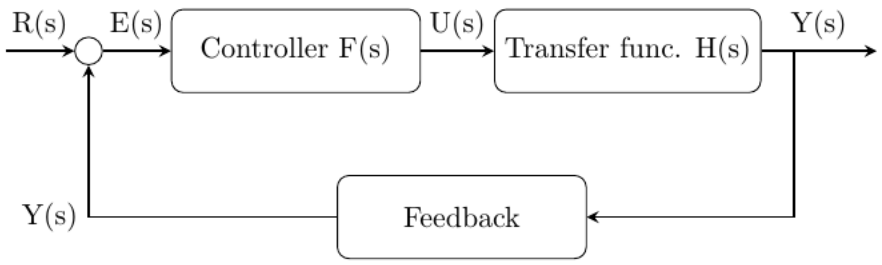


Figure 8: Feedback-loop

Control systems are created with the combination of measuring a magnitude and a controller. The magnitude measurements and controller create the system input u(t). A system can also utilize the output from the control. This is possible with a feedback loop and is a very common practice in control theory. The quest for all control systems is to create a stable system without delay or overshoot. (Ska lägga till närmare förklaring för dessa begrepp) In the course ERE103 this is fulfilled by combining tree different parameters (P, I, D) to create different controllers f(t). P is just a constant and can decrease the delay of the system. P creates the most basic controller f(t) = P. The I parameter is the integrating part of the controller and is used to remove the stationary fault. The D parameter is used to avoid overshoot. These parameters can be combined to fulfill

you specifications. The most common combination of these parameters in this course are PD, PI and PID. A controller is just a function and can therefore easily be modeled in Haskell.

Control systems are often modelled in block diagrams as you can see above. When we analyze control systems we often transform the functions from the time domain to the frequency domain with the laplace transform. The laplace transform is very useful and will be explained more in-depth later.

In control systems we want our systems to be stable. The results of unstable systems can be horrific so we want to avoid this at all cost. We can use a few methods to ensure this like Ruth-Hurwitz and the Nyqvist Theorem.

4 Laplace transform

```haskell
module LaplaceTransform where
    import Complex
```

The Laplace transform is a tool that enables us to look at a function or equation from a different perspective. More specifically, it takes a function of time and transforms it into a function of frequency. The Laplace transform is often used to solve differential equations, but more on that later.

4.1 Conventions and notations

First, some conventions: it is common to call the Laplace transform of a function by the capital version of the original function, e.g. the Laplace transform of f is F, not to be confused with the primitive function of f. Unless otherwise specified, from here on F will signify the Laplace transform of f. Although we won't use it, some texts write $\widetilde{f}$ for the Laplace transform of f. Note also that we use "curly brackets" ({ and }) around the function. Some texts use ordinary parentheses, but we will stick to curly brackets to avoid ambiguities.

TODO:

- `laplace` (or maybe `L`?) is the Haskell version of $\mathscr{L}$
- will either write functions without parameter (i.e. f is the function, not $f(t)$), or in specific instances like anonymous function (e.g. $t \mapsto e^{at}$). Alone $f(t)$ means f applied to t.

4.2 Types

The definition of the Laplace transform is

$$\mathscr{L}\{f\}(s) = \int_0^\infty e^{-st} f(t)\,\mathrm{d}t.$$

What are the types? An initial inspection allows us to identify

```haskell
t   :: R                    -- t for time
f   :: R       -> R         -- f for function
exp :: Complex -> Complex   -- the exponential function
```

and from DSLsofmath we know that

Add citation, probably around page 60

```haskell
integ :: R -> R -> (R -> R) -> R -- definite integral
```

i.e. the integral takes two real arguments (the limits) and a function, returning a real number. Letting s be a real number as well works fine, but in fact it's possible to be a bit more general and let it be a complex number.

From this we can read that the Laplace transform should have the type

```haskell
laplace :: (R -> R) -> (Complex -> Complex)
```

i.e. it's something that takes a function and transforms it into another function. Note that the function `laplace` will be our haskell representation of $\mathscr{L}$ above.

4.3 The most common rules and their usage

TODO: add all we want to table, preferably so it's similar to the one they're given in the course.

Table 1: Some common Laplace Transforms and rules.

f(t)	F(s)
$af(t) + bg(t)$	$aF(s) + bG(t)$
$f'(t)$	$sF(s) - f(0)$
$e^{\alpha t}$	$\frac{1}{s-\alpha}$

The definition of the Laplace transform is clunky to actually work with, so common practise is to have a table of common functions and their Laplace transforms, and use some rules to calculate the harder problems. We will use this tabular approach. For example, we might be asked to find the Laplace transform of

$$f(t) = e^{\alpha t}.$$

If we look in table 1, we can see that the corresponding Laplace transform is

$$\mathscr{L}\left\{t \mapsto e^{\alpha t}\right\} = s \mapsto \frac{1}{s-\alpha}.$$

If we wanted to solve this using the definition, we would have to solve the integral

$$F = s \mapsto \int_0^\infty e^{-st} e^{-\alpha t} \, \mathrm{d}t.$$

While doable, it's definitely easier to solve this using the tabular approach.

4.3.1 Superposition

Write something about higher order types of addition etc., maybe?

Superposition is also called linearity; the Laplace transform is linear. This means that if you have a sum that you want to transform, you can just transform the terms separately and then sum them. Mathematically this is written (if f and g are two functions to be transformed, and α and β are two real numbers):

$$\mathscr{L}\{\alpha f + \beta g\} = \alpha \mathscr{L}\{f\} + \beta \mathscr{L}\{g\}$$

In Haskell, this could look like

```haskell
laplace (a * f + b * g) = \s -> a * (F s) + b * (G s)
                where F = laplace f
                      G = laplace g
```

i.e. we find the laplace transform of the functions, scale the results and finally sum them.

Example 1. Use the rule

$$\mathscr{L}\{t \mapsto e^{at}\} = s \mapsto \frac{1}{s-a}$$

to Laplace transform

$$f = t \mapsto (3e^{-t} + 5e^{-3t})$$

or, written in Haskell-style

```haskell
    laplace (\t -> 3 * exp (-t) + 5 * exp ((-3) * t))
```

Solution.

4.3.2 Derivative rule

Sometimes we want to find the Laplace transform of the derivative of a function. This most often occurs when trying to solve differential equations. The typical way this is written in math texts is

$$\mathscr{L}\{f'\}(s) = s\mathscr{L}\{f\}(s) - f(0).$$

If we assume that we have `D`, a haskell implementation of derivative, e.g. `D sin = cos`. Then this rule can be written

```haskell
laplace (D f) s = s * (laplace f s) - f 0
```

Example 2. Use the Laplace transform to find f satisfying

$$f' = -f \qquad f(0) = 1$$

Solution. What we want to do is apply the Laplace transform to both sides of the equation, and then use the resulting equation to find an explicit expression for $F(s)$.

First, we apply the Laplace transform to the left hand side, which gives us

```haskell
laplace f' = \s -> s * laplace f s - f 0
```

Then we do the same thing to the right hand side.

```haskell
laplace (-f) = \s -> - (laplace f s)
```

if we let `F = laplace f` and write these two sides as equal to each other:

I'm still not sure what to do here. Let it be like this? Do it mathy anyways? Find a solution with

```haskell
s * F s - 1 = - (F s)
```

if we move all instances of F s to one side and everything else to one side we get

```
s * F s + F s = 1.
```

Factor out F s:

```
F s * (s + 1) = 1
```

and divide by (s + 1):

```
F s = 1 / (s + 1)
```

or in mathematical writing:

$$F = s \mapsto \frac{1}{s+1}$$

Now, if we look at the table of Laplace transforms, we can find that this is the transform of $t \mapsto e^{-t}$, so we can assume that the answer is $f = t \mapsto e^{-t}$ (and this is in fact correct!).

Although we are done with the exercise now, a good habit to get into early is to always double check if the function satisfies the differential equation. If we differentiate f we get

$$f' = t \mapsto -e^{-t} = -f.$$

Thus we know that one of the conditions holds. Inserting $t = 0$ into f gives us

$$f(0) = e^{-0} = 1,$$

so the second condition holds as well. Now we know that $f = t \mapsto e^{-t}$ really is a solution to the differential equation.

Checking if the function satisfied the equation might seem excessive, but when solving slightly more complicated problems one often have to use partial fraction decomposition, in which it's very easy to make mistakes. Checking if the solution matches the equation is a simple way to see if the result is correct.

4.3.3 Integral rule

Now one might wonder: if there is a rule for derivatives, is there one for integrals? Luckily, such a rule exists!

$$\mathscr{L}\left\{t \mapsto \int_0^t f(x)\,\mathrm{d}x\right\}(s) = \frac{\mathscr{L}\{f\}(s)}{s}$$

Note that the integral

$$\int_0^t f(x)\,\mathrm{d}x = F(t) - F(0)$$

(where F is the primitive function of f, not the laplace transform), i.e. this integral functions more like an indefinite integral than a definite one.

It's interesting to note how the Laplace transform turns the derivatives and integrals, which are each others inverses, into multiplications and divisions, which also are each others inverses.

If we assume that we have a Haskell command

```
integ :: R -> R -> (R -> R) -> R
-- the integral from a to b of the function f is written integ a b f
```

which calculates the definite integral of a function. Then the rule can be written

```
laplace (\t -> integ 0 t f) = \s -> (laplace f s) / s
```

4.3.4 Convolution

For combining two functions describing LTI systems together, we use convolution. (More on this later, in 5.4). For two functions f and g defined on $[0, \infty)$, convolution is defined as

$$(f \circledast g)(t) = \int_0^t f(\tau) g(t - \tau) \, \mathrm{d}\tau$$

Normally convolution is denoted with an asterisk, $*$, but since this coincides with the Haskell symbol for multiplication we will use $\circledast$ instead.

This might be implemented

```
conv f g t = integ (\tau -> (f tau) * (g (t - tau))) 0 t
```

The Laplace transform of convolution is

$$\mathscr{L}\{f \circledast g\} = \mathscr{L}\{f\} \cdot \mathscr{L}\{g\},$$

i.e. the laplace transform turns convolution into multiplication.

```
laplace (conv f g) = laplace f * laplace g
```

TODO: Make sure that we explain what $\cdot$ is, either here or earlier.

4.3.5 Time

Another rule that often shows up handles time shift in a function, i.e. what does the function look like if you start it after τ time. Mathematically, this is written

$$\mathscr{L}\{t \mapsto f(t - \tau)\}(s) = s \mapsto e^{-sT} \mathscr{L}\{f\}(s).$$

If we have a function

```
shift f tau = \t -> f (t - tau)
```

this could be written

```
laplace (shift f tau) = s -> exp ((-s) * tau) * (laplace f s)
```

TODO: Something about when it's used?

4.3.6 Exponential decay (Frequency shift)

Models damping of a system. If $f(t)$ is a function that models a physical system, then $e^{-\alpha t} f(t)$ is the damped version of that function. The Laplace transform of a damped function is given by

$$\mathscr{L}\left\{t \mapsto e^{-\alpha t} f(t)\right\}(s) = \mathscr{L}\{f\}(s + \alpha)$$

```haskell
laplace (\t -> exp ((-a)*t) * f t) = \s -> laplace f (s+a)
```

Note that there is a kind of symmetry here with time shift, only in this case it's on the other side. Since there's a shift in the frequency instead of time, this could be called the "Frequency shift"-rule.

4.4 Inverse Laplace transform

So far, whenever we've found an expression for the Laplace transform of a function, we've hand-waved it by saying "and we recognize that this other function transforms into this function, thus it should be the answer". This gives the right answer (obviously, or else we wouldn't teach you that...), but it's not entirely rigorous (and mathematicians love rigor!)

The "proper" way to do it is by using what's (appropriately) called the inverse Laplace transform.

Like the regular Laplace transform, the inverse Laplace transform is defined using an integral. This one is even more clunky than the ordinary Laplace transform, however. The definition goes that the inverse Laplace transform of a function F, denoted $\mathscr{L}^{-1}\{F\}$, is given by

$$\mathscr{L}^{-1}\{F\}\, s = \frac{1}{2\pi} \lim_{T\to\infty} \int_{\gamma-iT}^{\gamma+iT} e^{st} F(s) \mathrm{d}s.$$

Now, your reaction here might be: what's this monstrosity? An integral where both endpoints are complex numbers, and a limit outside of that? Luckily, we don't have to calculate it[1], since we can just use the rules to find the inverse instead!

5 Transfer Functions and LTI Systems

```haskell
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE StandaloneDeriving #-}

module LTIandTF where
  import Lib
```

A transfer function is a function describing how a linear time-invariant (LTI) system (or part of such a system) transforms a time-varying input signal to give the system output.A transfer function is stated in the (complex) frequency domain (i.e. after using the Laplace transform). Thus—from the naming convention from the Laplace transform—a function f describing an LTI system has a corresponding transfer function labelled F.

In order to understand transfer functions well, we need to first gain some insight into LTI systems:

5.1 Linear Time-Invariant Systems

A linear time-invariant (LTI) system is a system that is only indirectly dependent on time (t). More precisely, input signals may be linearly combined (i.e. superposed) or differentiated/integrated with respect to time. As a consequence, an LTI system reacts exactly the same regardless of what value t has, given identical inputs. (As a real-life parallel: the shower will change the water

[1] However, if you want to understand how to calculate it, we recommend:

Add recommendation

temperature in the same way when turning the valve, regardless of whether you do it in the morning or in the afternoon, today or tomorrow). We can express this property in code in the following way:

```
sys :: LTI; d :: Time; f :: Time -> Amplitude
shift :: Time -> (Time -> Amplitude) -> (Time -> Amplitude)
shift d f = \t -> f (t - d)

sys (shift d f) == shift d (sys f)
```

In this code a signal is represented by the type `Time -> Amplitude`, and `shift d f` shifts a signal `f` with some time `d`. `sys` represents an LTI system. We will explore its type in a bit. We can then express the property of linearity for LTI systems in code like this:

```
a, b :: Double; f, g :: Time -> Signal
(+) :: (a -> b) -> (a -> b) -> (a -> b)
f + g = \x -> f x Prelude.+ g x
-- Scales the signal (amplitude) with
scale :: Double -> (Time -> Signal) -> (Time -> Signal)
scale k f = \t -> k * f t

sys (scale a f + scale b g)
    == scale a (sys f) + scale b (sys g)
```

In the code above, `scale k f` scales the signal amplitude of f with k.

These systems are described with transfer functions and what we call LTI-functions (i.e. the inverse Laplace transform of the transfer function), but in an unusual way, which turns out to make the math easier, but tends to impede understanding. Let's take a closer look at the types involved to understand this:

5.2 Types

Let's start with the LTI system itself. It takes an input signal ($u(t)$), transforms it, and yields an output signal ($y(t)$). The first thing to take notice of is what type the in- and output signals have:

$$Time = \mathbb{R}^{>0},\ Signal = \mathbb{R}$$
$$u, y : Time \to Signal$$

The LTI system, *lti*, transforms its input signal in full. In other words, *lti* is a second-order function—mapping a (single-parameter) function to another such function:

$$lti : (Time \to Signal) \to (Time \to Signal)$$

The LTI system needs the entirety of its input function—and not just the momentary value at t—in order to transform it into the output function. It turns out that we can describe this transformation with another function of the same type as the in- and output functions. This is what we call the LTI-function, whose Laplace transformation is the system's transfer function.

Because of this roundabout way of describing this system, we cannot easily link two LTI systems together in an intuitive manner. Sure, we can define a system to be composed of two other LTI systems, like the following:

```
lti = lti1 . lti2
```

This is works well since *lti* is an endofunction (i.e. a function of type $a \to a$), but we do generally not have an expression for this transformation. What we do have, however, is the LTI-function

(more on how it is defined in the next section; 5.3). The equivalent to composing LTI systems when working with LTI-functions (and in-/outputs to such systems since they have the same type) is convolution, which we saw (in 4.3.4) simplifies to regular multiplication when Laplace-transformed. With an input function $u(t)$, an LTI system `sys` with LTI-function $f(t)$, and an output function $y(t)$, we have the following relations:

```
lti u == y
U = laplace u; Y = laplace y; F = laplace f
Y s == \s -> U s * F s
y t == \t -> u t |*| f t
```

Consider the LTI system shown in the following block diagram:

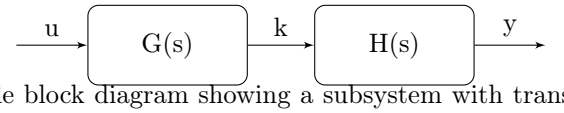


Figure 9: A simple block diagram showing a subsystem with transfer function G(s) feeding into a subsystem with transfer function H(s). u(t) and y(t) are the in- and output signals, respectively. k(t) is the input of the H-system and output of the G-system.

The functions U(s), G(s), K(s), H(s), and Y(s) are transfer functions derived from the LTI-functions u(t), g(t), k(t), h(t), and y(t). The LTI-functions are functions of time that yield a signal strength, so we label their types:

$$Time = \mathbb{R}^{\geq 0},\ Signal = \mathbb{R}$$
$$u, g, k, h, y : Time \to Signal$$

```
type Time   = Real
type Signal = Real
u, g, k, h, y :: Time -> Signal
```

This determines the type of their transfer functions. Recall, $F(s) = \mathscr{L}\{f\}(s)$ and the type of the Laplace transform is $\mathscr{L} : (\mathbb{R}^{\geq 0} \to \mathbb{R}) \to (\mathbb{C} \to \mathbb{C})$, making $F(s) : \mathbb{C} \to \mathbb{C}$. Thus,

$$U, G, K, H, Y : \mathbb{C} \to \mathbb{C}$$

Note that even though we call all these functions transfer functions (and they are all of the same type), they do not all serve the same purpose. G(s) is the transfer function for the system represented by the left block in the block diagram and H(s) is the transfer function for the right one. U(s) is a transfer function representing the input signal. We can imagine this to be the transfer function of a system with $\delta(t)$ as input, feeding into the G-system (see figure 10). We here think of δ as a function that actuates the system. k(t) is an intermediate function sometimes written out to give a variable to multiple systems (this can be useful in more complex systems)—in this case $K(s) = U(s) \cdot G(s)$, i.e. the combined system of the input signal and the G-system. Y(s) is the transfer function for the full system *and* U(s). The transfer function for the entire system is therefore $Y(s)/U(s) = G(s) \cdot H(s)$.

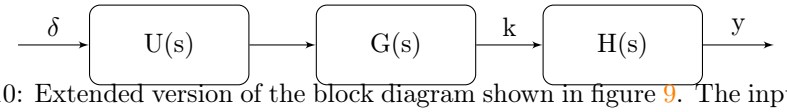


Figure 10: Extended version of the block diagram shown in figure 9. The input signal u(t) has been replaced by its transfer function with δ as input, acting as an actuator for the system. This is identical to the previous representation, but highlights how the input can be shown as a system with a transfer function U.

Let's summarise this with some relations:

```
type LTI = (Time -> Signal) -> (Time -> Signal)
sys :: LTI
f :: Time -> Signal; F :: C -> C

sys = (|*| f)
```

Or equivalently:

```
sys = \sig -> invlaplace $ laplace sig * F
```

where `sys` has LTI-function f and transfer function F.

5.3 Linear Time-Invariant Systems – Part 2

As mentioned above (in 5.1), a linear time-invariant (LTI) system is a system that is only indirectly dependent on time. An interesting property, stemming from this time invariance and the linear property, is that any sine function input into the system would still be a sine function, whereas any other type of wave function would change shape (not counting translation and scaling). We let the following represent a signal processed by an LTI system:

```haskell
data Signal a where
  -- Amp -> Freq -> Time-shift -> (a -> a)
  Sin   :: Num a => a -> a -> a -> Signal a
  Sum   :: Signal a -> Signal a -> Signal a
  Scale :: a -> Signal a -> Signal a
  Deriv :: Signal a -> Signal a
  Integ :: Signal a -> Signal a
deriving instance Eq a => Eq (Signal a)
```

It should now be possible to take any Signal instance and simplify it to a single Sin constructor. Or, at least as long as frequencies match. But this is okay since an LTI system cannot change the frequency of a sine function.

```haskell
simplifySignal :: (Num a, Eq a, Floating a, Ord a) => Signal a -> Signal a
simplifySignal (Scale k (Sin a f s)) = Sin (a * k) f s
```

Recall how the derivative of $sin(t)$ is $cos(t)$, and that its antiderivative is $-cos(t)$, as well as the relationship $cos(t) = sin(\pi/2 - t)$, in order to find the Sin constructor form of the following:

```haskell
simplifySignal (Deriv   (Sin a f s)) = Sin (a * f) (-f) (pi/2 - s)
simplifySignal (Integ   (Sin a f s)) = Sin (-a/ f) (-f) (pi/2 - s)
```

Summing two sine functions isn't quite as neat (no need to pay attention to the details here). And as mentioned, we disallow differing frequencies:

```haskell
simplifySignal (Sum (Sin a1 f1 s1)
                    (Sin a2 f2 s2)) | f1 == f2 = Sin a f1 s where
                      a = sqrt $ q1 ^ 2 + q2 ^ 2
                      s = atan $ q2 / q1
                      q1 = a1 * cos s1 + a2 * cos s2
                      q2 = a1 * sin s1 + a2 * sin s2
```

Since we now have this line which is not guaranteed to evaluate to a Sin constructor (because f1 might not equal f2) and in order to propagate the function `simplifySignal` over the syntax tree, we include the following to the function definition:

```
27 simplifySignal other = let ss = simplifySignal in case other of
28   Scale k (Sum o1 o2)              -> Sum (ss $ Scale k o1) (ss $ Scale k o2)
29   Deriv   (Sum o1 o2)              -> Sum (ss $ Deriv   o1) (ss $ Deriv   o2)
30   Integ   (Sum o1 o2)              -> Sum (ss $ Integ   o1) (ss $ Integ   o2)
31   Scale k a                        -> ss $ Scale k $ ss a
32   Deriv   a                        -> ss $ Deriv   $ ss a
33   Integ   a                        -> ss $ Integ   $ ss a
34   Sum a b | ss a /= a || ss b /= b -> ss $ Sum (ss a) (ss b)
35   Sum a@(Sum a1 a2) b@(Sum _ _)    -> ss $ Sum (ss $ Sum a1 b) (ss a2)
36   Sum a b | maxfreq a < maxfreq b  -> ss $ Sum (ss b) (ss a)
37   Sum (Sum c a@(Sin _ af _)) b@(Sin _ bf _)
38                        | af == bf -> ss $ Sum (ss c) (ss $ Sum a b)
39   _                               -> other
```

Code 1: The function `maxfreq` returns the maximum frequency in the expression tree. See Appendix C.2 for details.

Apart from the caveat with frequency, we can quite easily see how it is that sine functions retain their shape in an LTI system. We could exclude frequency from the definition of the data type in order to make things easier, but that would at the same time make it less expressive.

We could explore this with laplace (or fourier would suffice) to cast perhaps better light onto this fact. Fourier of a sum of sines of equal freq would yield one peak.

Lastly, let's define the semantics of the data type:

```
40 evalSignal :: (Num a, Floating a, Eq a, Ord a) => Signal a -> (a -> a)
41 evalSignal     (Sin a f s) t = a * sin (f * t + s)
42 evalSignal     (Sum   a b) t = evalSignal a t + evalSignal b t
43 evalSignal     (Scale a b) t = a * evalSignal b t
```

Since we cannot find a derivative only from `evalSignal a` and `t`, we will have to rely on `simplifySignal` to evaluate `Deriv` and `Integ`:

```
44 evalSignal sig@(Deriv a)   t = evalSignal (simplifySignal sig) t
45 evalSignal sig@(Integ a)   t = evalSignal (simplifySignal sig) t
```

5.4 Combining Transfer Functions

Since the LTI-functions are not endofunctions (i.e. are not functions *on* a set), they cannot be composed. So how do we combine several transfer functions in a larger system? To understand that, we first have to understand exactly what a transfer function is describing:

The LTI system is by definition time-invariant, so functions are not directly expressed in time, though, they vary with time. They are not expressed as a function of their input signal (but rather with their time-varying LTI-function), so how does it vary with input signal? The basic idea is that its LTI-function, $f(t)$, expresses the "impulse response" of the system, i.e. the output signal given an instantaneous impulse as input (the dirac delta function δ). Any other input to the system has to be convoluted with the system function (defined in 4.3.4. The LTI-function is defined this way because δ is the identity of the convolution operator—in other words:

```
(dirac |*|) == (id :: (Time -> Signal) -> (Time -> Signal))
```

Luckily, the Laplace transform of two convoluted LTI-functions is simply the product of their transfer functions.

```
\f -> laplace (\t -> diracdelta t |*| f t)
\f -> laplace (\t -> diracdelta t) * laplace (\t -> f t)
\f -> (\t -> step t) * (\t -> F t)
{- step = const 1 when t>=0 -}
\f -> \t -> F t
```

Because this "impulse response" is the same thing as what we've been calling "LTI-function," this is what the function is usually referred to as. But this name suggests that it is merely a special case among input responses, when, in fact, it is essential for defining LTI systems. We will henceforth use the commonly accepted term "impulse response," but do also think of the function as the function that specifies how an LTI system transforms its input.

5.5 Finding Transfer Functions

An LTI system is often described with a differential equation. In this section we will take a closer look at how to find a transfer function for an LTI system, given a differential equation.

We will use the following differential equation as a running example (as per usual, u is the input signal and y is the output signal):

$$9y(t) + 6\dot{y}(t) + \ddot{y}(t) = 3u(t) + 2\dot{u}(t)$$

We give the following representation for a linear differential equation with constant coefficients:

```
46 data LCDE a where
47   LCDE :: ([a],[a]) -> LCDE a
48
49 lcde = LCDE ([9,6,1], [3,2]) :: Fractional a => LCDE a
```

The type consists of two lists, representing each side of the equation. The leftmost (first) digit in each list represents the coefficient of the lowest order derivative. No constant term is included (since this would make both the math and the syntax more complicated).

$$\text{LCDE}([a_0, a_1, ..., a_k], [b_0, b_1, ..., b_l]) \iff \sum_{n=0}^{k} a_n y^{(n)}(t) = \sum_{n=0}^{l} b_n u^{(n)}(t)$$

We provide a definition of the Laplace transform (introduced in the previous section), specific to this set of equations. For the purposes of the demonstration of transfer functions an exact definition is not essential, but is provided below in code snippet 2.

In a similar manner to the above type we use a list definition, but a third list is needed to represent additional constant–s^n-terms. In order to calculate this last list's coefficients, values of $y^{(n)}(0)$ and $u^{(n)}(0)$ are needed (notice how `laplace` in code snippet 2 requires these values). Specifically, c_n below (equation 1) is a sum of $a_i y^{(v)}(0) - b_j u^{(w)}$. (We are going to simplify this in a bit).

$$\begin{aligned} &\text{Transformed}([a_0, a_1, ..., a_k], [b_0, b_1, ..., b_l]; [c_0, c_1, ..., c_m]) \iff \\ &\iff \sum_{n=1}^{k} a_n s^{n-1} Y(s) = \sum_{n=1}^{l} b_n s^{n-1} U(s) + C, \\ &C = \sum_{n=0}^{m} c_n s^n \end{aligned} \tag{1}$$

```haskell
data Transformed a where
  Transformed :: ([a],[a]) -> [a] -> Transformed a

--                  equation  f^i(0) g^j(0)   L{equation}
laplace :: Num a => LCDE a -> [a] -> [a] -> Transformed a
laplace (LCDE (lhs, rhs)) f0 g0 = transformed where
  transformed = Transformed (lhs, rhs) (zipWithL 0 (-) lc rc)
  lc = map sum [zipWith (*) f0 (drop n lhs) | n <- [1..length lhs-1]]
  rc = map sum [zipWith (*) g0 (drop n rhs) | n <- [1..length rhs-1]]
```

Code 2: A data type representing the equation after applying the Laplace transform to its left and right hand sides; A function definition for the Laplace transform of an instance of LCDE. The function `zipWidthL` referenced works like `zipWith` but extends the shorter list with the first parameter such that the output has the same length as the longer of the two lists. See appendix C.1 for details.

Albeit a bit convoluted, we now have a representation of LCDE when it is laplace transformed. Notice how LCDE (lhs, rhs) $\mapsto$ Transformed (lhs, rhs) _. The hole is a polynomial in s, $p(s)$, that gets evaluated with values of $y^{(n)}(0)$ and $u^{(n)}(0)$. We make the assumption that all of these values are constant zero, $f^{(n)}(0) = 0$, since we can assume $u(t)$ is zero $\forall t \leq 0$, and thus we can drop these two extra lists of values and re-define the data type as follows:

```haskell
data Transformed a where
  Transformed :: ([a],[a]) -> Transformed a deriving Eq

laplace :: Num a => LCDE a -> Transformed a
laplace (LCDE (lhs, rhs)) = Transformed (lhs, rhs)
```

The values don't change here—only the semantic information. For our running example, lcde defined above, would evaluate to the following:

```haskell
lcdeTrans1 = laplace lcde
lcdeTrans2 = Transformed ([9,6,1], [3,2])
truth = lcdeTrans1 == lcdeTrans2
```

We expect our linear differential equation with constant coefficients to result in a rational expression, so let's define a data type for rational expressions.

```haskell
data RatExpr a where
  RatExpr :: Num a => ([a],[a]) -> RatExpr a
```

We find the rational expression simply by solving for G(s)=Y(s)/U(s):

$$\sum_{n=0}^{k} a_n s^n Y(s) = \sum_{n=0}^{l} b_n s^n U(s) \tag{2}$$

$$Y(s) \sum_{n=0}^{k} a_n s^n = U(s) \sum_{n=0}^{l} b_n s^n \tag{3}$$

$$Y(s)/U(s) = \sum_{n=0}^{l} b_n s^n / \sum_{n=0}^{k} a_n s^n \tag{4}$$

```haskell
solve2RatExpr :: Num a => Transformed a -> RatExpr a
solve2RatExpr (Transformed (lhs, rhs)) = RatExpr (lhs, rhs)
```

Again, this is in the end just a change of constructor name, i.e. changing the semantics. Combining the two we see nothing new:

```haskell
findTF :: Num a => LCDE a -> RatExpr a
findTF = solve2RatExpr.laplace
```

To make sense of this construction, we can make a simple eval-function:

```haskell
evalRE :: Fractional a => RatExpr a -> a -> a
evalRE (RatExpr (den,num)) s = numerator / denominator where
  numerator   = unravel num
  denominator = unravel den
  unravel []   = 0
  unravel list = head list + s * unravel (tail list)

tfG s = evalRE (findTF lcde) s
```

The function tfG above is the transfer function from the differential equation lcde.

We can use this for any arbitrary differential equation of the same form. Consider $3\dot{y}(t) + 2y(t) = \ddot{u}(t) - u(t)$, and the following code snippet:

```haskell
de = LCDE ([2,3],[-1,0,1]) :: Fractional a => LCDE a
tf2 = evalRE $ findTF de
```

5.6 Step Response

necessary?

6 Nyquist

To explain the Nyqvist theorem of stability we must start with Cauchy's argument principle. This principle is used to find the difference between the number of poles and zeroes inside some closed contour.

6.1 Cauchy's argument principle

According Chauchy's we start by mapping each value on our contour to our function and create a new plot (works exactly the same as Haskell's map). This new plot will, among other properties, show us the difference between poles and zeroes. Depending on how many times the plot surround the origin, and in which direction, we can get information about the poles and zeroes. Let me show you with a picture.

Inside our contour we got four zeroes and tree poles. After we map all the values in the contour the plot in the w-plane is created. The plot surrounds the origin one time clockwise. This tells us that we got one more zero than pole inside our contour. If the plot was counterclockwise we would have had more poles than zeroes instead.

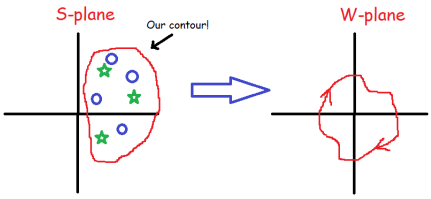


Figure 11: Mapping all of our values to create the plot

6.2 Nyqvist theorem

Nyqvist theorem will use this principle to find out if a system i stable or not. The Nyqvist theorem is graphical technique and will result in a Nyqvist plot. In the case of the a closed loop system the transfer function is defined as

$$F(s)\ (1 + F(s)H(s))$$

That means if our denominator is equal to zero our system is unstable. That would be bad news!

To solve this we want to find out if we got any zeros in our right half plane. Therefore we surround our entire right half plane with a contour with infinite radius. We could map all these values to our denominator

$$1 + F(s)H(s)$$

but instead we are going to map our values to

$$F(s)H(s)$$

to make our lives easier. The will result of this will is the Nyqvist plot. Since we removed our +1 we a going to count the encirclements around -1 instead of the origin. In the picture you can take a look at this.

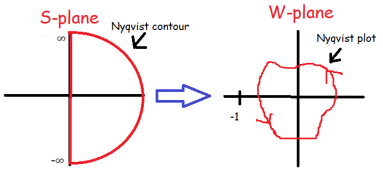


Figure 12: Mapping all of our values from our Nyqvist contour to create the Nyqvist plot

Unfortunately this will only give us the difference between the number of zeroes and poles. To solve this will will utilise the equation $Z = N + P$. Where Z is the number of zeroes, N the number of clock wise encirclements and P is the number of poles in the open loop. In other words Nyqvist let us solve the closed-loop system with some help from the open-loop system.

This is how a Nyqvist plot is created and what i tell us about a transefer function. But since neither we or Haskell's map function can't compute with a infinite large input we have to find a way around this. We can instead focus on four important points and get a non-continuous plot. w=0, w=inf, w when we cross the real axis and w when we cross the imaginary axis. This will be enough for the most basic transfer functions you will see in the course.

A Table of Laplace transforms

B Analysis (/Calculus?)

Added some preliminary tables for analysis, maybe we wanna write something more?

B.1 Types

TODO:

- Type of derivative, integral
- Higher order functions? $(+, \cdot, \dots)$

B.2 Derivatives

Table 2: Derivatives

$f(x)$	$f'(x)$
x^a	ax^{a-1}
e^x	e^x
e^{kx}	ke^{kx}
a^x	$a^x \ln a$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\tan x$	$\frac{1}{\sin} x$
$\arcsin x$	$\frac{1}{\sqrt{1-x^2}}$
$\arccos x$	$-\frac{1}{\sqrt{1-x^2}}$
$\arctan x$	$\frac{1}{1+x^2}$

B.3 Partial derivatives

B.4 Integrals

Table 3: Integrals

$f(x)$	$F(x)$
x^a	$\frac{x^{a+1}}{a+1}$
e^x	e^x
e^{kx}	$\frac{e^{kx}}{k_x}$
a^x	$\frac{a^x}{\ln a}$
$\sin x$	$-\cos x$
$\cos x$	$\sin x$
$\tan x$	$-\ln\lvert\cos x\rvert$
$\arcsin x$	$x \arcsin x + \sqrt{1-x^2}$
$\arccos x$	$x \arccos x + \sqrt{1-x^2}$
$\arctan x$	$\frac{1}{2}\left[2x \arctan x - \ln(1+x^2)\right]$

C Code

C.1

```haskell
--       null-val zip-function   left   right  zipped
zipWithL :: a -> (a -> a -> b) -> [a] -> [a] -> [b]
zipWithL i f (l:left) (r:right) = f l r : zipWithL i f left right
zipWithL _ _ []       []        = []
zipWithL i f []       (r:right) = f i r : zipWithL i f []   right
zipWithL i f (l:left) []        = f l i : zipWithL i f left []
```

Code 3: Extension of `zipWith`. If lists are equal in length `zipWithL` and `zipWith` are identical. When one list has exhausted their elements, `i` is used instead. `zipWithL k f [a0,a1,a2] [b0,b1,b2,b3,b4]` becomes `[a0 `f` b0, a1 `f` b1, a2 `f` b2, i `f` b3, i `f` b4]`.

C.2

```haskell
maxfreq :: (Ord a, Num a) => Signal a -> a
maxfreq (Sin _ f _) = f
maxfreq (Scale _ a) = maxfreq a
maxfreq (Deriv   a) = maxfreq a
maxfreq (Integ   a) = maxfreq a
maxfreq (Sum a b) | af < bf   = bf
                  | otherwise = af where
                    af = maxfreq a
                    bf = maxfreq b
```

Code 4: `maxfreq` is a helper function on the `Signal` data type for the function `simplifySignal` (see code snippet 1). It returns the maximal frequency found in the given `Signal` expression tree defined by `Sum` constructors as tree nodes and `Sin` as leaves.

List of Figures

List of Tables