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` captures ~~preserves~~ the relationship of the ODE, so that ~~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. The ~~Four~~ selected libraries cannot solve the higher-order ODE directly, but they can solve its equivalent system of first-order equations. ~~ODE~~. We know that most ways of solving ODEs are intended for systems of first-order ODEs, so we want to convert the higher-order ODE to a system of first-order ODE (13). Firstly, we 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 ODEs numerically by producing a list of values of dependent variables based on time. 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 the Drasil Code Generator to generate a program that produces a numerical solution for a system of first-order ODEs. Lastly, we will discuss about how to automate the generation ~~generating~~ process.

4.1 Higher Order to First Order

following

be written in the form:

more equation to right after you mention it

Any linear

~~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~~ which is often time. The y, y', ..., and y^{n-1} ~~means~~ represent the dependent variable y, the first derivative of y, ~~and until~~ etc., up to the (n-1)th derivative.

$(n-1)^\text{th}$ requires \usepackage{amsmath}

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

For the next step

To ~~the next step~~

~~Then~~, we ~~start to~~ introduc~~ing~~e the following 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 (n). The new relationship is shown below.

$$x_1 = y \tag{4.1.2}$$

$$x_2 = y'$$

$$\ldots$$

$$x_n = y^{n-1}$$

~~After that~~ next, we ~~start to differentiating~~ differentiate x_1, x_2, ..., ~~and~~ x_n ~~in~~ in ~~and substitute into~~ Equation 4.1.2 ~~. This step helps us to get new~~ to establish the following relationships between ~~each~~ the variables:

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

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

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

where $b_0(t)$, ..., $b_{n-1}(t)$ and $h(t)$ are constant functions. do you actually require this? The coefficients are allowed to be functions of the independent variable (t); they don't have to be constants in general.

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

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

~~Then,~~ We can rewrite Equation 4.1.5 in ~~a~~ matrix form:

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

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

normally column vectors are lower case - please make this change throughout.

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

$x' = Ax + c$

should be in italics to match the eq.

The $\mathbf{A}$ is a coefficient matrix, and $\mathbf{c}$ is a constant vector. The ~~$\mathbf{X}$~~ x is the unknown vector that contains functions of the independent variable~~, often time~~ [illegible]. The ~~$\mathbf{X}$’~~ x' is a vector that consists of first derivatives of functions of ~~$\mathbf{X}$~~ x.

4.2 Connect Explicit Equations to Libraries

citation

In previous research ~~conducted by~~ Brooks~~, we~~ wrote the ODE in a text-based form and stored ~~them~~ it 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~~ & in the previous approach the user has to repeat information to generate code. through a manually create~~d~~ a `data type`, called ODEInfo, to ~~make study cases~~ studies work. ~~We~~ we extract useful information from the original ODE and construct `ODEInfo` for the Dasil Code Generator automatically. 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 a first-order ODE numerically in Brooks’s thesis (2).

In our improved approach

I think I updated this correctly. The use of we for both what Brooks did and what you did makes it confusing.

If I remember correctly, what you have done is transform a manual process to an automated one. Maybe this is a point you make later?

You should make a list of the improvements to Drasil's work ~~issue~~ at the start of this discussion so that the reader can keep track as you discuss each improvement.

I suggest a subsection for each improvement and a subsection for the double pendulum example.

However, the previous research citation only could ~~completed~~ generat~~ing~~e a program for a single first-order ODE. Before ~~the~~ our changes, `ODEInfo` only ha~~s~~d an option~~s~~ to provide one 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 the Drasil Code Generator to work ~~generating problem~~ for higher-order ODE, the `ODEInfo` needs to store multiple initial values. For example, When we can convert a fourth-order initial value problem (IVP) ~~ODE~~ into a system of first-order ODE, we need four initial values. ~~To solve the system ODE as an IVP, we need four initial values.~~ Thus, 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 to store multiple initial values in a list.

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

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

Code 4.1: Source code for initial values

We also have to ensure that the 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 handl~~ing~~e a list of `data type` by matrix. In Code 4.2, the `matrix` can wrap a `[CodeExpr]` into a

```
-- Old
initVal info

-- New
matrix[initVal info]
```

Code 4.2: Source code for initial values in Code Generator

CodeExpr and the code initVal info retrieves the initVal from an ODEInfo data type. This change effects the generated code. For Python Scipy library, in Code 4.3, line 6 initialize the initial value with a list rather than one entity. The same thing happens in C# OSLO. In Code 4.4, line 5 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 for artifact in those two languages.

```
1 # Old
2   r.set_initial_value(T_init, 0.0)
3   T_W = [T_init]
4 
5 # New
6   r.set_initial_value([T_init], 0.0)
7   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.
```

What did you need to change in the generator to get generate the new version?

Code 4.3: Source code for initial values in Python

Allowing multiple initial values unlocks the potential for Drasil to generate a program that produce numerical solution for a system of first-order ODE. Every higher-order linear ODE has its equivalent system of first-order ODE, and the solution for the system of first-order ODE is also the solution for the higher-order ODE. The

```
// 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#

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~~ via the four selected external libraries.

Put this in its own subsection

Our focus to this point has been on linear ODEs. However,

Despite the Double Pendulum case study containing a ~~higher-order~~ non-linear ODE, the Drasil framework can still generate a program to solve it numerically. In the double pendulum case study, we ~~eventually~~ want to solve ~~Equation 4.2.1~~ the following equation:. There are two second-order ODEs in one system. To solve this system of ODEs we convert them into a system of first-order ODE. The transformation follows the methodology we discussed in Section 4.1. We transform 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 ($\mathbf{X}' = \mathbf{A}\mathbf{X} + \mathbf{c}$) because the ODE is not a linear ODE.

more eq. here

should be in italics bold form

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

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

$$
\begin{aligned}
&\theta_1' = \omega_1 \qquad (4.2.2)\\
&\theta_2' = \omega_2\\
&\omega_1' = \frac{-g(2m_1 + m_2)\sin\theta_1 - m_2 g \sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2) m_2(\omega 2^2 L_2 + \omega 1^2 L_1 \cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}\\
&\omega_2' = \frac{2\sin(\theta_1 - \theta_2)(\omega_1{}^2 L_1(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \omega_2{}^2 L_2 m_2 \cos(\theta_1 - \theta_2))}{L_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}
\end{aligned}
$$

Move this to the beginning of the section.

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

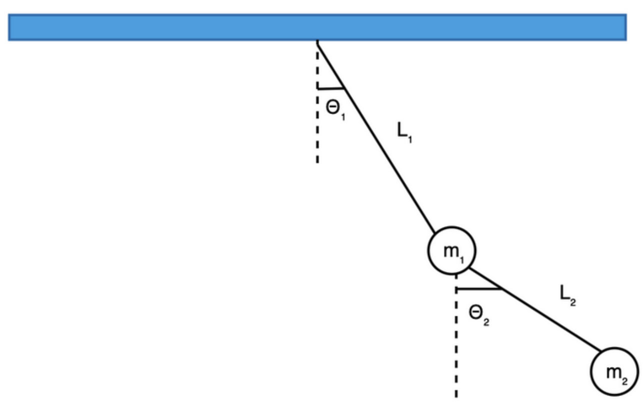

Figure 4.1: Double Pendulum Demonstration

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 the generated code for double pendulum ~~will show~~ are in Appendix A.4. However, the Double

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

Pendulum case study is unable to utilize any function introduced in the next section because they were 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 type. In the next section, we will discuss how to automate the transformation from a higher-order linear 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 to be as fully automatic~~ally~~ 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 can extract information from this structure and generate a form of the ODE ~~which~~ that selected

```
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

external libraries can utilize. Creating the `DifferentialModel` data structure satisfies the need of this idea. We can restructure an ODE based 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 ODEs.

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 reverse the order of $\mathbf{X}$ to x_n, ..., x_1. The coefficient matrix $\mathbf{A}$ is also changed, but $\mathbf{X}$' and $\mathbf{c}$ remain unchanged.

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

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' ~~**X**'~~ and ~~**X**~~ x in yellow ~~colour~~ in Equation 4.3.1. The number of elements in x' ~~**X**'~~ and x ~~**X**~~ depends on how many new dependent variables are introduce~~s~~d. 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' ~~**X**'~~, knowing it is $ $ n-th subscript order ODE, we parameterize x' from 1 to $ $ n. For x ~~**X**~~, knowing it is $ $ n-th the order ODE, we parameterize x from $ $ n to 1.

what do you mean by parameterize?

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

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

For ~~If it is~~ a fourth-order ODE, the **A** will be a~~n~~ 4×4 matrix. Equation 4.3.3 shows a completed transformation for a fourth-order ODE.

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

All variables should be in italic font, as in the equation. I'll stop highlighting them.

italic superscript

The orange part starts at the (n-1)th row with $[1, 0, \ldots]$. If there is a second row, we add $[0, 1, \ldots]$ above 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 generating 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 add all rows in the orange part together.

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 0s. 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 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.

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

```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)

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

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

Secondly, we ~~cancel out the~~ divide both side of the equation by the coefficient $a_n(t)$.

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

```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**

Then, this can be written in ~~a~~ matrix form as:

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

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, $\dots$, x_1.

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

Lastly, we replac~~ing~~e new variables in Equation 4.3.1, ~~we~~ To 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 blue highlighted. The `divideCosntants` cancels out the coefficient in the red highlight~~ed~~s.

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~~ producing a `DifferentialModel`.

In Chapter 3, we mentioned we would solve the ODE as an IVP. In Code 4.11, we create `InitialValueProblem` to store IVP-related information, includes initial time,

```
-- | 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

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 **A** (`[[Expr]]`), the unknown vector **X** (`[Unknown]`), and the constant vector **c** (`[Expr]`). Then, we form responsible expressions. The first row of **A** cross product the **X**, then we add all

?

(do you really mean cross product?

From Wikipedia

$\underline{a} \times \underline{b} = \|\underline{a}\|\|\underline{b}\| \sin\theta \, \underline{n}$ } I don't think you mean this.

Do you just mean a ~~col~~ row vect. times a column vector? This is actually the dot product.


```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, $\mathbf{X'} = \mathbf{AX} + \mathbf{c}$

terms together with a responsible constant term. In the follwoing 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 the 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 -> []
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