An Introduction to BioCro for Those Who Want to Add Models

Justin McGrath

July 28, 2020

1 Plant growth as a system of differential equations

1.1 Overview

BioCro is used to calculate aspects of plant growth, such as the change in the mass of a plant, given aspects of a plant and its environment that are already known. For example, one can calculate leaf and stem mass over thermal time given measures of the climate (Figure 1).

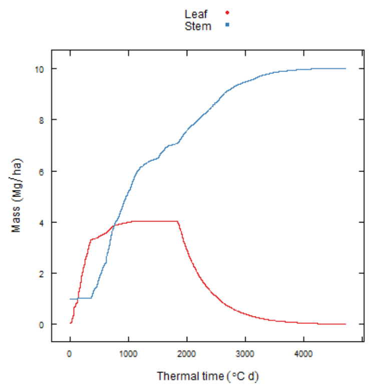


Figure 1: Mass over time of willow.

BioCro is designed to reflect differential equation models. In this section, we present some of the terminology and notation we will use to describe such models. Readers already familiar with such models may want to skim though this section and then move on to section 1.2.

We'll call the set of all of the variables in the model (mass, temperature, wind speed, etc.) the *state*. More precisely, a *state* is described by the set of values assumed by these variables *at some particular moment*. We want to calculate a sequence of states—a series of snapshots of the system being modelled as it evolves over some period of time. We'll denote the system comprising these variables by $\mathbf{X}$ and the state of that system at some specific time $t = t_i$ by $\mathbf{X}_{t_i}$. (Here, t_i denotes the i'th instant of time (counting from 0) in some sequence $a = t_0 < t_1 < t_2 < \cdots < t_n = b$ of times, where $[a, b]$ is the time interval of interest.)

Some parts of the state are taken as known for the entire period, and we'll denote this component of the system as $\mathbf{X}_{\mathsf{K}}$ (K for known) and denote the known portion of the state at time $t = t_i$ as $\mathbf{X}_{t_i,\mathsf{K}}$. These values that are known beforehand are inputs of the model, and in the literature, people typically say that these variables "drive" the model.

Some state variables can be calculated from other state variables without explicit dependence on time. For example, the total mass of the plant is the sum of leaf, stem, and root masses. This set of variables we'll denote as $\mathbf{X}_{\text{S}}$ (S for secondary variable).[1]

Other variables must be calculated from their rate of change. For example, the rate of change of leaf mass is calculated from the photosynthetic rate, so the leaf mass at 10 a.m. is the leaf mass at 9 a.m. plus the rate of change in units of mass per hour times one hour; that is, $\text{mass}_{10\,\text{AM}} = \text{mass}_{9\,\text{AM}} + \frac{d\text{mass}}{dt} * 1\,\text{h}$.[2][3] The variables we calculate from rates of change we'll denote as $\mathbf{X}_{\text{CD}}$ (CD for calculated from differential equations).

Analogously to writing $\mathbf{X}_{t_i}$ to denote the state of the system $\mathbf{X}$ at time $t = t_i$, the "CD" component of this state will be denoted $\mathbf{X}_{t_i,\text{CD}}$. Since in general, each $\mathbf{X}_{t_i,\text{CD}}$ (for $i > 0$) depends on $\mathbf{X}_{t_{i-1}}$ (the state at time t_{i-1}), an initial value $\mathbf{X}_{t_0,\text{CD}}$ of the CD component of the state must be given as input to the model.[4]

We'll denote the function that describes how to calculate secondary variables ($\mathbf{X}_{t,\text{S}}$) from the other variables as $\mathbf{g}$. Thus, at any particular time t_i, $\mathbf{X}_{t_i,\text{S}} = \mathbf{g}(\mathbf{X}_{t_i,\text{K}} \cup \mathbf{X}_{t_i,\text{CD}})$.[5] Note that, as this equation shows, $\mathbf{X}_{t_i,\text{S}}$, the "S" component of the state at time t_i, depends only on the values of the variables in the other components of the state at time t_i.

As for the variables in the $\mathbf{X}_{\text{CD}}$ component of $\mathbf{X}$, we can calculate the value of their derivatives with respect to time—their rate of change—at any particular time t_i from some or all of the variable values that comprise the totality of the state $\mathbf{X}$ at time t_i. We'll use $\mathbf{h}$ to denote the function that yields the derivative of the $\mathbf{X}_{\text{CD}}$ component of the state at any time t_i given the totality of the state at time t_i. Thus, $\frac{d\mathbf{X}_{\text{CD}}}{dt}(t_i) = \mathbf{h}(\mathbf{X}(t_i))$[6]

The model can be solved by iterating through the following process for each time point t_i starting with time t_0:

1. Use the function $\mathbf{g}$ to calculate the value of the secondary variables at time t_i from the values at time t_i of the variables in K and CD.[7]

2. We now have full knowledge of the three components—the known variables, the secondary variables, and the variables that depend on differential equations—that comprise the full state $\mathbf{X}_{t_i}$ at time t_i.

[1] These were called "steady state" variables in a previous version of this document, and as of this writing, that terminology is still reflected in the BioCro software—both in the naming of variables and types of modules, and in the names of function parameters. It was felt, however, that this was a somewhat confusing appropriation of a term that usually means "unvarying over time". A possible alternative name was "intermediate variable", since a primary use of these variables is as convenience variables used in the calculating of derivatives. But they are also useful program output in their own right, so "intermediate" didn't seem entirely appropriate.

[2] Here, $\frac{d\text{mass}}{dt}$ is the *average* rate of change of mass over the period from 9 a.m. to 10 a.m. When using the forward Euler method, the derivative at the beginning of the time interval (at 9 a.m.) is taken as a reasonable approximation of this value. Other numerical methods used by BioCro to estimate how the state changes are more complex, but all involve the equations giving the *rate of change* of the variables in the state at a given time t based on the *value* of the variables in the state at time t.

[3] We have used time units of hours here, but it is a goal to eventually use only SI units within BioCro. The SI base unit of time is the second, not the hour.

[4] $\mathbf{X}_{t_0} = \mathbf{X}_{t_0,\text{K}} \cup \mathbf{X}_{t_0,\text{S}} \cup \mathbf{X}_{t_0,\text{CD}}$, but since $\mathbf{X}_{t_i,\text{K}}$ is assumed to be known for all times t_i (including t_0) and since $\mathbf{X}_{t_0,\text{S}}$ can be computed from $\mathbf{X}_{t_0,\text{K}} \cup \mathbf{X}_{t_0,\text{CD}}$, the only remaining missing piece is $\mathbf{X}_{t_0,\text{CD}}$.

[5] Denoting the set of known variables by K, the set of variables calculated from differential equations by CD, and the *number* of variables in K, CD, and S by $|\text{K}|$, $|\text{CD}|$, and $|\text{S}|$ (respectively), then $\mathbf{g}$ is a vector-valued vector function from the $(|\text{K}| + |\text{CD}|)$-dimensional Euclidean space whose axes are labelled by the variables of K and CD to the $|\text{S}|$-dimensional Euclidean space whose axes are labelled by the variables of S.

Alternatively, we can think of $\mathbf{g}$ as a collection of functions $\{g_v : v \in \text{S}\}$ where each g_v maps a collection of values for the variables in K and CD to a value for the variable v. These functions g_v *almost* correspond to what we currently call *steady-state* modules in the BioCro software. There are two ways in which they might differ, however. First, these modules may compute values for two or more variables rather than just one. A module which calculates the values of variables u and v, for example, would correspond to a function whose range has dimension two and which is defined by the rule $\mathbf{x} \mapsto (g_u(\mathbf{x}), g_v(\mathbf{x}))$. The second way in which a module may differ from a function g_v is that it may, for convenience, take as input previously-computed values of other variables in S. While in theory, each steady-state module could be restricted to use only variables in $\text{K} \cup \text{CD}$ as input, in practice this would sometimes involve repetitious calculation.

[6] The right-hand side could be written as $\mathbf{h}(\mathbf{X}_{t_i})$; the two expressions $\mathbf{X}(t_i)$ and $\mathbf{X}_{t_i}$ essentially designate the same thing. The connotations may be slightly different though. Using $\mathbf{X}(t_i)$ emphasizes that $\mathbf{X}$ is a state function, and when we write $\mathbf{X}(t_i)$, we are evaluating that function at time t_i. Writing $\mathbf{X}_{t_i}$ emphasizes that we are dealing with the state yielded by that evaluation.

For the left-hand side, another commonly used notation is $\left.\frac{d\mathbf{X}_{\text{CD}}}{dt}\right|_{t=t_i}$.

[7] Recall that values of the variables in CD are assumed to be known at time t_0. For $i > 0$, the values of the variables in CD at time t_i are calculated in step 4 of the previous iteration.

3. Use the function $\mathbf{h}$ to calculate the derivatives (rates of change) at time t_i of the variables in $\mathbf{X}_{\mathsf{CD}}$.
4. Use the rates of change to calculate new values (that is, the values at time t_{i+1}) for the variables CD that depend on differential equations.

This process is described somewhat more formally in the next section.

1.2 Mathematical summary

1.2.1 Model inputs

The inputs to the model are the following:[8]

$\mathbf{X}_{\mathsf{K}}$:	The component of the system given as known for the entire simulation period.
$\mathbf{X}_{0,\mathsf{CD}}$:	The initial values of those variables calculated using differential equations.
$\mathbf{g}$:	A function for obtaining the values $\mathbf{X}_{t,\mathsf{S}}$ from those of $\mathbf{X}_{t,\mathsf{K}}$ and $\mathbf{X}_{t,\mathsf{CD}}$ for any given time t.
$\mathbf{h}$:	A function for obtaining the *derivatives* of the variables in $\mathbf{X}_{\mathsf{CD}}$ from the values $\mathbf{X}_t$ for any given time t.

Table 1: Inputs to the model

1.2.2 Model equations

Whereas in the real world the state function $\mathbf{X}$ is a continuous function of time on some time interval of interest $[t_0, t_n]$, in practice we consider only the value of $\mathbf{X}$ for some finite monotonically increasing sequence of points of time t_0, t_1, t_2, ..., t_n within that interval. This is both because the so-called "known" variables are known only at some finite set of instants in that interval and because it is a requirement of the numerical methods used to solve the differential equations. The model can be solved using Euler's method[9] by iterating through the following process for each $t = t_i$ starting at $t = t_0$:

$$\begin{aligned} \mathbf{X}_{t_i,\mathsf{S}} &= \mathbf{g}(\mathbf{X}_{t_i,\mathsf{K}} \cup \mathbf{X}_{t_i,\mathsf{CD}}) \\ \mathbf{X}_{t_i} &= \mathbf{X}_{t_i,\mathsf{K}} \cup \mathbf{X}_{t_i,\mathsf{S}} \cup \mathbf{X}_{t_i,\mathsf{CD}} \\ \frac{d\mathbf{X}_{\mathsf{CD}}}{dt}(t_i) &= \mathbf{h}(\mathbf{X}(t_i)) \\ \mathbf{X}_{t_{i+1},\mathsf{CD}} &= \mathbf{X}_{t_i,\mathsf{CD}} + \frac{d\mathbf{X}_{t_i,\mathsf{CD}}}{dt} \times \Delta t \end{aligned} \tag{1}$$

Here, $\Delta t = t_{i+1} - t_i$. In general, it is assumed that the instants t_0, t_1, t_2, ..., t_n are equally spaced so that Δt is of fixed size.

[8]Strictly speaking, it is probably more accurate to call only $\mathbf{X}_{\mathsf{K}}$ and $\mathbf{X}_{0,\mathsf{CD}}$ *inputs* to the model, and say that $\mathbf{g}$ and $\mathbf{h}$ *define* the model. Here, we are somewhat anticipating the terminology of the BioCro software where all four items are input parameters to a solver function that computes the output of the model. (See section 1.3.1.)

[9]Other generally better methods for solving systems are available in BioCro, but in the discussion here, we shall stick to Euler's method so as not to overly complicate the presentation.

1.3 Relating the mathematics to the program code

1.3.1 Function inputs

The R function `Gro_solver()` accepts five parameters that correspond to the model inputs given in Table 1. For convenience, $\mathbf{X}_{\text{K}}$ is separated into variables that do or do not vary over the simulation period.[10]

Gro_solver() input	model equivalent
`initial_values`	$\mathbf{X}_{t_0,\text{CD}}$
`parameters`	$\mathbf{X}_{\text{K}}$ that do not vary with time.
`varying_parameters`	$\mathbf{X}_{\text{K}}$ that do vary with time.
`steady_state_module_names`	$\mathbf{g}$
`derivative_module_names`	$\mathbf{h}$

Table 2: `Gro_solver`'s inputs

State variables are represented as a paired name and value, for example ("Leaf", 10). In programming parlance, this is called a key-value pair; here "Leaf" is the key and "10" is the value. In R, the data types used to represent collections of such pairs are `list` (more specifically, a `list` with named components) and `data.frame`.[11] For example, if `CD` consists of a variable each for *stem* and *leaf* biomass, then to specify initial values (that is, $\mathbf{X}_{t_0,\text{CD}}$) one could use the following:

```
# The list() function takes any number of key=value pairs, separated
# by commas.
> example_initial_values = list(Stem = 3, Leaf = 5)

# The str() function prints useful information about any object.
> str(example_initial_values)
List of 2
 $ Stem: num 3
 $ Leaf: num 5

# You can get a value using the key and the '$' operator ...
example_initial_values$Leaf
[1] 5

# ... or with the double-bracket operator '[[' operator and the
# string value of the key.
> example_initial_values[["Leaf"]]
[1] 5
> key_variable <- "Leaf"
> example_initial_values[[key_variable]]
[1] 5
```

Lists are also used to specify values for the `parameters` argument.[12]

[10]The "unvarying" parameters are probably more properly viewed as parameters of the equations that make up $\mathbf{g}$ and $\mathbf{h}$ rather than being considered to be part of the $\mathbf{X}_{\text{K}}$ component of the state function. Some of them in fact are physical constants and so shouldn't be viewed as parameters at all and are only a part of the *state* of the system in the most metaphysical of senses. But in the programmatic implementation of the model, it proves useful to treat them as components of the state.

[11]In the underlying C++ code, the corresponding structure is called a *map*. In mathematics, both "map" and "function" are used. In our case, for example, the initial state $\mathbf{X}_{t_0}$ is a mapping from the set of variable names to their values at time t_0.

[12]The `Gro` function also uses an R `list` to specify the modules to be used. The keys in this case are a fixed set of six module types,

Lists of parameters and modules are provided for sorghum, miscanthus, and willow and are named using names of the form *cropname*`_initial_state`, *cropname*`_parameters`, and *cropname*`_modules`;[13] for example, `willow_initial_state`:

```
> str(head(willow_initial_state)) # head truncates the list to six items
List of 6
 $ Rhizome  : num 0.99
 $ Leaf     : num 0.02
 $ Stem     : num 0.99
 $ Root     : num 1
 $ Grain    : num 0
 $ waterCont: num 0.32
```

`varying_parameters`, since it is made up of variables whose value changes over the course of time, must be specified somewhat differently. In particular, the *values* of the key-value pairs comprising `varying_parameters` will be vectors rather than single values, and the number of elements in these vectors will correspond to the number of time points t_0, t_1, t_2, ..., t_n being sampled. Moreover, in order to correlate the values in these vectors to particular points of time, vectors specifying the time must be included, and the time should be a monotonically-increasing function of the vector index (in other words, the time values should be in “chronological order”).[14]

In the following example, the time is specified using vectors labelled *year*, *doy* (“day-of-year”), and *hour*.

```
> example_varying_parameters = data.frame(
              year = c(2005, 2005),
              doy = c(1, 1),
              hour = c(0, 1),
              solar = c(0, 0),
              temp = c(4.04, 3.03))
> print(example_varying_parameters)

  year doy hour solar temp
1 2005   1    0     0 4.04
2 2005   1    1     0 3.03
```

Data frames of weather data are provided to pass to `varying_parameters`. These are typically for one year (January 1 to December 31) and should be subsetted to include only the period of growth. The function `get_growing_season_climate()` is provided as one means of subsetting climate data.[15] Here is a display of weather data for 2005 showing the first few rows of the growing season:

with the corresponding value naming an appropriate choice of module for the type specified. The `Gro` function parses out the module names into two R *character* vectors, one containing the names of “steady-state” modules, the other names of “derivative” modules. These are then passed to `Gro_solver`.

[13]Such a list of modules is intended to be used with the `Gro` function; see the previous footnote.

[14]As of this writing, the variable representing time that is most directly used in calculations is called `doy_dbl`. The integral portion of a `doy_dbl` value represents the day-of-year (that is, 1 = January 1 and 365 = December 31 (or December 30 if it is a leap year)) and the fractional portion represents the hour of the day. For example, 73.5 would represent 12-noon on the 73rd day of the year. Note that this means that `doy_dbl` represents the fractional number of days from the first instant of the new year *plus one*. In other words, the `doy_dbl` values for January 1 run from 1.0 up to 2.0 rather than from 0 up to 1.0 so that 1.5 represents noon on January 1. Also, no allowance is made for switching to daylight-savings time.

The `doy_dbl` variable may be provided by the user directly, but usually, the user will supply separate vectors `doy` (for day-of-year) and `hour`, and the `doy_dbl` component of `varying_parameters` will be computed and added by the software on the fly.

Currently, the `year` component, if given, is for informational purposes only and does not figure in the computation of `doy_dbl`. This means that no provision is given for running systems with input data spanning multiple years except in a kind of hackish way involving supplying `doy_dbl` directly and allowing values for it above 366.

[15]The `get_growing_season_climate()` function requires its argument to have a `doy` component as well as a `temp` (temperature) component.

```
> head(get_growing_season_climate(weather05))
     year doy hour solar    temp     rh windspeed precip
2953 2005 124    0     0  3.5972 0.3235    0.7603      0
2954 2005 124    1     0  1.3938 0.2496    0.7603      0
2955 2005 124    2     0 -0.2969 0.1928    0.7603      0
2956 2005 124    3     0 -1.3597 0.1572    0.7603      0
2957 2005 124    4     0 -1.7222 0.1450    0.7603      0
2958 2005 124    5     0 -1.3597 0.1572    0.7603      0
```

1.3.2 Function output

The output of `Gro_solver` (and of `Gro`) is a data.frame, which may be viewed as a table having one column for each variable in `initial_values` as well as columns for the time variables `doy` (day of year), `hour`, and `doy_dbl`. (Other columns may be included as well.) The table has one row for each row in `varying_parameters`, that is, for each point of time for which there is input data.

```
     year doy hour   Stem   Root    Leaf
   1 2005   1    0  0.990   1.00   0.020
   2 2005   1    1  0.990   1.00   0.020
   3 2005   1    2  0.990   1.00   0.021
   4 2005   1    3  0.990   1.00   0.022
...
8759 2005 365   22 10.016   2.14   9e-08
8760 2005 365   23 10.016   2.14   9e-08
```

Table 3: A truncated listing of the output used to produce Figure 1

1.3.3 An example

In R, you can use the `Gro()` function to simulate the development of a crop as follows:

```
library(BioCro)
library(lattice)  # This is a package that creates figures.

result = Gro(sorghum_initial_state, sorghum_parameters,
             get_growing_season_climate(weather05), sorghum_modules)
xyplot(Stem + Leaf + Root ~ TTc, data=result)  # The output is not shown here.
```

2 Modifying the BioCro code

2.1 Organization of the source files

BioCro is provided as a package for R. The package subdirectories containing the source code are *R* for R code and *src* for C/C++ code.

To understand the organization of the code, it is necessary to know a little about the data types in R and how the R environment accesses compiled C/C++ code.

R provides C libraries that allow R code to call compiled code using C data types specific to the R environment. Here, these libraries will be called R-to-C libraries.[16] As an example, in R there is a `numeric` type that represents real numbers. The closest equivalent to this in C is the `double` type.

As an example, we show how to write and compile a C function that squares its argument and how to call that function from within R:

Contents of file `squarer.c`

```
/* Rinternals.h contains the bulk of the R-to-C library definitions: */
#include <Rinternals.h>

SEXP my_function(SEXP x) {
    double new_x = REAL(x)[0];
    SEXP result;
    PROTECT(result = Rf_allocVector(REALSXP, 1));
    REAL(result)[0] = new_x * new_x;
    UNPROTECT(1);
    return result;
}
```

To compile this,[17] one may then run

```
R CMD SHLIB squarer.c
```

This will produce the library file `squarer.so`.

Then to use this from R, open an R session and type the following commands:

```
> dyn.load("squarer.so") # make the C function available to R
> .Call("my_function", pi) # call my_function from R with argument pi
[1] 9.869604
```

The `SEXP` data type is provided by the R-to-C libraries and can accommodate any of the data types used in the R environment (for instance, numeric or character). The author of the C function must know what data type is intended to be passed to the function. In this example, the `REAL` macro is used to convert `x` to an array of `double`s, and since there is only one element in the array, the first index (`0`)[18] is accessed. `PROTECT()`, `UNPROTECT()`, and `Rf_allocVector()` are also required and are provided by the R-to-C libraries, but understanding them is not necessary here.

Using the R-to-C libraries is tedious and error prone, and it is extremely easy to write code that will run but produce hard-to-spot errors.[19] The use of the R-to-C libraries should be limited, and they are not necessary at all to add new models to BioCro. The libraries are described here only to fully understand the organization of the code in the R package.

To sequester code that uses the R-to-C libraries, the code is conceptually organized into three groups: R code, R-to-C code, and C/C++ code. R code is in the *R* directory. R-to-C code is contained in files that have names that start with "`R_`"; these files and the files containing C/C++ code are in the *src* directory.

[16] In despite of the appellation "R-to-C", keep in mind that data conversion happens in both directions: R input data types are converted to C data types to use as C/C++ function input. The C data types returned by these functions are then converted to R data types suitable for R function output.

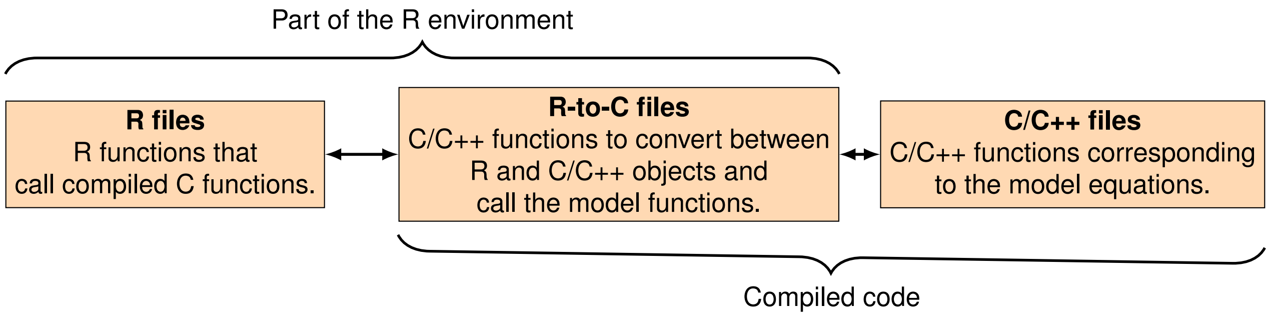
[17] If this were to be compiled as a C++ program (as the BioCro source code is), one would have to wrap the `my_function` definition with `extern "C" { ... }`.

[18] Note that in R arrays begin with index 1 whereas C arrays begin with index 0.

[19] A newer, purportedly better, R-C++ interface called Rcpp exists, but BioCro doesn't currently use it.

The functions that define the model—the functions that make up the vector-valued functions $\mathbf{g}$ and $\mathbf{h}$—and the code that iterates through the the equations in equation set 1 to solve the system being modelled are written in C++ and are thus contained in files in the *src* directory (or one of its subdirectories). The functions that implement $\mathbf{g}$ and $\mathbf{h}$ should be designed so that they do not rely on the R-to-C libraries in any way. Such a design helps prevent mistakes from the error-prone R-to-C libraries and allows the functions to be used without R—in a stand-alone C++ application, for instance.

Figure 2: The R-to-C libraries provide an interface between R scripts and compiled code. The files are organized so that the R-to-C libraries are not mixed with the models. Model equations should only be written in the C/C++ code.


Part of the R environment
R files
R functions that call compiled C functions.
R-to-C files
C/C++ functions to convert between R and C/C++ objects and call the model functions.
C/C++ files
C/C++ functions corresponding to the model equations.
Compiled code

R code should be written so that it only checks validity of arguments and calls R-to-C functions. R-to-C code should only provide error checking and call C/C++ functions. That is, R and R-to-C functions should only provide a way to access the models written in C/C++, and no modeling should be done in R or R-to-C code.

2.2 Adding new models[20]

[This section is outdated and is in the process of being rewritten.]

The C++ code is designed so that the functions have notation similar to the mathematical model. That is, they look like $\frac{d\mathbf{X}_{t,\text{CD}}}{dt} = \mathbf{h}(\mathbf{X}_t)$. In BioCro, functions that implement $\mathbf{h}(\mathbf{X}_t)$ are called modules.

As in the R code, both $\frac{d\mathbf{X}_{t,\text{CD}}}{dt}$ and $\mathbf{X}_t$ are represented as sets of key-value pairs. The data type used for this in C++ is a `state_map`. As an example, to model leaf growth rate as half of canopy assimilation rate going to leaves, the following code would be used:

```
state_map simple_leaf_growth::do_operation(state_map const &s) const
    state_map partial_rates_of_change;
    partial_rates_of_change["Leaf"] = s.at("Assim") * 0.5;
    return partial_rates_of_change;
}
```

The function is named `do_operation` and is part of the `simple_leaf_growth` model. It accepts a `state_map` named `s` and returns a `state_map` named `partial_rates_of_change`. The current state of the model ($\mathbf{X}_t$), is passed in as `s`. The `const` keywords are required, but do not need to be understood here. Canopy assimilation rate is calculated in a different part of the model, and it is stored in `s` with the name

[20]Up until now, we have used the term *model* to refer to the system as a whole, and the dynamics of how it changes over time. Here we are using it in a more specialized sense: we model the relationship between attributes of a state—how the value of some select group of attributes of the state determine the value(s) or the rate of change of the value(s) of other attributes of the state. A model of the dynamics of soil evaporation, for instance, is a component of the model of the system as a whole.

`Assim`. To access values in a `state_map`, use the `at()` function. To assign a value to parameters within a `state_map`, use the `[]` operator. Here, half of the assimilation rate is added to the partial rate of change of leaf growth. Note that the call statement has the same form as $\frac{d\mathbf{X}_{t,\text{CD}}}{dt} = \mathbf{h}(\mathbf{X}_t)$:

```
rates_of_change = simple_leaf_growth(current_state);
```

A separate module can be written to describe loss of leaf mass. To describe leaf loss as a fraction of the current leaf mass, the following model could be used:

```
state_map fractional_leaf_loss::do_operation(state_map const &s) const
    state_map partial_rate_of_change;
    partial_rate_of_change["Leaf"] = -s.at["Leaf"] * 0.01;
    return partial_rate_of_change;
}
```

The addition of the partial rates is handled elsewhere in the code, and does not need to be handled when writing new modules. This design allows one to write modules that operate independently, so that one does not need to know how the entire model works in order to modify a specific aspect of the model. To write a new module, one needs to know only what parameters are defined in the state ($\mathbf{X}$), which is a relatively short list.