

Google ways to make the look better in LaTeX.

Chapter 3

External libraries

are all external libraries lang. dependent?

External libraries ^{are} come from an outside source ^{that} and do not originate from the source project. ^{Our current interest is for libraries that are used to support solving} and many of them can solve scientific problems. Most external libraries are

language-dependent, and the Drasil framework can generate five different languages:

including Python, Java, C++, C#, and Swift. Among those five languages, four

programming languages have ODE libraries for solving ODEs. In Python, Scipy Li-

brary (8) is a well-known scientific library for solving scientific problems, and has sup-

port for solving ODEs. In Java, a library ^{the} is called Apache Commons Maths (ACM) (2), ^{including}

^{provides a} and it is a supplementary library for solving mathematical and statistical problems, ^{space}

^{that not available in the} which build in Java programming language is unavailable. ^{ACM includes} It has support to solve

ODEs as well. Two less known libraries ^{to} can solve an ODE in ODEINT Library (4) ^{are}

ⁱⁿ C++ and OSLO Library (6) C#. ^{the} Last, we did not find a suitable library for Swift. For

those four selected libraries, they have some commonalities and variabilities. Firstly,

they all ^{have the commonality of providing} 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

say which one doesn't?

move earlier

New Paragraph

Some citations or commonality analysis would be good. You can find references in the pub repo. I suggest you start with Smith And Chen 2004, Smith E 2005, Wells 1997, Smith McCutcheon And Co 2007

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} my study. Last, ^{ly}Scipy and OSLO Library 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 the Drasil framework.

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 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

I'm not sure what you are saying here. Do you mean that the lessons learned with for ODEs can be extended to other libraries. That would be a good thing to say.

→ say how a BVP is different

of first-order ODE.

$$x_1(t)' = x_2(t) \quad (3.1.1)$$

$$x_2(t)' = -(1 + K_d)x_2(t) - (20 + K_p)x_1(t) + r_t K_p$$

In ~~Example~~ 3.1.1, there are two dependent variables, x_1 and x_2 . The $x_1(t)$ is the function of the independent variable, ~~often~~ ^{in this case} time, which produces dependent variables x_1 over time. The $x_2(t)$ is the function of the independent variable, ~~often~~ ^{over} time, which produces dependent variables x_2 over time. The x_1 is the process variable, and the x_2 is the first derivative with respect to the time of the process variable x_1 . The $x_1'(t)$ is the first derivative of the function $x_1(t)$, and the $x_2'(t)$ is the first derivative of the function $x_2(t)$. The K_d , K_p , and r_t are constant variables, and they ~~remain~~ ^{have} the same meaning in ~~Example 2.2.2 and example 3.1.1~~ ^{Example 3.1.1}. We can encode the ~~Example 3.1.1~~ ^{Example 3.1.1} in all four libraries. *as in* *throughout the examples also say the library you are using*

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

```
1 def f(t, y_t):
2     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 the 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, $x_2(t)' = -(1 + K_d)x_2(t) - (20 + K_p)x_1(t) + r_t K_p$, in the ~~Example~~ ^{Example} 3.1.1. In Java, we

can write the example as the following code:

```

1 public void computeDeriv(double t, double[] y_t, double[] dy_t) {
2     dy_t[0] = y_t[1];
3     dy_t[1] = -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0] + r_t
    ↪ * K_p;
4 }

```

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

```

1 void ODE::operator()(vector<double> y_t, vector<double> &dy_t,
    ↪ double t) {
2     dy_t.at(0) = y_t.at(1);
3     dy_t.at(1) = -(1.0 + K_d) * y_t.at(1) + -(20.0 + K_p) *
    ↪ y_t.at(0) + r_t * K_p;
4 }

```

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

```

1 Func<double, Vector, Vector> f = (t, y_t_vec) => {
2     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);
3 };

```

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~~ ^{an} 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~~ ^{specified}, including the start time, end time ~~of the simulation~~ ^{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~~ ^{Example 3.1.1} will ~~show up~~ ^{are shown} in Appendix A.2, code 3.1, and code 3.2.

3.2 Algorithm Options

~~In reality,~~ ^{with} we can solve an ODE in many algorithms, ~~and those four libraries provide~~ ^{the selected each} many algorithms. We roughly classified ^{by} 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 other methods. The commonality analysis ^{a "catchall" category of} on available algorithms is ^{we provide} an ~~approximation and incomplete~~ ^{incomplete}. ^{starting point.} Getting a complete ~~commonality analysis~~ ^{we provide} will

^{It is} ~~require some help from domain experts in the ODE area. It will be far out of my study,~~ ~~so we only provide a rough classification in this study.~~ 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 will show in Appendix A.3.

Last, 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 instead using 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. 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 ODE as a Function

The ODE provides a mathematical expression for describing a function of time. The numerical solution can instantiate an ODE, but it only provides a partial solution. If we are interested in the continuous change of the function, the numerical solution will not help. Instead of providing points, providing a function of time would be a better choice to represent the solution of ODE. In the Drasil framework, users have an option to generate a runnable program or a library. The runnable program will contain the main function so users can run generated code directly. If the generated code is a library, other outside programs can utilize the generated library by calling

<div>Library Algorithm</div>	Scipy-Python	ACM-Java	ODEINT-C++	OSLO-C#
Family of Adams	<ul style="list-style-type: none"> • Implicit Adams 	<ul style="list-style-type: none"> • Adams Bashforth • Adams Moulton 	<ul style="list-style-type: none"> • Adams Bashforth Moulton 	
Family of BDF	<ul style="list-style-type: none"> • BDF 			<ul style="list-style-type: none"> • Gear's BDF
Family of RK	<ul style="list-style-type: none"> • Dormand Prince (4)5 • Dormand Prince 8(5,3) 	<ul style="list-style-type: none"> • 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) 	<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • Dormand Prince RK547M
Others		<ul style="list-style-type: none"> • Gragg Bulirsch Stoer 	<ul style="list-style-type: none"> • Gragg Bulirsch Stoer 	

Table 3.1: Algorithms support in external libraries

it via ^{its} interfaces. Thus, by outputting the ODE as a function, other outside program can solve an ODE by calling the generated ODE library. Not all libraries have options to represent the ODE as a function. Among ~~four~~ ^{the} libraries, Scipy Library and OSLO Library have a similar option to support outputting a function. In the Scipy library, we can return a generic interface called `scipy.integrate.ode` (9) which is a generic interface that can store ODE's information. Later if we want to solve the ODE, we can add other ODE-related configuration information ~~afterward~~ ^{selected}. The code 3.1 shows the full details of how to solve the ~~Example~~ ^{Example} 3.1.1 in the Scipy library.

```

1  def func_y_t(K_d, K_p, r_t, t_sim, t_step):
2      def f(t, y_t):
3          return [y_t[1], -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) *
4                  ↪ y_t[0] + r_t * K_p]
5
6      r = scipy.integrate.ode(f)
7      r.set_integrator("dopri5", atol=Constants.Constants.AbsTol,
8          ↪ rtol=Constants.Constants.RelTol)
9      r.set_initial_value([0.0, 0.0], 0.0)
10     y_t = [[0.0, 0.0][0]]
11     while r.successful() and r.t < t_sim:
12         r.integrate(r.t + t_step)
13         y_t.append(r.y[0])
14
15     return y_t


```


Code 3.1: Source code of solving PDController in Scipy

Between line 2 and line 3, we encode the ODE equation of the example 3.1.1 in a list. In line 5, we call `scipy.integrate.ode` to pack ODE information in the generic interface. 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 `scipy.integrate.ode(f)` capture the information of the ODE. We can output the class `scipy.integrate.ode` as a function of time in a generated library. If there is an outside source that wants to use this function, they can write a proper interface to connect the function with their programs.

In the OSLO library, the function `Ode.RK547M` returns an enumerable sequence of solution points, and it is an endless sequence of solution points. The endless points contains all numerical solutions of the ODE, and we can use it to derive a partial numeric solution. Later, we can call `SolveFromToStep` to return a list of points based on start time, end time, and time interval. We can treat the return of `Ode.RK547M` as a function of time. The code 3.2 shows the full details of how to solve the  example 3.1.1 in the OSLO library.

In code 3.2, 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 points. In line 12, we use `SolveFromToStep` to get a partial solution 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)`

```
1 public static List<double> func_y_t(double K_d, double K_p, double  
  ↪ r_t, double t_sim, double t_step) {  
2     List<double> y_t;  
3     Func<double, Vector, Vector> f = (t, y_t_vec) => {  
4         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);  
5     };  
6     Options opts = new Options();  
7     opts.AbsoluteTolerance = Constants.AbsTol;  
8     opts.RelativeTolerance = Constants.RelTol;  
9  
10    Vector initv = new Vector(new double[] {0.0, 0.0});  
11    IEnumerable<SolPoint> sol = Ode.RK547M(0.0, initv, f, opts);  
12    IEnumerable<SolPoint> points = sol.SolveFromToStep(0.0, t_sim,  
  ↪ t_step);  
13    y_t = new List<double> {};  
14    foreach (SolPoint sp in points) {  
15        y_t.Add(sp.X[0]);  
16    }  
17  
18    return y_t;  
19 }
```

Code 3.2: Source code of solving PDController in OSLO

captures the information of the ODE, and the return object represents a total solution of the ODE. Anyone who is interested in a partial solution can use `SolveFromToStep` to filter out. Therefore, we can output the result of `Ode.RK547M` as a function of time in a generated library. If there is an outside source that wants to use this function, they can write a proper interface to connect the function with their programs.

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. It is not practical because of the amount of space external libraries occupy. Moreover, external libraries do not share across study cases, and each study case will have its own copy of external libraries. This research has uncovered the current way of handling dependencies in the Drasil framework is problematic, and the team would like to find a better way to handle dependencies. We used a temporary solution, symbolic links, to share external libraries without duplications. By creating a symbolic link file, external libraries become sharable.

The team will conduct further studies to tackle this problem.

In the future,

I think the difference between ext lib that output sequences of IR versus output functions can be made clearer with type info. I'll give a link with Kirk's hints in the issue tracker.