SOLVING HIGH-ORDER LINEAR ODES IN DRASIL

SOLVING HIGH-ORDER LINEAR ODES IN DRASIL

BY

DONG CHEN, M.Eng.

A REPORT
SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING

Master of Engineering (2022) McMaster University
(Department of Computing and Software) Hamilton, Ontario, Canada

TITLE:	Solving High-order Linear ODEs in Drasil
AUTHOR:	Dong Chen M.Eng. in (Systems Engineering), Boston University, Massachusetts, USA
SUPERVISOR:	Spencer Smith and Jacques Carette
NUMBER OF PAGES:	xiv, 67

Lay Abstract

A lay abstract of not more 150 words must be included explaining the key goals and contributions of the thesis in lay terms that is accessible to the general public.

Abstract

Abstract here (no more than 300 words)

Your Dedication

Optional second line

Acknowledgements

Acknowledgements go here.

Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Notation

$\mathbb{R}$	any real number in (-∞, ∞)
$\mathbb{R}^n$	an infinite sequence that contains real numbers, n is an infinite integer
$\mathbb{R}^k$	a finite sequence that contains real numbers, k is a finite integer
$\mathbb{R}^i$	a finite sequence that contains real numbers, i is the number of equation in the ODE
$\mathbb{R} \to \mathbb{R}^i$	a function takes a real number and outputs a sequence of real numbers

Definitions

Drasil Framework

Drasil Code Generator

Drasil Printer

Abbreviations

ODE	Ordinary differential equation
SRS	Software requirements specification
SCS	Scientific computing software
NoPCM	Solar water heating system without PCM
PDController	Proportional derivative controller
DblPendulum	Double pendulum
SglPendulum	Single Pendulum
IVP	Initial Value Problem
BVP	Boundary Value Problem
ACM	Apache Commons Maths
BDF	Differentiation Formula Method

RK	Runge-Kutta
GOOL	Generic Object-Oriented Language

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 0

Software Automation

From the Industrial Revolution (1760-1840) to the mass production of automobiles that we have today, human beings never lack innovation to improve the process. In the Industrial Revolution, we start to use machines to replace human labour. Today, we have been building assembly lines and robots in the automobile industry to reach a scale of massive production. Hardware automation has been relatively successful in the past one hundred years, and they have been producing mass products for people at a relatively low cost. With the success story of automating hardware, could software be the next one? Nowadays, the software is used every day in our daily life. Most software still requires a human being to write them. Programmers usually write software in a specific language and produce other byproducts during development time. Whether in an enterprise or research institution, manually creating software is prone to errors and is not as efficient as a code generator. In the long term, a stable code generator usually beats programmers in performance. They will eventually bring the cost down because of the labour cost reduction. Perhaps this is why human beings consistently seek to automate work. History demonstrates that we successfully automate manual processes in hardware. With fairly well-understood knowledge of software, creating a comprehensive system to produce software is not impossible. Can you imagine that programmers no longer programming in the future world? In the future world, code generators will generate software. There will be a role called "code alchemist" who is responsible to write the recipe for the code generator. The recipe will indicate what kind of software people want. In other words, the recipe is also a software requirement document that the code generator can understand. The recipe can exist in the form of a high-end programming language. Once the code generator receives the recipe, it will automatically produce software artifacts. The code generator exists in the form of a compiler. The described above is revolutionary if there is such a code generator, and the Drasil framework could be it.

Chapter 1

Introduction

Drasil is a framework that generates software, including code, documentation, software requirement specification, user manual, axillary files, and so on. We call those artifacts "software artifacts". By now, the Drasil framework targets generating software to overcome scientific problems. Recently, the Drasil team has been interested in expanding its knowledge to solve a higher-order ordinary differential equation (ODE). It would not be difficult to directly add ODE knowledge into the Drasil framework because this requires Drasil to have codified knowledge in ODE, which Drasil currently doesn't have. Thus, we believe a compromised way to solve a higher-order ODE is to generate a program interface that connects with its ODE external libraries. There are three main reasons why we want to do that.

1. Scientists and researchers frequently use ODE as a research model in scientific problems, and this model describes the nature phenomenons. Building a research model in software is relatively common, and the software that the Drasil framework generates can solve scientific problems. Thus, expanding the Drasil framework's potential to solve all ODE would solve many scientific problems. Currently, the Drasil

can only solve first-order ODEs.

2. Many external libraries are hard to write and embody much knowledge, so the Drasil team wants to re-use them instead of reproducing them. Among many external libraries, libraries that solve ODEs are probably the most important ones.

3. Another reason is that the Drasil team is interested in how the Drasil framework interacts with external libraries. Once the team understands how to interact between the Drasil framework and external libraries, they will start to add more external libraries. In this way, it would unlock the potential to allow the Drasil framework to solve more scientific problems than before.

However, the Drasil framework neither captures ODE knowledge nor solves higher-order ordinary differential equations. The previous researcher researched to solve a first-order ODE, but it only covers a small area of the knowledge of ordinary differential equations. Adding higher-order linear ODEs into the Drasil framework will expand the area where it has never reached before. Therefore, my research will incorporate higher-order linear ODEs in a complex knowledge-based and generative environment that can link to externally provided libraries.

To solve a higher-order linear ODE, we have to represent ODEs in the Drasil database. On the one hand, users can input an ODE as naturally as writing an ODE in mathematical expressions, such as the example 2.2.2. On the other hand, they can display the ODE in the style of conventional mathematical expressions. The data representation will preserve the relationship between each element in the equation. Then, we will analyze the commonality and variability of selected four external libraries. This analysis will lead us to know how external libraries solve ODEs, what their capabilities are, what options they have, and what interfaces look like. Last, we need to bridge the gap between the Drasil ODE data representation and external libraries. The Drasil ODE data representation can not directly communicate with external libraries. Each library has its standard in terms of solving ODEs. The existing gap requires a transformation from the Drasil ODE data representation to a generic data form before solving ODE in each programming language. Finally, users can run software artifacts to get the numerical solution of the ODE.

Before conducting my research, the Drasil framework can solve explicit equations and numerically solve a first-order ODE. After my research, the Drasil framework will have full capability to solve a higher-order linear ODE numerically. Cases study of NoPCM and PDController will utilize a newly created model to generate programs to solve a higher-order linear ODE in four different programming languages. In addition, we will explore the possibility of solving a system of ODE numerically. We will introduce a new case study, the double pendulum, which contains an example that solves a system of higher-order non-linear ODE.

Chapter 2 will cover how to represent the data of linear ODE in Drasil. Then, in Chapter 3, we will analyze external libraries. In Chapter 4, we will explore how to connect the Drasil ODE data representation with external libraries.

Chapter 2

ODE Data Representation

In the Drasil framework, there is a single data structure containing all the information for all products, and we call it System Information. The giant System Information collects a multitude of pieces of information; whenever we need it, we extract the information from the System Information. In previous research, we store all ordinary differential equations (ODEs) information in the System Information. However, that information existed in the form of plain text. In other words, we explicitly wrote ODEs in the text without any advanced data structure. Although this method maintains the relationship of ODEs, it restricts any transformation of ODEs. For example, if the text-based ODE is higher-order linear ODE, we can not transform it to its equivalent system of first-order ODE. Therefore, the Drasil team is exploring new approach to store ODEs in a new data structure, and the new structure would allow ODEs be isomorphic, which means we can map the ODE from one form to other forms. Once we capture ODEs information in this data structure, we can generate its equivalent forms. This approach is contrasting to previous method, and it only requires users write ODEs once. This chapter we will discuss the problem occurs in text-based expression, introduce where the new data structure comes from, how the new data structure captures ODE information, how to use the new data structure, and how the new data structure interacts with the Drasil printer.

2.1 Explicit Equation

Before we conduct this research, the Drasil framework can generate software that provides numerical solutions for a first-order ODE by explicitly writing the equation. We re-write the ODE equation and pass it to the Drasil code generator. In Equation 2.1.1, the model describes the energy balance of water. In NoPCM case study, we can find the temperature of the water base on it.

$$T'_w(t) + \frac{T_w(t)}{\tau_w} = \frac{T_c}{\tau_w} \tag{2.1.1}$$

The $T_w(t)$ is a function of the independent variable, in this case time. The T_w is the temperature of water ($°C$). The $T'_w(t)$ is the first directive of the function $T_w(t)$ respect time. The T_c is the temperature of the heating coil (°C), and the τ_w is the ODE parameter for water related to decay time (s). We can later isolate the $T'_w(t)$ to the left-hand side and move the rest terms to the right-hand side. Then, we can get Equation 2.1.2.

$$T'_w(t) = \frac{T_c - T_w(t)}{\tau_w} \tag{2.1.2}$$

Based on Equation 2.1.2, we can write it into a text-based form and pass it to the Drasil code generator. Code 2.2 shows how to encode Example 3.1.1 by writing the explicit equation. Brooks's thesis (2) documented how the Drasil framework solves Equation 2.1.2 with manually created ODEInfo. The `ODEInfo` is a data type holds

ODE information.

The user will first encode the ODE equation in the general data pool. Whenever we need it, we retrieve the ODE equation from it. However, there is a gap between the original equation and external libraries (Chapter 3). The external libraries can not understand the original ODE equation from the general data pool. Therefore, the Drasil team manually transforms the original ODE equation into another form (`ODEInfo`), which external libraries can use to produce a numerical solution.

Here is an example of how we manually close the gap between the text-based ODE and external libraries. In Code 2.1, we encode Equation 2.1.2 and put it into the general data pool. During printing the SRS, we retrieve the text-based ODE and print it. The Drasil printer is capable of displaying encoded ODE in text. However, external libraries require a specific format for the ODE and can not utilize the original ODE. Therefore, we manually create Code 2.2 so external libraries can solve the ODE. They both describe the same ODE, but we write it twice in the Drasil framework. Therefore, there is an information duplication. We can transform from Code 2.1 to Code 2.2 with human interference. However, without human interference, we can not complete the transformation because Code 2.1 lacks the necessary structure. To reduce the information duplication, the Drasil team decided to make an advanced data structure to hold the ODE information.

```
-- Pesodu Code
T_w'(t) = reciprocal τ_w * (T_c - T_w(t))
```

Code 2.1: NoPCM equation for SRS

```
-- Pesodu Code
reciprocal τ_w * (T_c - T_w[0])
```

Code 2.2: NoPCM equation for the Drasil Code Generator

2.2 Matrix Form

In general, an equation contains a left-hand expression, a right-hand expression, and an equal sign. The left-hand and right-hand expressions connect by an equal sign. A linear ODE also has its left-hand and right-hand sides. Each side has its unique shape. We can write a linear ODE in the shape of

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.2.1}$$

On the left-hand side, **A** is an m $\times$ n matrix, and **x** is an n-vector. On the right-hand side, **b** is an m-vector. The **A** is commonly known as the coefficient matrix, **b** is the constant vector, and **x** is the unknown vector. The equation 2.2.1 can represent not only a single linear ODE, but also represent a linear system of ODE. A linear system of ODE is a finite set of linear differential equations. In this research, we only have case studies for single ODE, and all examples will demonstrate on single ODEs. The new data structure is capable to store information for a system of ODE, but its related functions only support for instances of single ODE.

Given the ODE example 2.2.2 in PDContoller case study,

$$y_t''(t) + (1 + K_d) \cdot y_t'(t) + (20 + K_p) \cdot y_t(t) = r_t \cdot K_p \tag{2.2.2}$$

In Example 2.2.2, there is only one dependent variable y_t. The y_t(t) is a function of independent variable, in this case time. The y_t'(t) is the first derivative of y_t(t) respect time. The y_t''(t) is the second derivative of y_t(t) respect time. The y_t is the process variable, and the y_t' is the rate of change of y_t. The y_t'' is the rate of change of the rate of change of y_t. The K_d , K_p, and r_t are constant variables. The K_d is Derivative Gain, K_p is Proportional Gain, and r_t is Set-Point. We can write this equation as follows.

$$\begin{bmatrix} 1, & 1+K_d, & 20+K_p \end{bmatrix} \cdot \begin{bmatrix} y_t''(t) \\ y_t'(t) \\ y_t(t) \end{bmatrix} = \begin{bmatrix} r_t \cdot K_p \end{bmatrix} \tag{2.2.3}$$

The relationship between the matrix form 2.2.1 and the example 2.2.3 is not hard to find. Firstly, the coefficient matrix **A** is a 1×3 matrix that consists of 1, $1+K_d$, ane $20+K_p$. Secondly, the unknown vector **x** is a 3×1 vector with y_t'', y_t', and y_t. Last, the constant vector **b** is a 1×1 vector with $r_t \cdot K_p$. The matrix form 2.2.1 very well captures all the knowledge we need to present an ODE. But, how a matrix form looks like in a nth-order linear ODE? Based on Paul's Online Notes (1), we can write all linear ODEs in the shape of

$$a_n(t) \cdot y^n(t) + a_{n-1}(t) \cdot y^{n-1}(t) + \cdots + a_1(t) \cdot y'(t) + a_0(t) \cdot y(t) = h(t) \tag{2.2.4}$$

The coefficient $a_0(t), \ldots, a_n(t)$ and $g(t)$ can be constant or non-constant functions, in our case they are constant functions. We also can write Equation 2.2.4 in a matrix form.

$$\begin{bmatrix} a_n(t), & a_{n-1}, \ldots, & a_0(t) \end{bmatrix} \cdot \begin{bmatrix} y_t^n(t) \\ y_t^{n-1}(t) \\ \ldots \\ y_t(t) \end{bmatrix} = \begin{bmatrix} h(t) \end{bmatrix} \tag{2.2.5}$$

Therefore, we decided to create a datatype called `DifferentialModel` to preserve ODEs information. The `DifferentialModel` has six records, and here is the representing code for `DifferentialModel`.

```
data DifferentialModel = SystemOfLinearODEs {
  _indepVar :: UnitalChunk,
  _depVar :: ConstrConcept,
  _coefficients :: [[Expr]],
  _unknowns :: [Unknown],
  _dmConstants :: [Expr],
  _dmconc :: ConceptChunk
}
```

Previous to this research, UnitalChunk, ConstrConcept, Expr, and ConceptChunk already existed in Drasil. We created an `Unknown` type for this experiment. Their semantics will show up in table 2.1

The `_indepVar` represents the independent variable, and it is often time. The `_depVar` represents the dependent variable. Combing `_depVar` and `_indepVar`, it represents a function produce dependent variables over time. The `_coefficients` is a list of lists `Expr`, and it represents the coefficient matrix **A**. The `_unknowns` is

Type	Semantics
`UnitalChunk`	concepts with quantities that must have a unit definition.
`ConstrConcept`	conceptual symbolic quantities with Constraints and maybe a reasonable value.
`Expr`	a type encode mathematical expression.
`ConceptChunk`	a concept that contains an idea, a definition, and an associated domain of knowledge
`Unknown`	synonym of Integer

Table 2.1: Type use in DifferentialModel

a list of `Unknown`, and `Unknown` is synonym of integers. The `_unknowns` represent a list of numbers of derivatives of the function. Combining `_depVar`, `_indepVar` and `_unknowns`, they can represent the unknown vector **x**. The `_dmConstants` is a list of `Expr`, and it represents the constant vector **b**. Last, the `_dmconc` contains metadata of this model. To represent example 2.2.2 in `DifferentialModel`, `_indepVar` is time, `_depVar` is y_t, `_coefficients` is the 1×3 matrix, `_unknowns` is the 3×1 vector, `_dmConstants` is the 1×1 vector, and `_dmconc` is `ConceptChunk` that describes what this model is. Code 2.3 shows the internal data representation of the example 2.2.2 in `DifferentialModel`.

```
_indepVar = time
_depVar = y_t
_coefficients = [[1, 1 + K_d, 20 + K_p]]
_unknowns = [2, 1, 0]
_dmConstants = [r_t * K_p]
_dmconc = ... -- Drasil definition for chuck concept
```

Code 2.3: Internal Data Representation for Example 2.2.2

Currently, the `DifferentialModel` only captures the knowledge of ODEs with one dependent variable, and it is a special case of the family of linear ODEs. Studying this special case will help the Drasil team better understand how to capture the knowledge of all ODEs and eventually lead to solving a system of linear ODE with multiple dependent variables. On top of that, there is one assumption. The `_coefficients` can only be functions of independent variable, often time. In other word, the `_coefficients` does not depend on `_depVar`.

2.3 Input Language

There are many reasons why we want to provide an input language for users to input ODE equations. One major reason is that it could be over complicated for users to input a single ODE in a matrix form. While inputting a single ODE, one obvious way is directly passing value to each record via constructors of `DifferentialModel`. The Code 2.3 shows how to encode Example 2.2.2 in the `DifferentialModel`. However, it would be not so elegant to set a single ODE in the example, because users have to extracts the coefficient matrix $\mathbf{A}$, unknown vector $\mathbf{x}$ and constant vector $\mathbf{b}$ from the original equation manually. Once the coefficient matrix, unknown vector and constant vector is ready, we can set value into `_depVar`, `_coefficients`, `_unknowns`, and `_dmConstants` accordingly. This process is ideal when the ODE is a system of ODE, and it would be over-complicated for user to do extraction for a single ODE. Therefore, we decided create a helper function to ease this issue. On top of that, the Drasil printer will print a single ODE in SRS with a more familiar "one line equation" form. Another advantage of having an helper function to input an ODE is that it can reduce human error and make sure the equation is well-formed. We call this helper function input language, and what will this input language looks like?

The input language is inspired by how a linear nth-order ODE looks like. Based on Paul's Online Notes (1), we can write all linear ODEs in the shape of Equation 2.2.4. On the left-hand side of Equation 2.2.4, the expression is a collection of terms. Each term consists of a coefficient function and a derivative of the function y(t). With ideas of term, coefficient, and derivative, we create new data types to mimic the mathematical expression of a linear ODE. The following is the detail of the code for new data types and operators.

```
type Unknown = Integer
data Term = T{
  _coeff :: Expr,
  _unk :: Unknown
}
type LHS = [Term]

($^^) :: ConstrConcept -> Integer -> Unknown
($^^) _ unk' = unk'

($*) :: Expr -> Unknown -> Term
($*) = T

($+) :: [Term] -> Term -> LHS
($+) xs x  = xs ++ [x]
```

For new types, the `LHS`, the short name for the left-hand side, is a list of `Term`.

This corresponds to the left hand side is a collection of terms. Each `Term` has an `Expr` and `Unknown`. This corresponds to a term consists of a coefficient and a derivative of the function. Although `_unk` is an integer, combining `_unk`, `_depVar` and `_indepVar` we can get the derivative of the function. For new operators, they are inspired by the linear equation 2.2.4. The `$^^` operator take a variable and a integer, and it represents the derivative of the function. For instance, in example 2.2.2, we can write $y_t(t)\hat{}\hat{}2$ to represent $y_t''(t)$. One thing we want to notice here is that we store $y_t(t)$ in `_depVar` and `_indepVar`. The operator `$^^` will ignore the first parameter, and store the second parameter in `_unknowns`. The reason to positioning a dummy variable before `$^^` is becasue this will maintain the whole input structure as close as a linear ODE. The `$*` operator creates a term by combining a coefficient matrix and a derivative function. For instance, in example 2.2.2, we can write $(1 + K_d)\$*(y_t\ \$\hat{}\hat{}1)$ to represent $(1 + K_d) \cdot y_t'(t)$. Last, the `$+` operator will append all terms into a list. Let's write pseudo code (Code 2.4) for the example matrix form 2.2.2 in the newly introduced input language. The full detail of the input language for the PDController example will show up in A.1.

```
-- in Example 2.2.2: y_t'' + (1 + K_d)y_t' + (20 + K_p)y_t = r_t K_p
-- left hand side = y_t'' + (1 + K_d)y_t' + (20 + K_p)y_t
-- right hand side = r_t K_p

lhs = [1 $* (y_t $^^ 2)]
  $+ (1 + K_d) $* (y_t $^^ 1)
  $+ (20 + K_p) $* (y_t $^^ 0)
rhs = r_t * K_p
```

Code 2.4: Input language for the example 2.2.2

2.4 Two Constructors

There are many way to create the a `DifferentialModel`. One most obvious way is to set each record directly by passing values in the constructor and `makeASystemDE` constructor serve as this role. We also designed another constructor, `makeASingleDE`, for users who want to use input language to create a `DifferentialModel`.

For `makeASystemDE` constructor, a user can set the coefficient matrix, unknown vector, and constant vector by explicitly giving `[[Expr]]`, `[Unknown]`, and `[Expr]`. There will be several guards to check whether inputs are well-formed.

1. The coefficient matrix and constant vector dimension need to match. The `_coefficients` is an m $\times$ n matrix, and `_dmConstants` is an m vector. This guard makes sure they have the same m dimension. If an error says "Length of coefficients matrix should equal to the length of the constant vector.", it means `_coefficients` and `_dmConstants` has different m dimension, violating mathematical rules.

2. The dimension of each row in the coefficient matrix and unknown vector need to match. The `_coefficients` use a list of lists to represent an m $\times$ n matrix. It means each list in `_coefficients` will have the same length n, and `_unknowns` is an n-vector. Therefore, the length of each row in the `_coefficients` should equal the length of `_unknowns`. If an error says, "The length of each row vector in coefficients need to equal to the length of unknowns vector.", it means `_coefficients` and `_unknowns` violate mathematical rules.

3. The order of the unknown vector needs to be descending due to design decisions. We have no control over what users will give to us, and there are infinite ways to represent a linear equation in the matrix form 2.2.1. We strictly ask users to input the unknown vector descending, so we can maintain the shape of a normal form of

linear ODE 2.2.4. This design decision will simplify the implementation for solving a linear ODE numerically in Chapter 3. If an error says, "The order of giving unknowns needs to be descending.", it means the order of unknown vector is not descending.

The following pseudo-code shows how to directly set the example 2.2.2's coefficient matrix, unknown vector, and constant vector. The full detail of how to directly set the coefficient matrix, unknown vector, and constant vector for the PDContoller example will show up in the Appendix A.1.

```
coefficient = [[1, 1 + K_d, 20 + K_p]]
unknowns    = [2, 1, 0]
constants   = [r_t * K_p]
```

The second constructor is called `makeASingleDE`. This constructor uses the input language to simplify the input of a single ODE. In `makeASingleDE`, we create the coefficient matrix, unknown vector, and constant vector based on restricted inputs. In other words, users can no longer set the data by directly giving values. The `DifferentialModel` will generate all data for the coefficient matrix, unknown vector, and constant vector accordingly. The constructor first creates a descending unknown vector base on the highest number of its derivatives. To take the code 2.4 as an example, the highest order of its derivative on the left-hand side of the equation is 2, so we will generate the unknown vector, and it is a list that contains 2, 1 and 0. Then, we will create the coefficient matrix by finding its related coefficient based on the descending order of the unknown vector. The main advantage of this design decision is that the `DifferentialModel` will no longer require users to input the unknown vector in descending order. Any order of the unknown vector will be acceptable because we will generate relative data in `DifferentialModel`. The pseudo-code 2.4 shows how to use the input language to set the example 2.2.2's coefficient matrix, unknown vector, and constant vector. The full detail of how to use the input language set the coefficient matrix, unknown vector, and constant vector for the PDContoller example will show up in the Appendix A.1.

We know Unknown a synonym of Integer. In Code 2.5, it generate `[Unknown]` for unknowns in `DifferentialModel`. We search through the `LHS` to get the largest `Unknown`. Then, we create a list of Unknown from the largest `Unknown` to 0.

```
-- | Find the highest order in left hand side
findHighestOrder :: LHS -> Term
findHighestOrder = foldr1 (\x y -> if x ^. unk >= y ^. unk then x
↪  else y)

-- | Create all possible unknowns based on the highest order.
-- | The order of the result list is from the highest degree to
↪  zero degree.
createAllUnknowns :: Unknown -> ConstrConcept -> [Unknown]
createAllUnknowns highestUnk depv
  | highestUnk  == 0  = [highestUnk]
  | otherwise = highestUnk : createAllUnknowns (highestUnk - 1)
↪  depv
```

Code 2.5: Emulate Unknown

In Code 2.6, it generates `[Expr]` for coefficient in `DifferentialModel`. We have a `LHS` and a `[Unknown]`. We construct the coefficients matrix by searching each `Unknown` in `LHS`. Once we find the matched `Unknown`, we collect its related coefficient. If we did not find the matched `Unknown`, we assume there is no related `Term` exist, so we put a 0 as the coefficient.

```
-- | Create Coefficients base on all possible unknowns
-- | The order of the result list is from the highest degree to zero degree.
createCoefficients :: LHS -> [Unknown] -> [Expr]
createCoefficients [] _ = error "Left hand side is an empty list"
createCoefficients _ [] = []
createCoefficients lhs (x:xs) = genCoefficient (findCoefficient x lhs) : createCoefficients lhs xs

-- | Get the coefficient, if it is Nothing, return zero
genCoefficient :: Maybe Term -> Expr
genCoefficient Nothing = exactDbl 0
genCoefficient (Just x) = x ^. coeff

-- | Find the term that match with the unknown
findCoefficient :: Unknown -> LHS -> Maybe Term
findCoefficient u = find(\x -> x ^. unk == u)
```

Code 2.6: Create a coefficient matrix

2.5 Display Matrix

After a `DifferentialModel` obtains ODE information, we want to display them in the software requirements specification (SRS). Previously, we mentioned the Drasil framework able to generate software artifacts, and SRS is a part of them. This section will discuss two ways to display ODEs in the SRS.

1. We can display ODEs in a matrix form. The matrix form 2.2.3 demonstrates how the ODE will appear in a matrix form in the SRS. In the `DifferentialModel`, the coefficient matrix is a list of lists expression, the unknown vector is a list of integers, and the constant vector is a list of expressions. It should be fairly straightforward for the Drasil printer to display them by printing each part sequentially. The example for this option shows in Figure 2.1a. However, we explicitly force the Drasil printer to

Equation	$$\begin{bmatrix}1 & 1+K_{\mathrm{d}} & 20+K_{\mathrm{p}}\end{bmatrix} \cdot \begin{bmatrix}\frac{d^2 y_{\mathrm{t}}}{dt^2} \\ \frac{dy_{\mathrm{t}}}{dt} \\ y_{\mathrm{t}}\end{bmatrix} = \begin{bmatrix}r_{\mathrm{t}} K_{\mathrm{p}}\end{bmatrix}$$

(a) Displaying ODE in a matrix form

Equation	$$\frac{d^2 y_{\mathrm{t}}}{dt^2} + (1+K_{\mathrm{d}})\frac{dy_{\mathrm{t}}}{dt} + (20+K_{\mathrm{p}})y_{\mathrm{t}} = r_{\mathrm{t}} K_{\mathrm{p}}$$

(b) Displaying ODE in a linear equation

Figure 2.1: Options of Displaying an ODE

display a single ODE in shape of a linear equation, because displaying a single ODE in matrix from would be over-complicated. The example is a demo shows the Drasil printer is capable to display an ODE in a matrix form.

2. We also can display ODEs in a shape of a linear equation. The example 2.2.2 demonstrates how the ODE will show up in the shape of a linear equation in the SRS. Displaying a single ODE in a linear equation is a special case. When there is only one single ODE, it would be over complicated to display it in a matrix form. This is the same reason we want to create an input language to manage the input of a single ODE better. The example for this option shows in Figure 2.1b.

In the future, the Drasil team wants to explore more variability in displaying ODEs. One topic highlighted in the discussion is showing an ODE in a canonical form. However, many mathematicians have different opinions on a canonical form, and the name of canonical form has been used differently, such as normal form or standard form. More research on this part would help us better understand the knowledge of ODE.

Chapter 3

External libraries

External libraries are from an outside source; they do not originate from the source project. Our current interest is for libraries that are used to support solving scientific problems. Most external libraries are language-dependent, and the Drasil framework can generate five different languages: Python, Java, C++, C#, and Swift. Among those five languages, four programming languages have ODE libraries for solving ODEs and we did not find a suitable library for Swift. In Python, the Scipy library (11) is a well-known scientific library for solving scientific problems, including support for solving ODEs. In Java, a library called Apache Commons Maths (ACM) (5) provides a supplementary library for solving mathematical and statistical problems not available in the Java programming language. ACM includes support to solve ODEs. Two less known libraries to solve ODEs are ODEINT Library (7) in C++ and the OSLO Library (9) in C#. There could be multiple external libraries to solve the ODE in one language, but we only find one external library for each selected library.

We believe it is beneficial to conduct a commonalty analysis for all four selected libraries because the Drasil framework wants to generate a program family. A Program families (3) is a sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. In this case, we may want to instruct the Code Generator to create programs that solve ODEs in multiple algorithms or allow other output types to interact with other modules. Those programs vary in application demand and different algorithms, so we can take advantage of developing them as a family (4).

The four selected libraries have some commonalities and variabilities. Firstly, they all provide a numerical solution for a system of first-order ODEs. Each library can output a value of the dependent variable at a specific time, and we can collect those values in a time range. Secondly, they all provide different algorithms for solving ODEs numerically, and we will conduct a rough commonality analysis of available algorithms. A completed commonality analysis would be too time-consuming and out of the scope of our study. Lastly, Scipy and OSLO libraries have the potential to output an ODE as a function. This discovery will provide options for the Drasil framework to solve an ODE by generating a library rather than a standalone executable program. Besides commonalities and variabilities, the Drasil team has to learn how to manage external libraries in general. The four selected external libraries are just examples, and there are many useful external libraries out there. The team will likely encounter difficulties of handling external libraries, such as how to handle dependencies in this Drasil framework. This research will start surface some related challenges.

This chapter will discuss topics related to the commonalities and variabilities of four libraries, including numerical solutions, algorithms options and outputting an ODE as a function. Last, we will discuss how we handle dependencies in the framework.

3.1 Numerical Solutions

We use algorithms to make approximations for mathematical equations and create numerical solutions. All numerical solutions are approximations, and some numerical solutions that utilize better algorithms can produce a better result than others. All selected libraries provide numerical solutions for a system of first-order ODE as an initial value problem (IVP). The IVP requires an initial condition that specifies the function's value at the start point, contrasting with boundary value problem (BVP). In a BVP, we apply boundary conditions instead of initial condition. In this research, we will solve each scientific problem as an IVP. Let's see how to solve a system of first-order ODE with an example. Here is an example of a system of first-order ODE.

$$\begin{aligned} x_1'(t) &= x_2(t) \qquad (3.1.1) \\ x_2'(t) &= -(1 + K_d) \cdot x_2(t) - (20 + K_p) \cdot x_1(t) + r_t \cdot K_p \end{aligned}$$

In Example 3.1.1, there are two dependent variables: x_1 and x_2. Both $x_1(t)$ and $x_2(t)$ are functions of the independent variable, in this case time. The x_1 is the process variable, and the x_2 is the rate of change of x_1. The x_1'(t) is the first directive of the function $x_1(t)$ respect time, and the x_2'(t) is the first derivative of the function x_2(t) respect time. The K_d , K_p, and r_t are constant variables, and they remain the same meaning in Example 2.2.2 and example 3.1.1. We can encode the Example 3.1.1 in all four libraries.

In Python Scipy library, we can write the example as the following code:

```
def f(t, y_t):
    return [y_t[1], -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0]
     ↪  + r_t * K_p]
```

In this example, the y_t is a list of dependent variables. The index 0 of y_t is the dependent variable x_1, and the index 1 of y_t is the dependent variable x_2. The y_t[1] represent the first equation $x_1'(t) = x_2(t)$ in Example 3.1.1. The -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0] + r_t * K_p represents the second equation, $x_2'(t) = -(1 + K_d) \cdot x_2(t) - (20 + K_p) \cdot x_1(t) + r_t \cdot K_p$, in Example 3.1.1. In Java ACM library, we can write the example as the following code:

```
public void computeDeriv(double t, double[] y_t, double[] dy_t) {
    dy_t[0] = y_t[1];
    dy_t[1] = -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0] + r_t
 ↪  * K_p;
}
```

In C++ ODEINT library, we can write the example as the following code:

```
void ODE::operator()(vector<double> y_t, vector<double> &dy_t, double t) {
    dy_t.at(0) = y_t.at(1);
    dy_t.at(1) = -(1.0 + K_d) * y_t.at(1) + -(20.0 + K_p) * y_t.at(0) + r_t * K_p;
}
```

In C# OSLO library, we can write the example as the following code:

```
Func<double, Vector, Vector> f = (t, y_t_vec) => {
    return new Vector(y_t_vec[1], -(1.0 + K_d) * y_t_vec[1] + -(20.0 + K_p) * y_t_vec[0] + r_t * K_p);
};
```

Once we capture the information of the system of ODE, we have to give an initial condition for solving an ODE as an IVP. To solve the Example 3.1.1, we must provide the initial value for both x_1 and x_2. Overall, an ODE is a simulation, and it simulates a function of time. Before we start the simulation, other configurations need to be specified, including the start time, end time, and time step between each iteration. We can also provide values for each library's absolute and relative tolerance. Those two tolerances control the accuracy of the solution. As we mentioned before, all numerical solutions are approximations. High tolerances produces less accurate solutions, and smaller tolerances produce more accurate solutions. Last, we have to collect the numerical output for each iteration. The full details on how each library solves the Example 3.1.1 are shown in Appendix A.2, code 3.2, and code 3.1.

3.2 Algorithm Options

We can solve an ODE with many algorithms. The four selected libraries each provide many algorithms. We roughly classify available algorithms into four categories based on the type of algorithm they use. They are a family of Adams methods, a family of backward differentiation formula methods (BDF), a family of Runge-Kutta (RK) methods, and a "catch all" category of other methods. The commonality analysis we provide on available algorithms is a starting point. It is an incomplete approximation. Getting a complete commonality analysis will require help from domain experts in ODEs. Although the commonality is incomplete, the team still benefits from the current analysis. Not only can a future student quickly access information on which algorithm is available in each language, but also the analysis reminds us that we can increase the consistency of artifacts by providing one-to-one mapping for each algorithm in the four libraries. For example, if a user explicitly chooses a family of Adams methods as the targeted algorithm, all available libraries should use a family of Adams methods to solve the ODE. Unfortunately, not all libraries provide a family of Adams methods. Here table 3.1 shows the availability of a family of an algorithm in each library. The full details of each library's algorithm availability are shown in Appendix A.3.

There are some improvements that the Drasil team can do to make the ODE solution better. For example, we found some algorithms use a fixed step size for calculating numerical solutions, and others use an adaptive step size. We add the step size with the current time value to calculate the next value of dependent variables. A fixed step size means the step size is the same in each iteration. An adaptive step size means the step size is not always the same and could change based on other factors.

Algorithm \ Library	**Scipy-Python**	**ACM-Java**	**ODEINT-C++**	**OSLO-C#**
Family of Adams	• Implicit Adams	• Adams Bashforth • Adams Moulton	• Adams Bashforth Moulton	
Family of BDF	• BDF			• Gear's BDF
Family of RK	• Dormand Prince (4)5 • Dormand Prince 8(5,3)	• Explicit Euler • 2ed order • 4th order • Gill fourth order • 3/8 fourth order • Luther sixth order • Higham and Hall 5(4) • Dormand Prince 5(4) • Dormand Prince 8(5,3)	• Explicit Euler • Implicit Euler • Symplectic Euler • 4th order • Dormand Prince 5 • Fehlberg 78 • Controlled Error Stepper • Dense Output Stepper • Rosenbrock 4 • Symplectic RKN McLachlan 6	• Dormand Prince RK547M
Others		• Gragg Bulirsch Stoer	• Gragg Bulirsch Stoer	

Table 3.1: Algorithms support in external libraries

In Table A.3, the ACM library divides algorithms into one group that uses a fixed step and others that uses an adaptive step. This discovery can further influence the design choice of solving ODE numerically in the Drasil framework. Currently, Drasil treats all step sizes as a fixed value, and it would be ideal to allow the step size to be either fixed or adaptive in future.

3.3 Output an ODE

In the Drasil framework, we can generate modularized software. A modularized software will contain a controller module, an input module, a calculation module, and an output module. The controller module contains the main function, the start of the software. The input module handles all input parameters and constraints. We manually create a txt file that contains all input information. The input module will read this file and covert the information to its environment. The calculation module contains all the logic for solving the scientific problem. For example, in higher-order linear ODE, the calculation module contains all functions of calculating the numerical solution. Lastly, the output module will output the solution. In all ODE case studies, it will write the object passed by the calculation module as a string in a txt file. With each module interacting with others, we would like to study the output of the calculation module in ODE case studies. Currently, the calculation module will output a finite sequence of real numbers, $\mathbb{R}^k$. However, a finite sequence of real numbers only captures a partial solution, and we ideally want to capture a complete solution. In other words, we would like to output an infinite sequence of real numbers, $\mathbb{R}^n$, to represent the complete numerical solution. If anyone is interested in a partial solution, we can filter it bases on given constraints, such as the time range and the time step. The reality is outputting an infinite sequence is not always available in the four selected libraries. Most of them only provide numerical solutions in the form of a finite sequence of real numbers, $\mathbb{R}^k$. However, the C# OSLO library not only supports outputting a finite sequence of real numbers but also an infinite sequence of real numbers.

In C# OSLO library, we can get a complete numerical solution that contains all the values of the dependent variable over time. The function `Ode.RK547M` returns an enumerable sequence of solution points, and it is an infinite sequence of real numbers. We can derive a partial numeric solution based on the infinite sequence by calling `SolveFromToStep` with parameters such as start time, end time, and time interval. The return of `Ode.RK547M` is equivalent to an infinite sequence of real numbers, $\mathbb{R}^n$. The return of `sol.SolveFromToStep` is equivalent to a finite sequence of real numbers, $\mathbb{R}^k$. Code 3.1 shows the full details of how to solve Example 3.1.1 in the OSLO library.

In Code 3.1, between line 3 and line 4, we encode the ODE of the Example 3.1.1 in a `Vector`. Between line 7 and line 8, we set the absolute and relative tolerance in the `Options` class. In line 10, we initialize initial values. Next, in line 11, we use `Ode.RK547M` to get an endless sequence of real numbers, $\mathbb{R}^n$. In line 12, we use `SolveFromToStep` to get a partial solution ($\mathbb{R}^k$) base on the start time, the final time, and the time step. Last, between line 13 and line 15, we run a for loop to collect the process variable x_1. With the workflow we described above, the `Ode.RK547M(0.0, initv, f, opts)` captures the information of the ODE, and the return object represents a complete numerical solution of the ODE. Anyone interested in a partial solution can use `SolveFromToStep` to filter out. Therefore, C# OSLO library provides two output types for calculating the ODE.

```
public static List<double> func_y_t(double K_d, double K_p, double r_t, double t_sim, double t_step) {
    List<double> y_t;
    Func<double, Vector, Vector> f = (t, y_t_vec) => {
        return new Vector(y_t_vec[1], -(1.0 + K_d) * y_t_vec[1] + -(20.0 + K_p) * y_t_vec[0] + r_t * K_p);
    };
    Options opts = new Options();
    opts.AbsoluteTolerance = Constants.AbsTol;
    opts.RelativeTolerance = Constants.RelTol;

    Vector initv = new Vector(new double[] {0.0, 0.0});
    IEnumerable<SolPoint> sol = Ode.RK547M(0.0, initv, f, opts);
    IEnumerable<SolPoint> points = sol.SolveFromToStep(0.0, t_sim, t_step);
    y_t = new List<double> {};
    foreach (SolPoint sp in points) {
        y_t.Add(sp.X[0]);
    }

    return y_t;
}
```

Code 3.1: Source code of solving PDController in OSLO

Another output of the calculation module for solving the ODE is to output the solution as a function $\mathbb{R} \rightarrow \mathbb{R}^i$. The i is the number of equations in the ODE. The input is the independent variable, often time, and the output is a sequence of real numbers. In Example 3.1.1, the function type will be $\mathbb{R} \rightarrow \mathbb{R}^2$. The idea of outputting an ODE as a function can be useful when the Drasil framework generates a library. Users have the option to generate a runnable program or a standalone library.

On the one hand, the runnable program contains the main function so users can run generated software directly. On the other hand, the library contains all functions to solve the ODE so that outside software can utilize the generated library via its interfaces. The generated library can provide support for calculating the numerical solution of the ODE, and we find Python Scipy library supports outputting ODE as a function.

In the Python Scipy library, we can return a generic interface called `scipy.integrate.ode` (12), which is a generic interface that can store ODE's information. It contains the relationship between the dependent variable, the independent variables, and other variables. Given an independent variable time, the `scipy.integrate.ode` can calculate dependent variables. If we are interested in a partial numerical solution, we can add other ODE-related information, such as the start time, the end time, and the time step. Code 3.2 shows the full details of how to solve Example 3.1.1 in the Scipy library.

```
def func_y_t(K_d, K_p, r_t, t_sim, t_step):
    def f(t, y_t):
        return [y_t[1], -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0] + r_t * K_p]

    r = scipy.integrate.ode(f)
    r.set_integrator("dopri5", atol=Constants.Constants.AbsTol, rtol=Constants.Constants.RelTol)
    r.set_initial_value([0.0, 0.0], 0.0)
    y_t = [[0.0, 0.0][0]]
    while r.successful() and r.t < t_sim:
        r.integrate(r.t + t_step)
        y_t.append(r.y[0])

    return y_t
```

Code 3.2: Source code of solving PDController in Scipy

In Code 3.2, between line 2 and line 3, we encode the ODE equation of Example 3.1.1 in a list. In line 5, we call `scipy.integrate.ode` to packing ODE information in the generic interface ($\mathbb{R} \rightarrow \mathbb{R}^i$). In line 6, we set the configuration for algorithm choices and how much absolute and relative tolerance are. In line 7, we set initial values and the start time. In line 8, we initialize the result collection. We specify which initial value we want to put in the result collection. In line 9, the while loop represents the whole iteration to calculate the ODE. Line 10 adds the time step in each iteration. In this example, we are only interested in collecting the process variable x_1, so we only collect the process variable in line 11. Last, we return the collection of results in line 13. With the workflow described above, the generic interface `scipy.integrate.ode(f)` captures the information of the ODE, and it represents the ODE as a function.

Table 3.2 summarizes the availability of the calculation module's output type for solving an ODE numerically in the four selected libraries.

3.4 Management Libraries

Once the Drasil framework generates code, the generated code relies on external libraries to calculate an ODE. In the current setting, the Drasil framework keeps copies of external libraries in the repository. In the long run this is not practical because of the amount of space external libraries occupy. Moreover, external libraries are not currently shared across case studies, and each case study will have its own copy of external libraries. The current research has uncovered that the current way of handling dependencies in the Drasil framework is problematic. In the future, the team would like to find a better way to handle dependencies. We used a temporary

Library	**Available Output Type**
Scipy-Python	• $\mathbb{R}^k$ (k is a finite integer) • $\mathbb{R} \to \mathbb{R}^i$ (i is the number of equations in the ODE)
ACM-Java	• $\mathbb{R}^k$
ODEINT-C++	• $\mathbb{R}^k$
OSLO-C#	• $\mathbb{R}^k$ • $\mathbb{R}^n$ (n is an infinite integer)

Table 3.2: Available output type in external libraries

solution, symbolic links, to share external libraries without duplications. By creating a symbolic link file, external libraries become sharable. In the future, the team will conduct further studies to tackle this problem.

Chapter 4

Connect Model to Libraries

In Chapter 2, we stored the information of a higher-order linear ODE in the `DifferentialModel`. This `data type` preserves the relationship of the ODE, and we can transform the ODE into other forms. In Chapter 3, we discuss how to solve a system of first-order ODE numerically in four selected external libraries. However, there is a gap between the `DifferentialModel` and external libraries. If we can close it, the Drasil framework can generate a program that unitize external libraries to solve a high-order ODE numerically. All external libraries cannot directly utilize `DifferentialModel`, so this is the problem we will solve in this chapter. We know that most ways of solving ODEs are intended for first-order ODE, so we want to convert most higher-order ODEs to a system of first-order ODE (13). Therefore, we first transform a higher-order linear ODE into a system of first-order ODE. Then, generating a program that contains proper interfaces for utilizing four selected external libraries. This program solves the system of first-order ODE numerically. The original higher-order linear ODE is equivalent to the system of first-order ODE we solved. Thus, by transitivity, the numerical solution for the system of first-order ODE is also the numerical solution of the higher-order ODE.

In this chapter, we will first discuss how to convert any higher-order linear ODE to a system of first-order ODE in theory. Then, we will discuss about how to enable Drasil Code Generator to generate a program that produce numerical solution for a system of first-order ODE. Lastly, we will discuss about how to automate the generating process.

4.1 Higher Order to First Order

Given a higher-order linear ODE, we can write it as Equation 4.1.1. We isolate the highest derivative y^n on the left-hand side and the rest of terms on the right- hand side. On the right hand side, $f(t, y, y', y'', \ldots, y^{n-1})$ means a function depends on variables t, y, y', ..., and y^{n-1}. The t is the independent variable, and it is often time. The y, y', ..., and y^{n-1} means the dependent variable y, the first derivative of y, and until the (n-1)th derivative.

$$y^n = f(t, y, y', y'', \ldots, y^{n-1}) \tag{4.1.1}$$

Then, we start to introducing new variables, x_1, x_2, ..., and x_n. The number of the newly introduced dependent variable is equal to the highest order of the ODE.

The new relationship is shown below.

$$
\begin{aligned}
x_1 &= y \\
x_2 &= y' \\
&\dots \\
x_n &= y^{n-1}
\end{aligned}
\tag{4.1.2}
$$

After that, we start to differentiating x_1, x_2, ..., and x_n in Equation 4.1.2. This step helps us to get new relationships between each variable.

$$
\begin{aligned}
x_1' &= y' = x_2 \\
x_2' &= y'' = x_3 \\
&\dots \\
x_{n-1}' &= y^{n-1} = x_n \\
x_n' &= y^n = f(t, x_1, x_2, \dots, x_n)
\end{aligned}
\tag{4.1.3}
$$

Since the higher-order ODE is linear ODE, $f(t, x_1, x_2, \dots, x_n)$ is a linear function. We can rewrite $f(t, x_1, x_2, \dots, x_n)$ as the following

$$b_0(t) \cdot x_1 + b_1(t) \cdot x_2 + \cdots + b_{n-1}(t) \cdot x_n + h(t) \tag{4.1.4}$$

$b_0(t)$, ..., $b_{n-1}(t)$ and $h(t)$ are constant functions.

Based on Equation 4.1.3 and Equation 4.1.4, we can we can get:

$$
\begin{aligned}
&x_1' = x_2 \\
&x_2' = x_3 \\
&\ldots \\
&x_{n-1}' = x^n \\
&x_n' = b_0(t) \cdot x_1 + b_1(t) \cdot x_2 + ... + b_{n-1}(t) \cdot x_n + h(t)
\end{aligned}
\tag{4.1.5}
$$

Then, we can rewrite Equation 4.1.5 in a matrix form.

$$
\begin{bmatrix} x_1' \\ x_2' \\ \ldots \\ x_{n-1}' \\ x_n' \end{bmatrix} = \begin{bmatrix} 0, & 1, & 0, & \ldots, & 0 \\ 0, & 0, & 1, & \ldots, & 0 \\ \ldots & & & & \\ 0, & 0, & 0, & \ldots, & 1 \\ b_0(t), & b_1(t), & b_2(t), & \ldots, & b_{n-1}(t) \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \ldots \\ 0 \\ h(t) \end{bmatrix} \tag{4.1.6}
$$

Lastly, we abstract Equation 4.1.6 into a general form, Equation 4.1.7.

$$
\boldsymbol{X}' = \boldsymbol{A}\boldsymbol{X} + \boldsymbol{c} \tag{4.1.7}
$$

The **A** is a coefficient matrix, and **c** is a constant vector. The **X** is the unknown vector that contains functions of the independent variable, often time. The **X**' is a vector that consists of first derivatives of functions of **X**.

4.2 Connect Explicit Equations to Libraries

In previous research conducted by Brooks, we wrote the ODE in a text-based form and stored them in a general data pool. Although, the Drasil printer can print a text-based ODE in the SRS, the Drasil Code Generator cannot utilize the text-based ODE. Therefore, we manually create a `data type`, called `ODEInfo`, to make study cases work. We extract useful information from the original ODE and construct `ODEInfo` for Dasil Code Generator. The Drasil Code Generator utilized `ODEInfo` to generate a program which produce a numerical solution. We can find details on how to generate a program which solve the ODE numerically in Brooks's thesis (2). However, the previous research only completed generating a program for a first-order ODE. Since it is a first-order ODE, we only provide an initial value. For a higher-order ODE, the current setting of `ODEInfo` does not hold all information we need to solve a ODE. Therefore, to enable Drasil Code Generator generating problem for higher-order ODE, the `ODEInfo` need to store multiple initial values. For example, we can convert a fourth-order ODE into a system of first-order ODE. To solve the system ODE as an IVP, we need four initial values. Then, the Drasil Code Generator must adapt to handle multiple initial values.

In Code 4.1, we change the `data type` of `initVal` from a `CodeExpr` to a list of `CodeExpr`. This change allows Drasil users store multiple initial values in a list. Before this change, we can only store one initial value.

We also have to ensure Drasil Code Generator can utilize the new `data type` `[CodeExpr]`. Previously, Drasil Code Generator only handling the `initVal` as `CodeEXpr`. Now the `initVal` becomes `[CodeEXpr]`. In the Drasil framework, we handling a list of `data type` by matrix. The code `initVal info` is just retrieves the `initVal` from an

```
-- Old
data ODEInfo = ODEInfo {
  ...
  initVal :: CodeExpr
  ...
}

-- New
data ODEInfo = ODEInfo {
  ...
  initVal :: [CodeExpr],
  ...
}
```

Code 4.1: Source code for initial values

```
-- Old
initVal info

-- New
matrix[initVal info]
```

Code 4.2: Source code for initial values in Code Generator

`ODEInfo` data type. In Code 4.2, the `matrix` can wrap a `[CodeEXpr]` into a `CodeEXpr`. This change effects generated code. In Python Scipe Code 4.3, it initialize the initial value with a list rather than one entity. The same thing happens in C# OSLO Code 4.4. It initializes a list rather than one object. In Java and C++, the backend code already handles the initial value as a list, so there is no change in artifact for those two languages.

Allowing multiple initial values unlocks the potential for Drasil to generate a program that produce numerical solution for a system of first-order ODE. Since every

```
# Old
  r.set_initial_value(T_init, 0.0)
  T_W = [T_init]

# New
  r.set_initial_value([T_init], 0.0)
  T_W = [[T_init][0]] # Initial values are also a part of the numerical solution, so we have to add the proper initial value to the list.
```

Code 4.3: Source code for initial values in Python

```
// Old
Vector initv = new Vector(T_init);

// New
Vector initv = new Vector(new double[] {T_init});
```

Code 4.4: Source code for initial values in C#

higher-order linear ODE has its equivalent system of first-order ODE, the solution for the system of first-order ODE is also the solution for the higher-order ODE. The same thing happens on non-linear higher-order ODE. If we could transform a higher-order non-linear ODE to a system of first-order ODE, we can solve it through four selected external libraries.

Despite the Double Pendulum case study containing a higher-order non-linear ODE, the Drasil framework can now generate a program to solve it numerically. In the double pendulum case study, we eventually want to solve Equation 4.2.1. There are two second-order ODEs in one system. To solve this system of ODE, we convert them into a system of first-order ODE. The transformation follows the methodology we discussed in Section 4.1. We can convert Equation 4.2.1 into Equation 4.2.2. Once the transformation is complete, we can encode Equation 4.2.2 and pass it to the Drasil Code Generator. However, we cannot show Equation 4.2.2 in the shape of Equation 4.1.7 (**X' = AX + c**) because the ODE is not a linear ODE.

$$\theta_1'' = \frac{-g(2m_1+m_2)\sin\theta_1 - m_2 g\sin(\theta_1-2\theta_2) - 2\sin(\theta_1-\theta_2)m_2({\theta_2'}^2L_2 + {\theta_1'}^2L_1\cos(\theta_1-\theta_2))}{L_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \tag{4.2.1}$$

$$\theta_2'' = \frac{2\sin(\theta_1-\theta_2)({\theta_1'}^2L_1(m_1+m_2) + g(m_1+m_2)\cos\theta_1 + {\theta_2'}^2L_2m_2\cos(\theta_1-\theta_2))}{L_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))}$$

$$\theta_1' = \omega_1 \tag{4.2.2}$$

$$\theta_2' = \omega_2$$

$$\omega_1' = \frac{-g(2m_1+m_2)\sin\theta_1 - m_2 g\sin(\theta_1-2\theta_2) - 2\sin(\theta_1-\theta_2)m_2(\omega 2^2L_2 + \omega 1^2L_1\cos(\theta_1-\theta_2))}{L_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))}$$

$$\omega_2' = \frac{2\sin(\theta_1-\theta_2)({\omega_1}^2L_1(m_1+m_2) + g(m_1+m_2)\cos\theta_1 + {\omega_2}^2L_2m_2\cos(\theta_1-\theta_2))}{L_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))}$$

Figure 4.1 demonstrates how a Double Pendulum example works in a lab environment. The full details of the Double Pendulum's SRS locates in Drasil website. Table 4.1 lists all variables in Double Pendulum example.

Now we have Equation 4.2.2, we can encode it in the Drasil. In Code 4.5, it shows the example of how we encode Equation 4.2.2 in the Drasil.

Once the `dblPenODEInfo` is ready, we will pass it to the Drasil Code generator. It will generate programs solve Double Pendulum in four languages. The details of

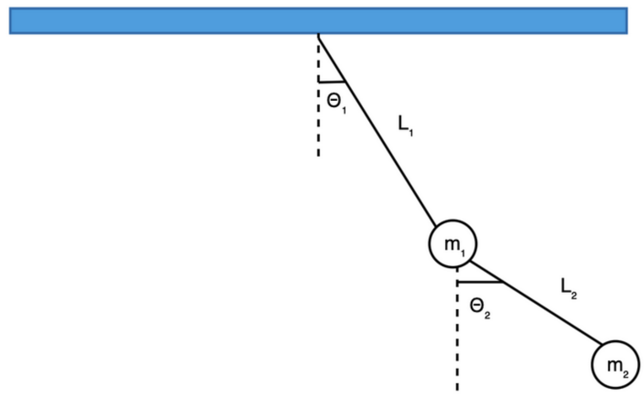

Figure 4.1: Double Pendulum Demonstration

Name	**Description**
$\mathtt{m}_1$	The mass of the first object
$\mathtt{m}_2$	The mass of the second object
$\mathtt{L}_1$	The length of the first rod
$\mathtt{L}_2$	The length of the second rod
$\mathtt{g}$	Gravitational acceleration
θ_1	The angle of the first rod
θ_2	The angle of the second rod
ω_1	The angular velocity of the first object
ω_2	The angular velocity of the second object

Table 4.1: Variables in Double Pendulum Example

generated code for double pendulum will show in Appendix A ("Should I list all generated code in Appendix, it looks ugly"). However, the Double Pendulum case study unable to utilize any function introduced in the next section because it was designed for a linear ODE.

The limitation of manually creating `ODEInfo` is that we will write the ODE twice. In this case, we encode both Equation 4.2.1 and Equation 4.2.2 in Drasil. They both demonstrate the phenomena of double pendulum, and exist in an isomorphic ODE

```
dblPenODEInfo :: ODEInfo
dblPenODEInfo = odeInfo
...
[3*π/7, 0, 3*π/4, 0]
[ ω₁,
  -g(2m₁ + m₂)sin θ₁ - m₂gsin (θ₁ - 2θ₂) - 2sin (θ₁ - θ₂)m₂(ω₂²L₂ +
↪  ω₁²L₁cos (θ₁ - θ₂)) / L₁(2m₁ + m₂ -m₂cos (2θ₁ - 2θ₂)),
  ω₂,
  2sin (θ₁ - θ₂)(ω₁²L₁(m₁ + m₂ ) + g(m₁ + m₂ )cos θ₁ + ω₂²L₂m₂cos (θ₁
↪  - θ₂ )) / L₂(2m₁ + m₂ -m₂cos (2θ₁ - 2θ₂))
]
...
```

Code 4.5: Source code for encoding double pendulum

type. In the next section, we would like to discuss how to automate the transformation from a higher-order ODE to a system of first-order ODE.

4.3 Generate Explicit Equations

Manually creating explicit equations is not ideal because it requires human interference and propagates duplicate information. We want to design the Drasil framework as fully automatically as possible, so we want to remove human interference. Therefore, an ideal solution is to encode the ODE in a flexible data structure. Then, we extract information from this structure and generate a form of ODE which selected external libraries can utilize. Creating the `DifferentialModel` data structure satisfies the need of this idea. We can restructure an ODE base on the information from `DifferentialModel`. This research's scope only covers generating explicit equations for a single higher-order ODE. In the future, we are looking to generate explicit equations for a system of higher-order ODE.

Once we encode the ODE in `DifferentialModel`, we want to restructure its equivalent system of first-order ODE in the shape of Equation 4.1.7. For the convenience of implementation, we shuffle the data around in Equation 4.1.6. We reversed the order of **X** to $x_n, \ldots, x_1$. The coefficient matrix **A** also changed, but **X'** and **c** remain unchanged.

$$\begin{bmatrix} x_1' \\ \cdots \\ x_{n-2}' \\ x_{n-1}' \\ x_n' \end{bmatrix} = \begin{bmatrix} 0, & 0, & \ldots, & 1, & 0 \\ \cdots & & & & \\ 0, & 1, & \ldots, & 0, & 0 \\ 1, & 0, & \ldots, & 0, & 0 \\ b_{n-1}(t), & b_{n-2}(t), & \ldots, & b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \cdots \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ h(t) \end{bmatrix} \tag{4.3.1}$$

Since Equation 4.3.1 is an expansion of Equation 4.1.7, we will use symbols in both equations to explain how to generate Equation 4.3.1. We highlighted **X'** and **X** in yellow colour in Equation 4.3.1. The number of elements in **X'** and **X** depends on how many new dependent variables introduces. If the higher-order ODE is second-order, we will introduce two new dependent variables. If the higher-order ODE is nth-order, we will introduce n new dependent variables. For **X'**, knowing it is n-th order ODE, we parameterize x' from 1 to n. For **X**, knowing it is n-th order ODE, we parameterize x from n to 1.

We highlighted the n×n coefficient matrix **A** in orange and blue colour in Equation 4.3.1. The orange part is an matrix looks like an identity matrix. For the lowest higher-order ODE second-order, the orange part is [1, 0]. Equation 4.3.2 shows a completed transformation for a second-order ODE.

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} \boxed{1}, & \boxed{0} \\ b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ h(t) \end{bmatrix} \tag{4.3.2}$$

If it is a fourth-order ODE, the **A** will be an n×n matrix. Equation 4.3.3 shows a completed transformation for a fourth-order ODE.

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \end{bmatrix} = \begin{bmatrix} \boxed{0}, & \boxed{0}, & \boxed{1}, & \boxed{0} \\ \boxed{0}, & \boxed{1}, & \boxed{0}, & \boxed{0} \\ \boxed{1}, & \boxed{0}, & \boxed{0}, & \boxed{0} \\ b_3(t), & b_2(t), & b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ h(t) \end{bmatrix} \tag{4.3.3}$$

The orange part starts at (n-1)th row with $[1, 0, \ldots]$. If there is a second row, we add $[0, 1, \ldots]$ before the start row and so on. We observe there is a pattern for the orange part, so that we can generate it. In Code 4.6, `constIdentityRowVect` and `addIdentityValue` are responsible for each row in the orange part. We first create a row containing a list of 0. Then, we replace one of 0s to 1. The `addIdentityCoeffs` run through a recursion to create a `[[Expr]]` that responsible to orange part.

We highlighted the constant vector **c** in gray and red colour in Equation 4.3.1. The vector **c** has the length of n. The last element of the constant vector **c** will be h(t), and anything above h(t) will be zeros. In Code 4.7, in `addIdentityConsts`, given the expression of h(t) and the order number of the ODE, we add (n-1) 0s above the h(t).

The blue and red parts in Equation 4.3.1 can be determined by Equation 2.2.4. The `DifferentialModel` preserves the relationship for Equation 2.2.4, but it does

```haskell
-- | Add Identity Matrix to Coefficients
-- | len is the length of the identity row,
-- | index is the location of identity value (start with 0)
addIdentityCoeffs :: [[Expr]] -> Int -> Int -> [[Expr]]
addIdentityCoeffs es len index
  | len == index + 1 = es
  | otherwise = addIdentityCoeffs (constIdentityRowVect len index : es) len (index + 1)

-- | Construct an identity row vector.
constIdentityRowVect :: Int -> Int -> [Expr]
constIdentityRowVect len index = addIdentityValue index $ replicate len $ exactDbl 0

-- | Recreate the identity row vector with identity value
addIdentityValue :: Int -> [Expr] -> [Expr]
addIdentityValue n es = fst splits ++ [exactDbl 1] ++ tail (snd splits)
  where splits = splitAt n es
```

Code 4.6: Source code for creating identity matrix(highlighted in orange)

not isolate the highest order to the left-hand side. To isolate the highest order, we have to shuffle terms between the left-hand side and right-hand side. The following is Equation 2.2.4.

$$a_n(t) \cdot y^n(t) + a_{n-1}(t) \cdot y^{n-1}(t) + \cdots + a_1(t) \cdot y'(t) + a_0(t) \cdot y(t) = h(t)$$

Firstly, we move every term from left to right, except the highest order term.

$$a_n(t) \cdot y^n(t) = -a_{n-1}(t) \cdot y^{n-1}(t) + \cdots + -a_1(t) \cdot y'(t) + -a_0(t) \cdot y(t) + h(t)$$

```haskell
-- | Add Identity Matrix to Constants
-- | len is the size of new constant vector
addIdentityConsts :: [Expr] -> Int -> [Expr]
addIdentityConsts expr len = replicate (len - 1) (exactDbl 0) ++
  ↪ expr
```

Code 4.7: Source code for creating constant matrix **c**

Secondly, we cancel out the coefficient, $a_n(t)$.

$$y^n(t) = \frac{-a_{n-1}(t) \cdot y^{n-1}(t) + \cdots + -a_1(t) \cdot y'(t) + -a_0(t) \cdot y(t) + h(t)}{a_n(t)}$$

Then, this can be written in a matrix form

$$\begin{bmatrix} y^n(t) \end{bmatrix} = \begin{bmatrix} -\frac{a_{n-1}(t)}{a_n(t)}, \ldots, & -\frac{a_1(t)}{a_n(t)} & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} y^{n-1}(t) \\ \cdots \\ y'(t) \\ y(t) \end{bmatrix} + \begin{bmatrix} \frac{h(t)}{a_n(t)} \end{bmatrix}$$

Since $x'_n = y_n$ (Equation 4.1.3), we can replace y_n with x'_n. Based on Equation 4.1.2, we replace all derivatives of y(t) with x_n, $\ldots$, x_1.

$$\begin{bmatrix} x'_n \end{bmatrix} = \begin{bmatrix} -\frac{a_{n-1}(t)}{a_n(t)}, \ldots, & -\frac{a_1(t)}{a_n(t)} & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \cdots \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} \frac{h(t)}{a_n(t)} \end{bmatrix}$$

Lastly, we replacing new variables in Equation 4.3.1, we can get a new matrix.

$$\begin{bmatrix} x_1' \\ \cdots \\ x_{n-2}' \\ x_{n-1}' \\ x_n' \end{bmatrix} = \begin{bmatrix} 0, & 0, & \cdots, & 1, & 0 \\ \cdots & & & & \\ 0, & 1, & \cdots, & 0, & 0 \\ 1, & 0, & \cdots, & 0, & 0 \\ -\frac{a_{n-1}(t)}{a_n(t)}, & -\frac{a_{n-2}(t)}{a_n(t)}, & \cdots, & -\frac{a_1(t)}{a_n(t)}, & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \cdots \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ \frac{h(t)}{a_n(t)} \end{bmatrix} \tag{4.3.4}$$

Here is the implementation for creating Equation 4.3.4 in Drasil. In Code 4.8, we remove the highest order because we want to isolate the highest order to the left-hand side.

```
-- | Delete the highest order
transUnknowns :: [Unknown] -> [Unknown]
transUnknowns = tail
```

Code 4.8: Source code for isolating the highest order

In Code 4.9, the `transCoefficients` cancel out the coefficient, $a_n(t)$, in highlighted blue. The `divideCosntants` cancels out the coefficient in red highlighted.

In Code 4.10, we create a new `data type` called `ODESolverFormat`. The `ODESolverFormat` contains information for the system of first-order ODE. The `coeffVects`, `unknownVect`, and `constantVect` are responsible for $\mathbf{A}$, $\mathbf{X}$, and $\mathbf{c}$ in Equation 4.1.7 ($\mathbf{X'} = \mathbf{AX} + \mathbf{c}$). The `makeAODESolverFormat` is a smart constructor to create an `ODESolverFormat` by giving a `DifferentialModel`.

In Chapter 3, we mentioned we would solve the ODE as an IVP. In Code 4.11, we

```haskell
-- | Cancel the leading coefficient of the highest order in the
↪ coefficient matrix
transCoefficients :: [Expr] -> [Expr]
transCoefficients es
  | head es == exactDbl 1 = mapNeg $ tail es
  | otherwise = mapNeg $ tail $ map ($/ head es) es
    where mapNeg = map (\x -> if x == exactDbl 0 then exactDbl 0
↪ else neg x)

-- | divide the leading coefficient of the highest order in
↪ constant
divideCosntants :: Expr -> Expr -> Expr
divideCosntants a b
  | b == exactDbl 0 = error "Divisor can't be zero"
  | b == exactDbl 1 = a
  | otherwise       = a $/ b
```

Code 4.9: Source code for canceling the coefficient from the highest order

create `InitialValueProblem` to store IVP-related information, includes initial time, final time and initial values.

Lastly, in Code 4.12, we create a new smart construct to generate the `ODEInfo` automatically. In `odeInfo'`, the first parameter is `[CodeVarChunk]`. There will likely be other variables in the ODE. The `[CodeVarChunk]` contains variables other than the dependent and independent variables. The ODEOptions is instructing external libraries on how to solve the ODE. The `DifferentialModel` contains core information for the higher-order ODE. Last, `InitialValueProblem` contains information for solving the ODE numerically. The `createFinalExpr` creates multiple expressions in a list. Those expressions were created based on information on the system of first-order ODE. The `formEquations` take parameters the coefficient matrix $\mathbf{A}$ (`[[Expr]]`), the unknown vector $\mathbf{X}$(`[Unknown]`), and the constant vector(`[Expr]`). Then, we form

```haskell
-- Acceptable format for ODE solvers
-- X' = AX + c
-- coeffVects is A - coefficient matrix with identity matrix
-- unknownVect is X - unknown column vector after reduce the highest order
-- constantVect is c - constant column vector with identity matrix,
-- X' is a column vector of first-order unknowns
data ODESolverFormat = X'{
  coeffVects :: [[Expr]],
  unknownVect :: [Integer],
  constantVect :: [Expr]
}

-- | Construct an ODESolverFormat for solving the ODE.
makeAODESolverFormat :: DifferentialModel -> ODESolverFormat
makeAODESolverFormat dm = X' transEs transUnks transConsts
  where transUnks = transUnknowns $ dm ^. unknowns
        transEs = addIdentityCoeffs [transCoefficients $ head (dm ^. coefficients)] (length transUnks) 0
        transConsts = addIdentityConsts [head (dm ^. dmConstants) `divideCosntants` head (head (dm ^. coefficients))] (length transUnks)
```

Code 4.10: Source code for generating Equation 4.1.7, **X’** = **AX** + **c**

responsible expressions. The first row of **A** cross product the **X**, then we add all terms together with a responsible constant term. In the second row of **A**, we do the same thing. The `formEquations` will output a list of expressions equivalent to Equation 4.1.5. Once the explicit equation for the higher-order ODE is created, we can pass it to Drasil Code Generator.

```
-- Information for solving an initial value problem
data InitialValueProblem = IVP{
  initTime :: Expr, -- inital time
  finalTime :: Expr, -- end time
  initValues :: [Expr] -- initial values
}
```

Code 4.11: Source code for IVP infomation

```haskell
odeInfo' :: [CodeVarChunk] -> ODEOptions -> DifferentialModel -> InitialValueProblem -> ODEInfo
odeInfo' ovs opt dm ivp = ODEInfo
  (quantvar $ _indepVar dm)
  (quantvar $ _depVar dm)
  ovs
  (expr $ initTime ivp)
  (expr $ finalTime ivp)
  (map expr $ initValues ivp)
  (createFinalExpr dm)
  opt

createFinalExpr :: DifferentialModel -> [CodeExpr]
createFinalExpr dm = map expr $ formEquations (coeffVects ode) (unknownVect ode) (constantVect ode) (_depVar dm)
  where ode = makeAODESolverFormat dm

formEquations :: [[Expr]] -> [Unknown] -> [Expr] -> ConstrConcept-> [Expr]
formEquations [] _ _ _ = []
formEquations _ [] _ _ = []
formEquations _ _ [] _ = []
formEquations (ex:exs) unks (y:ys) depVa =
  (if y == exactDbl 0 then finalExpr else finalExpr `addRe` y) : formEquations exs unks ys depVa
  where indexUnks = map (idx (sy depVa) . int) unks -- create X
        filteredExprs = filter (\x -> fst x /= exactDbl 0) (zip ex indexUnks) -- remove zero coefficients
        termExprs = map (uncurry mulRe) filteredExprs -- multiple coefficient with depend variables
        finalExpr = foldl1 addRe termExprs -- add terms together
```

Code 4.12: Source code for generating ODEInfo

Chapter 5

Summary of future works

1. generate explicit equation for a system of higher-order ODE 3. allow adaptive step 4. search better solution for handling dependency 5. remove external libraries, use Drasil framework to solve ODE.

Chapter 6

Conclusion

Every thesis also needs a concluding chapter

Appendix A

Your Appendix

This appendix provides detailed explanations of various parts of DifferentialModel.

A.1 Constructors of DifferentialModel

```
-- K_d is qdDerivGain
-- y_t is opProcessVariable
-- K_p is qdPropGain
-- r_t is qdSetPointTD
imPDRC :: DifferentialModel
imPDRC = makeASingleDE
  time
  opProcessVariable
  lhs
  rhs
  "imPDRC"
  (nounPhraseSP
  ↪  "Computation of the Process Variable as a function of time")
  EmptyS
  where
  lhs = [exactDbl 1 `addRe` sy qdDerivGain $* (opProcessVariable
  ↪  $^^ 1)]
  $+ (exactDbl 1 $* (opProcessVariable $^^ 2))
  $+ (exactDbl 20 `addRe` sy qdPropGain $* (opProcessVariable $^^
  ↪  0))
  rhs = sy qdSetPointTD `mulRe` sy qdPropGain
```

Code A.1: Using input language for the example 2.2.2 in DifferentialModel

```
imPDRC :: DifferentialModel
imPDRC = makeASystemDE
  time
  opProcessVariable
  coeffs = [[exactDbl 1, exactDbl 1 `addRe` sy qdDerivGain, exactDbl 20 `addRe` sy qdPropGain]]
  unknowns = [2, 1, 0]
  constants = [sy qdSetPointTD `mulRe` sy qdPropGain]
  "imPDRC"
  (nounPhraseSP "Computation of the Process Variable as a function of time")
  EmptyS
```

Code A.2: Explicitly set values for the example 2.2.2 in DifferentialModel

A.2 Numerical Solution Implementation

```java
public static ArrayList<Double> func_y_t(double K_d, double K_p, double r_t, double t_sim, double t_step) {
  ArrayList<Double> y_t;
  ODEStepHandler stepHandler = new ODEStepHandler();
  ODE ode = new ODE(K_p, K_d, r_t);
  double[] curr_vals = {0.0, 0.0};

  FirstOrderIntegrator it = new DormandPrince54Integrator(t_step, t_step, Constants.AbsTol, Constants.RelTol);
  it.addStepHandler(stepHandler);
  it.integrate(ode, 0.0, curr_vals, t_sim, curr_vals);
  y_t = stepHandler.y_t;

  return y_t;
}
```

Code A.3: A linear system of first-order representation in ACM

```
vector<double> func_y_t(double K_d, double K_p, double r_t, double t_sim, double t_step) {
  vector<double> y_t;
  ODE ode = ODE(K_p, K_d, r_t);
  vector<double> currVals{0.0, 0.0};
  Populate pop = Populate(y_t);

  boost::numeric::odeint::runge_kutta_dopri5<vector<double>> rk = boost::numeric::odeint::runge_kutta_dopri5<vector<double>>();
  auto stepper = boost::numeric::odeint::make_controlled(Constants::AbsTol, Constants::RelTol, rk);
  boost::numeric::odeint::integrate_const(stepper, ode, currVals, 0.0, t_sim, t_step, pop);

  return y_t;
}
```

Code A.4: A linear system of first-order representation in ODEINT

A.3 Algorithm in External Libraries

Name	Description
`zvode`	Complex-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
`lsoda`	Real-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides automatic method switching between implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
`dopri5`	This is an explicit runge-kutta method of order (4)5 due to Dormand & Prince (with stepsize control and dense output).
`dop853`	This is an explicit runge-kutta method of order 8(5,3) due to Dormand & Prince (with stepsize control and dense output).

Table A.1: Algorithm Options in Scipy - Python (12)

Name	Description
`RK547M`	This method is most appropriate for solving non-stiff ODE systems. It is based on classical Runge-Kutta formulae with modifications for automatic error and step size control.
`GearBDF`	It is an implementation of the Gear back differentiation method, a multi-step implicit method for stiff ODE systems solving.

Table A.2: Algorithm Options in OSLO - C# (10)

Step Size	Name	Description
Fixed Step	`Euler`	This class implements a simple Euler integrator for Ordinary Differential Equations.
	`Midpoint`	This class implements a second order Runge-Kutta integrator for Ordinary Differential Equations.
	`Classical RungeKutta`	This class implements the classical fourth order Runge-Kutta integrator for Ordinary Differential Equations (it is the most often used Runge-Kutta method).
	`Gill`	This class implements the Gill fourth order Runge-Kutta integrator for Ordinary Differential Equations.
	`Luther`	This class implements the Luther sixth order Runge-Kutta integrator for Ordinary Differential Equations.
Adaptive Stepsize	`Higham and Hall`	This class implements the 5(4) Higham and Hall integrator for Ordinary Differential Equations.
	`DormandPrince 5(4)`	This class implements the 5(4) Dormand-Prince integrator for Ordinary Differential Equations.
	`DormandPrince 8(5,3)`	This class implements the 8(5,3) Dormand-Prince integrator for Ordinary Differential Equations.
	`Gragg-Bulirsch-Stoer`	This class implements a Gragg-Bulirsch-Stoer integrator for Ordinary Differential Equations.
	`Adams-Bashforth`	This class implements explicit Adams-Bashforth integrators for Ordinary Differential Equations.
	`Adams-Moulton`	This class implements implicit Adams-Moulton integrators for Ordinary Differential Equations.

Table A.3: Algorithm Options in Apache Commons Maths - Java (6)

Name	**Description**
`euler`	Explicit Euler: Very simple, only for demonstrating purpose
`runge_kutta4`	Runge-Kutta 4: The classical Runge Kutta scheme, good general scheme without error control.
`runge_kutta_cash_karp54`	Cash-Karp: Good general scheme with error estimation.
`runge_kutta_dopri5`	Dormand-Prince 5: Standard method with error control and dense output.
`runge_kutta_fehlberg78`	Fehlberg 78: Good high order method with error estimation.
`adams_bashforth_moulton`	Adams-Bashforth-Moulton: Multi-step method with high performance.
`controlled_runge_kutta`	Controlled Error Stepper: Error control for the Runge-Kutta steppers.
`dense_output_runge_kutta`	Dense Output Stepper: Dense output for the Runge-Kutta steppers.
`bulirsch_stoer`	Bulirsch-Stoer: Stepper with step size, order control and dense output. Very good if high precision is required..
`implicit_euler`	Implicit Euler: Basic implicit routine.
`rosenbrock4`	Rosenbrock 4: Solver for stiff systems with error control and dense output.
`symplectic_euler`	Symplectic Euler: Basic symplectic solver for separable Hamiltonian system.
`symplectic_rkn_sb3a_mclachlan`	Symplectic RKN McLachlan: Symplectic solver for separable Hamiltonian system with order 6.

Table A.4: Algorithm Options in ODEINT - C++ (8)

```
def f(t, theta):
  return [theta[1], (-9.8 * (2.0 * m_1 + m_2) * math.sin(theta[0])
   ↪  - m_2 * 9.8 * math.sin(theta[0] - 2.0 * theta[2]) - 2.0 *
   ↪  math.sin(theta[0] - theta[2]) * m_2 * (theta[3] ** 2.0 * L_2
   ↪  + theta[1] ** 2.0 * L_1 * math.cos(theta[0] - theta[2]))) /
   ↪  (L_1 * (2.0 * m_1 + m_2 - m_2 * math.cos(2.0 * theta[0] -
   ↪  2.0 * theta[2]))), theta[3], 2.0 * math.sin(theta[0] -
   ↪  theta[2]) * (theta[1] ** 2.0 * L_1 * (m_1 + m_2) + 9.8 *
   ↪  (m_1 + m_2) * math.cos(theta[0]) + theta[3] ** 2.0 * L_2 *
   ↪  m_2 * math.cos(theta[0] - theta[2])) / (L_2 * (2.0 * m_1 +
   ↪  m_2 - m_2 * math.cos(2.0 * theta[0] - 2.0 * theta[2])))]
```

```
public void computeDeriv(double t, double[] theta, double[]
 ↪  dtheta) {
  dtheta[0] = theta[1];
  dtheta[1] = (-9.8 * (2.0 * m_1 + m_2) * Math.sin(theta[0]) - m_2
 ↪  * 9.8 * Math.sin(theta[0] - 2.0 * theta[2]) - 2.0 *
 ↪  Math.sin(theta[0] - theta[2]) * m_2 * (Math.pow(theta[3], 2.0)
 ↪  * L_2 + Math.pow(theta[1], 2.0) * L_1 * Math.cos(theta[0] -
 ↪  theta[2]))) / (L_1 * (2.0 * m_1 + m_2 - m_2 * Math.cos(2.0 *
 ↪  theta[0] - 2.0 * theta[2])));
  dtheta[2] = theta[3];
  dtheta[3] = 2.0 * Math.sin(theta[0] - theta[2]) *
 ↪  (Math.pow(theta[1], 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1 +
 ↪  m_2) * Math.cos(theta[0]) + Math.pow(theta[3], 2.0) * L_2 *
 ↪  m_2 * Math.cos(theta[0] - theta[2])) / (L_2 * (2.0 * m_1 + m_2
 ↪  - m_2 * Math.cos(2.0 * theta[0] - 2.0 * theta[2])));
}
```

```
void ODE::operator()(vector<double> theta, vector<double> &dtheta, double t) {
  dtheta.at(0) = theta.at(1);
  dtheta.at(1) = (-9.8 * (2.0 * m_1 + m_2) * sin(theta.at(0)) - m_2 * 9.8 * sin(theta.at(0) - 2.0 * theta.at(2)) - 2.0 * sin(theta.at(0) - theta.at(2)) * m_2 * (pow(theta.at(3), 2.0) * L_2 + pow(theta.at(1), 2.0) * L_1 * cos(theta.at(0) - theta.at(2)))) / (L_1 * (2.0 * m_1 + m_2 - m_2 * cos(2.0 * theta.at(0) - 2.0 * theta.at(2))));
  dtheta.at(2) = theta.at(3);
  dtheta.at(3) = 2.0 * sin(theta.at(0) - theta.at(2)) * (pow(theta.at(1), 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1 + m_2) * cos(theta.at(0)) + pow(theta.at(3), 2.0) * L_2 * m_2 * cos(theta.at(0) - theta.at(2))) / (L_2 * (2.0 * m_1 + m_2 - m_2 * cos(2.0 * theta.at(0) - 2.0 * theta.at(2))));
}
```

```
Func<double, Vector, Vector> f = (t, theta_vec) => {
  return new Vector(theta_vec[1], (-9.8 * (2.0 * m_1 + m_2) * Math.Sin(theta_vec[0]) - m_2 * 9.8 * Math.Sin(theta_vec[0] - 2.0 * theta_vec[2]) - 2.0 * Math.Sin(theta_vec[0] - theta_vec[2]) * m_2 * (Math.Pow(theta_vec[3], 2.0) * L_2 + Math.Pow(theta_vec[1], 2.0) * L_1 * Math.Cos(theta_vec[0] - theta_vec[2]))) / (L_1 * (2.0 * m_1 + m_2 - m_2 * Math.Cos(2.0 * theta_vec[0] - 2.0 * theta_vec[2]))), theta_vec[3], 2.0 * Math.Sin(theta_vec[0] - theta_vec[2]) * (Math.Pow(theta_vec[1], 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1 + m_2) * Math.Cos(theta_vec[0]) + Math.Pow(theta_vec[3], 2.0) * L_2 * m_2 * Math.Cos(theta_vec[0] - theta_vec[2])) / (L_2 * (2.0 * m_1 + m_2 - m_2 * Math.Cos(2.0 * theta_vec[0] - 2.0 * theta_vec[2]))));
};
```

Bibliography

[1] Dawkins, P. Paul's Online Notes linear differential equations. https://tutorial.math.lamar.edu/Classes/DE/Definitions.aspx. Accessed: 2022-08-31.

[2] Maclachlan, B. A design language for scientific computing software in drasil. *Masters thesis, McMaster University, Hamilton, Ontario, Canada* (2020), 91–103.

[3] Parnas, D. On the design and development of program families. *IEEE Transaction on Software Engineering 5*, 2 (March 1976), 1–9.

[4] Smith, S., and Chen, C.-H. Commonality and requirements analysis for mesh generating software. *McMaster University, Hamilton, Ontario, Canada* (2004), 2.

[5] The Apache Software Foundation. Commons math: The Apache commons mathematics library. https://commons.apache.org/proper/commons-math. Accessed: 2022-08-31.

[6] The Apache Software Foundation. Commons math: The Apache ordinary differential equations integration. https://commons.apache.org/proper/commons-math/userguide/ode.html. Accessed: 2022-08-31.

[7] The ODEINT Project. ODEINT solving odes in c++. http://headmyshoulder.github.io/odeint-v2/. Accessed: 2022-08-31.

[8] The ODEINT Project. ODEINT v2 solving ordinary differential equations in c++. https://www.codeproject.com/Articles/268589/odeint-v2-Solving-ordinary-differential-equations. Accessed: 2022-08-31.

[9] The OSLO Project. OSLO open solving library for odes. https://www.microsoft.com/en-us/research/project/open-solving-library-for-odes/. Accessed: 2022-08-31.

[10] The OSLO Project. OSLO open solving library for odes (oslo) 1.0 user guide, getting started, step 7. https://www.microsoft.com/en-us/research/wp-content/uploads/2014/07/osloUserGuide.pdf. Accessed: 2022-08-31.

[11] The SciPy Project. SciPy.org. https://www.scipy.org/. Accessed: 2022-08-31.

[12] The SciPy Project. SciPy.org scipy.integrate.ode. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html. Accessed: 2022-08-31.

[13] Young, T., and Mohlenkamp, M. J. Introduction to Numerical Methods and Matlab Programming for Engineers reduction of higher order equations

to systems. http://www.ohiouniversityfaculty.com/youngt/IntNumMeth/lecture29.pdf. Accessed: 2022-08-31.