# An Introduction to BioCro for Those Who Want to Add Models

Justin McGrath

May 14, 2021

## Contents

# 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).

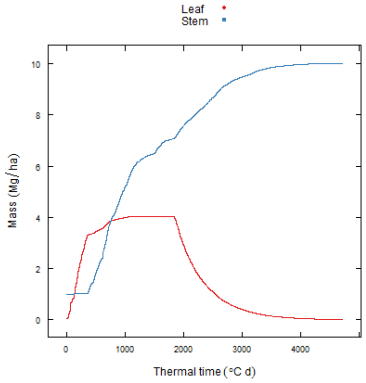



Figure 1: Mass over time of willow.

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 $\mathbf{X}_{\mathrm{K}}$ ($\mathrm{K}$ for known) and denote the known portion of the state at time $t = t_i$ as $\mathbf{X}_{t_i,\mathrm{K}}$. 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 $\mathbf{X}_{\mathrm{S}}$ ($\mathrm{S}$ 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

[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.

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] 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.[3]

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 $t_i$, $\mathbf{X}_{t_i,\text{S}} = \mathbf{g}(\mathbf{X}_{t_i,\text{K}} \cup \mathbf{X}_{t_i,\text{CD}})$.[4] 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, $\frac{d\mathbf{X}_{\text{CD}}}{dt}(t_i) = \mathbf{h}(\mathbf{X}(t_i))$[5]

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.[6]
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 $t_i$.
3. Use the function $\mathbf{h}$ to calculate the derivatives (rates of change) at time $t_i$ of the variables in $\mathbf{X}_{\text{CD}}$.
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

[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 $t$.

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.

[3] $\mathbf{X}_{t_0} = \mathbf{X}_{t_0,\text{K}} \cup \mathbf{X}_{t_0,\text{S}} \cup \mathbf{X}_{t_0,\text{CD}}$, 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 $\mathbf{X}_{t_0,\text{CD}}$.

[4] 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.

[5] 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 $\left.\frac{d\mathbf{X}_{\text{CD}}}{dt}\right|_{t=t_i}$.

[6] 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.

### 1.2.1 Model inputs

The inputs to the model are the following:[7]

$\mathbf{X}_{\mathrm{K}}$: The component of the system given as known for the entire simulation period.
$\mathbf{X}_{0,\mathrm{CD}}$: The initial values of those variables calculated using differential equations.
$\mathbf{g}$: A function for obtaining the values $\mathbf{X}_{t,\mathrm{S}}$ from those of $\mathbf{X}_{t,\mathrm{K}}$ and $\mathbf{X}_{t,\mathrm{CD}}$ for any given time $t$.
$\mathbf{h}$: A function for obtaining the *derivatives* of the variables in $\mathbf{X}_{\mathrm{CD}}$ 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 by iterating through the following process for each $t = t_i$ starting at $t = t_0$:[8]

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

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 $\Delta t$ is of fixed size.

## 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, $\mathbf{X}_{\mathrm{K}}$ is separated into variables that do or do not vary over the simulation period.[9]

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

[7] Strictly speaking, it is probably more accurate to call only $\mathbf{X}_{\mathrm{K}}$ and $\mathbf{X}_{0,\mathrm{CD}}$ *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.)

[8] 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.

[9] 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 $\mathbf{X}_{\mathrm{K}}$ 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.

| 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

collections of such pairs are `list` (more specifically, a `list` with named components) and `data.frame`.[10] 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.[11]

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`;[12] for example, `willow_initial_state`:

[10] 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 $t_0$.

[11] 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, 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`.

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

```
str(head(willow_initial_state)) # head truncates the list to six items

## List of 6
##  $ Grain        : num 0
##  $ Leaf         : num 0.02
##  $ leafdeathrate: num 5
##  $ LeafLitter   : num 0
##  $ Rhizome      : num 0.99
##  $ RhizomeLitter: num 0
```

`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").[13]

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.[14] Here is a display of weather data for 2005 showing the first few rows of the growing season:

```
head(get_growing_season_climate(weather05))

##      year doy hour solar       temp        rh windspeed precip
## 2953 2005 124    0     0  3.5972222 0.3235000 0.7602778      0
## 2954 2005 124    1     0  1.3938362 0.2495629 0.7602778      0
```

[13] 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.

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

```
## 2955 2005 124    2     0 -0.2968814 0.1928289 0.7602778      0
## 2956 2005 124    3     0 -1.3597109 0.1571645 0.7602778      0
## 2957 2005 124    4     0 -1.7222222 0.1450000 0.7602778      0
## 2958 2005 124    5     0 -1.3597109 0.1571645 0.7602778      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.[15] 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.

[15] 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.

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, one may then run

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

This will produce the library file `squarer.so`.[16]

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`) is accessed.[17] `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.[18] 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.

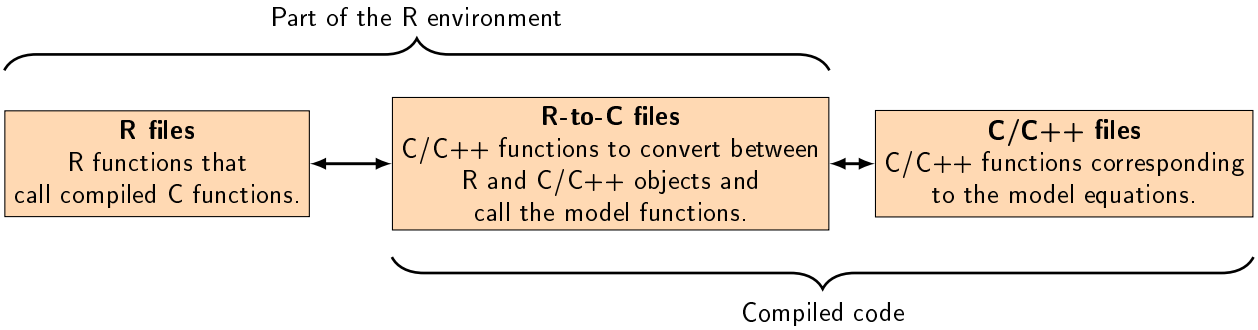
The functions that define the model—the functions that make up the vector-valued functions $\mathbf{g}$ and $\mathbf{h}$—and the code that iterates through the the equations in equation set 1 to solve the system being modelled are written in C++ and are thus contained in files in the *src* directory (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.

[16] 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"{ ... }`.

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

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

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.




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[19]

What follows are step-by-step instructions for writing modules and integrating them into the BioCro library. We show instructions both for writing steady-state modules and for writing derivative modules. In each case, we include a running example; the actual code changes needed are shown in shadowboxes at the end of each step. (Full listings of the finished code are given in the appendix.)

Following this, we show instructions for writing tests of the new modules.

### 2.2.1 Example: Steps for creating a steady-state module.

Suppose we want to write a module which builds upon the `solar_zenith_angle` module. That module, when given the latitude, day of the year, and hour of the day as inputs, computes the cosine of the solar zenith angle. We would like to use the cosine of the solar zenith angle to calculate the actual zenith angle in degrees. So we shall write a module taking one input, "cosine_zenith_angle", and computes the value of the angle itself. The steps are as follows.

1. Decide on a name for the module. While in theory, the name could be anything (ideally, something giving a concise description of the module), since we would like to follow the convention that the module name doubles as the name of the class that implements the module, the name (without quotes) should be a valid C++ identifier.[20]

[19] Up until now, we have (mostly) 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 values of some select group of attributes of the state determine the values or the rate of change of the values 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. When we are being careful, we will use the term *module* to refer to the code (or the C++ class) that implements a *model* (in this more restricted sense), but at times, we may use the two terms interchangably.

[20] The quoted name is the "public" name: when you specify the module in a BioCro R function, or if you use the BioCro C++ library in your own C++ code, it is the string name—the quoted name—that you use.

A valid C++ identifier must consist only of letters, digits and underscores; it cannot begin with a digit or match one of the C++ keywords. Additionally, to avoid conflicting with names reserved by the implementation, it should not begin with an underscore, and it should not contain a sequence of two or more underscores or an underscore followed by a capital letter; in fact, for the sake of

```
"solar_zenith_angle_in_degrees"
```

2. Make a new file called ⟨*module name*⟩`.h` in the `src/module_library` directory.[21]

```
solar_zenith_angle_in_degrees.h
```

3. Add appropriate header guards to the file.

```
#ifndef SOLAR_ZENITH_ANGLE_IN_DEGREES_H
#define SOLAR_ZENITH_ANGLE_IN_DEGREES_H
...
#endif
```

4. Add *include* directives to include the module base class declarations from `modules.h` and the type aliases from `state_map.h`.

```
#include "../modules.h"
#include "../state_map.h"
```

5. Begin a declaration of a class for the module, deriving it from SteadyModule. The identifier used for the class name should match the module name you decided upon in step 1.[22]

   Include a Doxygen-style comment (that is, starting with `/**`) describing the module and its purpose immediately above the class declaration. The first sentence should be a brief summary of the module. This is followed by additional details or a fuller description. This is an appropriate place to include citations to sources for equations used by the module in calculations, the reasoning and justification for the model, and information to help the user determine when use of the model is appropriate.[23]

   See section 2.2.3 for how to quickly generate and view the documentation Doxygen produces from the documentation markup that you write.

```
/**
 *  Given a value for the BioCro variable "cosine_zenith_angle",
 *  compute and output a value for the variable
 *  "zenith_angle_in_degrees" that corresponds to the angle having
 *  the given cosine.
 *
 *  The module allows negative cosine values, corresponding to
 *  positions of the sun below the horizon. If out-of-range values
 *  for the cosine are given, the output variable is set to NaN
 *  ("not-a-number").
 */
class solar_zenith_angle_in_degrees : public SteadyModule {
  ...
```

uniformity, it is preferable to avoid capital letters in module names altogether. Although non-latin letters are allowed, it is best to avoid them so encoding of source files does not become an issue.

[21] Here, and in what follows, we use italicized descriptors in angle brackets as placeholders for actual names or parts of names of files or variables. So, for example, if we have chosen "solar_zenith_angle_in_degrees" as the module name, then ⟨*module name*⟩`.h` designates `solar_zenith_angle_in_degrees.h`. Again, using the module name to form the file name is by convention, but a useful one if one is trying to find the code that implements a model.

[22] Here again, matching the class name to the module name is a convention. It won't break anything if you don't except perhaps the expectations of readers of your code.

[23] The function that actually *uses* the said equations will most likely be the `do_operation` function (or in some subsidiary function called by it). But `do_operation` is a private function, so it is more appropriate to put "public" information here. Documentation of `do_operation`, if any, should mostly be about programming details.

```
  ...
  ...
}
```

6. Inside the body of the class declaration, declare private data members corresponding to the input and output parameters: Each member corresponding to an input parameter should be a reference of type `const double&`.[24] By convention, the identifier used for the reference should be called ⟨*parameter name*⟩, that is, it should match the name of the corresponding parameter.[25] Members corresponding to output parameters should be of type `double*` and should have names of the form ⟨*parameter name*⟩`_op`.[26]

```
private:
   // References to input parameters:
   const double& cosine_zenith_angle;

   // Pointers to output parameters:
   double* zenith_angle_in_degrees_op;
```

7. Declare a public constructor with signature

   ⟨*class name*⟩`(const state_map* input_parameters, state_map* output_parameters)`

   The constructor needs to initialize the base class `SteadyModule` with the name for the module, and each parameter variable needs to be initialized with an initializer of the form

   ⟨*parameter name*⟩`{get_input(input_parameters, "`⟨*parameter name*⟩`")}`

   (for input parameters), or

   ⟨*parameter name*⟩`_op{get_op(output_parameters, "`⟨*parameter name*⟩`")}`

   (for output parameters).[27]

```
public:
   solar_zenith_angle_in_degrees(
       const state_map* input_parameters,
       state_map* output_parameters
   )
       : SteadyModule{"solar_zenith_angle_in_degrees"},
       cosine_zenith_angle{get_input(input_parameters, "cosine_zenith_angle")},
       zenith_angle_in_degrees_op{get_op(output_parameters, "zenith_angle_in_degrees")}
       {}
```

[24] Most existing modules use pointers rather than references for the input parameter values. But there is no reason not to use a reference, and it saves one from having to dereference the pointer when the value is used.

[25] There has been some discussion of relaxing this convention: As with module names, the (quoted) parameter name is the "public-facing" name known to users of the BioCro R functions or the BioCro C++ library. As such, it might be desirable to be able to use more natural-sounding and descriptive names (and be able to include spaces!) rather than being confined to only using names the can be used in a C++ identifier. In other words, it would be nice if these parameter names were "user-centric" instead of "programmer-centric": it is more user-centric, for example, to output simulation results with a column names like "Relative Humidity" and "Temperature" rather than "rh" and "temp". (Your users will know without thinking that you aren't talking about blood types and temporary workers!) There may, however, be other better ways of addressing this issue.

[26] Be sure to consult the master parameter list to ensure you are using the input and output parameters correctly—that you are not assigning new meanings to existing parameters, and that the units you are expecting as input and that you wish to see in output match the units designated in that list. *If you are using a parameter whose name is not in that list, you will have to add it.* [The afore-mentioned *master parameter list* is, unfortunately, as of this writing, only slightly less fictional than Borges' *Library of Babel*.]

[27] In existing modules, where pointers rather than references are used for input parameters, the initialization code will use the `get_ip` function instead of the `get_input` function: ⟨*parameter name*⟩`_ip{get_ip(input_parameters, "`⟨*parameter name*⟩`")}`

8. Add the following static function declarations to the public section of the class body.

```
static string_vector get_inputs();
static string_vector get_outputs();
```

9. Add the following declaration to the private section to override the corresponding virtual function in the base class.

```
// Implement the pure virtual function do_operation():
void do_operation() const override final;
```

10. Implement `get_inputs()`; the function should return a vector of the names of all of the input parameters. The function body can be just a return statement with the returned value written in the form of a {}-list, that is, a list of quoted strings of input parameter names, separated by commas and surrounded by curly braces.

```
string_vector solar_zenith_angle_in_degrees::get_inputs() {
    return {
        "cosine_zenith_angle"
    };
}
```

11. Implement `get_outputs()` similarly, returning a list of the names of all output parameters.

```
string_vector solar_zenith_angle_in_degrees::get_outputs() {
    return {
        "zenith_angle_in_degrees"
    };
}
```

12. Implement the `do_operation()` function. This is the heart of the module—it's where the real work gets done. Generally, the body of this function should contain a statement of the form

    `update(`⟨*output parameter*⟩`_op,`⟨*some value*⟩`);`

    for each output parameter. Here, ⟨*some value*⟩ is some numerical value calculated in the body of the function prior to the update statement, and it will generally be calculated from the values of the input parameters. Computations often may entail the use of additional include directives, for example `#include <cmath>` or `#include "../constants.h"`. In our example, the function definition (along with the required extra include directives) looks like this:

```
...
...
#include <cmath> // for acos
#include "../constants.h" // for pi
...
...
void solar_zenith_angle_in_degrees::do_operation() const {
    double zenith_angle { acos(cosine_zenith_angle)
                          * 180 / math_constants::pi };
    update(zenith_angle_in_degrees_op, zenith_angle);
}
```

A listing of the completed module code is shown in appendix B.1.

Once the new module has been written, it must be integrated into the rest of the BioCro code. This involves modifying the file `module_library/module_wrapper_factory.cpp` and requires two steps:

1. Add an include directive to include the header file for the new module.

```
#include "solar_zenith_angle_in_degrees.h"
```

2. Add an entry to `module_wrapper_factory::module_wrapper_creators`. This will be a bracketed pair of the form

   {"⟨*module name*⟩", &create_wrapper<⟨*module name*⟩>}

```
{"solar_zenith_angle_in_degrees", &create_wrapper<solar_zenith_angle_in_degrees>}
```

### 2.2.2 Example: Steps for creating a derivative module.

Suppose that we have weather data giving the ambient temperature over some period of time and we wish to write a module to help us calculate the (cumulative) number of heating degree days over that same period. This is essentially the integral of the amount the ambient temperature falls short of some "threshold" or "base" temperature, but counting negative amounts as zero. For this example, we will assume that the ambient temperature is given in a variable called "temp" and the base temperature variable is named "base_temperature". These will be varying and unvarying known variables, respectively.

The steps for writing such a module are similar to those for writing a steady-state module, but with some significant differences.

1. Decide on a name for the module.

```
"heating_degree_days"
```

2. Make a new file called ⟨*module name*⟩.h in the `src/module_library` directory.

```
heating_degree_days.h
```

3. Add appropriate header guards to the file.

```
#ifndef HEATING_DEGREE_DAYS_H
#define HEATING_DEGREE_DAYS_H
...
...
...
#endif
```

4. Add include directives to include the module base class declarations from `modules.h` and the type aliases from `state_map.h`.

```
#include "../modules.h"
#include "../state_map.h"
```

5. Begin a declaration of a class for the module, deriving it from DerivModule. The identifier used for the class name should match the module name you decided upon in step 1.

```
class heating_degree_days : public DerivModule {
  ...
  ...
  ...
}
```

6. Declare private data members corresponding to the input and output parameters. Just as with steady-state modules, input parameter references should be of type `const double&` and by convention should have identifiers matching the parameter name. Output parameter pointers should be of type `double*` and should have names of the form ⟨*parameter name*⟩`_op`.

```
private:
   // References to input parameters:
   const double& temp;
   const double& base_temperature;

   // Pointers to output parameters:
   double* heating_degree_days_op;
```

7. Declare a public constructor with signature

   ⟨*class name*⟩`(const state_map* input_parameters, state_map* output_parameters)`

   The constructor needs to initialize the base class `DerivModule` with the name for the module, and each parameter variable needs to be initialized with an initializer of the form

   ⟨*parameter name*⟩`{get_input(input_parameters, "`⟨*parameter name*⟩`")}`

   (for input parameters), or

   ⟨*parameter name*⟩`_op{get_op(output_parameters, "`⟨*parameter name*⟩`")}`

   (for output parameters).

```
public:
   heating_degree_days(
       const state_map*
           input_parameters,
       state_map*
           output_parameters
   )
   : DerivModule{"heating_degree_days"},
     temp{get_input(input_parameters, "temp")},
     base_temperature{get_input(input_parameters, "base_temperature")},
     heating_degree_days_op{get_op(output_parameters, "heating_degree_days")}
   {}
```

8. Add the following static function declarations to the public section of the class body.

```
static string_vector get_inputs();
static string_vector get_outputs();
```

9. Add the following declaration to the private section to override the corresponding virtual function in the base class.

```
// Implement the pure virtual function do_operation():
void do_operation() const override final;
```

10. Implement `get_inputs()`; the function should return a vector of the names of all of the input parameters. The function body can be just a return statement with the returned value written in the form of a {}-list, that is, a list of quoted strings of input parameter names, separated by commas and surrounded by curly braces.

```
string_vector heating_degree_days::get_inputs() {
    return {
        "temp",
        "base_temperature"
    };
}
```

11. Implement `get_outputs()` similarly, returning a list of the names of all output parameters.

```
string_vector heating_degree_days::get_outputs() {
    return {
        "heating_degree_days"
    };
}
```

12. Implement the `do_operation()` function. This is the heart of the module—it's where the real work gets done. Generally, the body of this function should contain a statement of the form

    `update(`⟨*output parameter*⟩`_op,`⟨*some value*⟩`);`

    for each output parameter. Here, ⟨*some value*⟩ is some numerical value calculated in the body of the function prior to the update statement, and it will generally be calculated from the values of the input parameters. Since this is a *derivative* module, it represents the *rate of change* ***per hour*** of the output parameter.

    Computations often may entail the use of additional include directives, for example `#include <cmath>` or `#include "../constants.h"`.

    In our example, the function definition looks like this:

```
void heating_degree_days::do_operation() const {
    double temperature_deficit { base_temperature > temp ?
                                 (base_temperature - temp) : 0};

    update(heating_degree_days_op, temperature_deficit / 24.0);[28]
}
```

A listing of the completed module code is shown in appendix B.2.

The steps for integrating the module into the rest of the BioCro code are exactly the same as for steady state modules, again only requiring a few modifications to `module_library/module_wrapper_factory.cpp`:

1. Add an include directive to include the header file for the new module.

[28] We must divide by 24 here because we want the rate of change of heating degree days *per hour*.

```
#include "heating_degree_days.h"
```

2. Add an entry to `module_wrapper_factory::module_wrapper_creators`. This will be a bracketed pair of the form

   {"⟨*module name*⟩", &create_wrapper<⟨*module name*⟩>}

```
{"heating_degree_days", &create_wrapper<heating_degree_days>}
```

### 2.2.3 Generating the Doxygen documentation

If you go to the BioCro `documentation` directory and type `make help` at the command line, you see information about multiple options for generating and displaying the Doxygen documentation in various configurations. When you are in the process of *writing* documentation, however, it is convenient to be able to quickly genererate the HTML- or PDF-style documents for just the portion of the documentation you are working on. One way to do this is as follows.

Suppose we are working on the documentation in the module file

```
src/module_library/solar_zenith_angle_in_degrees.h
```

Then we can quickly generate the HTML documentation for that file and view it by issuing the command

```
make source=../src/module_library/solar_zenith_angle_in_degrees.h \
     output_directory=minimal view
```

What we have done here is we have re-purposed the `make view` command, which normally generates *all* of the HTML documentation for the C++ source code, and overridden the default set of documented source files and the default documentation output directory so that we only spend time generating the documentation for the file of interest and so that the documentation is output to a different place (so that it won't corrupt any existing documentation). Thus, we set `source` to the relative path to the file we are working on, and we set `output` to the relative path to the directory under which we want the documentation files to appear.[29]

If we want to generate and view the PDF documentation, we could use this command instead:

```
make source=../src/module_library/solar_zenith_angle_in_degrees.h \
     output_directory=minimal view-pdf
```

## 2.3 Writing tests for new models

Once a new module has been written, a set of unit tests should be written to ensure that it works as expected. For more about unit tests in R, see https://r-pkgs.org/tests.html. For general, non-language-specific discussions of unit testing, see https://martinfowler.com/bliki/UnitTest.html or https://builtin.com/software-engineering-perspectives/what-is-unit-testing.

Here are step-by-step instructions for writing unit tests of modules, using as examples the modules we have just written.[30]

[29] The name *minimal* for the output directory is arbitrary except that it is best to choose a name that doesn't name an existing directory. Overriding *output* isn't strictly necessary, but if we don't, the output will be written to `documentation/doxygen_docs_complete`, a somewhat misleading name for the minimal documentation we are generating. Moreover, any existing documentation in that location would be corrupted by the new minimal documentation we generate.

[30] These instructions are for testing in R. It would be much more direct to use a C++ testing framework to test C++ code;

### 2.3.1 Writing unit tests for a steady-state module

In writing our tests, we will make use of the R `test_module` function. This function takes as input the name of a module (as a string), and a list specifying values for all of the input parameters of the module. It returns a list of values for the module's output parameters.

1. Create a file for the module tests in the directory `tests/testthat`. The file name should begin with "test-" and end with ".R". A good name for the file would be something of the form `test-`⟨*module name*⟩`.R`.

```
test-solar_zenith_angle_in_degrees.R
```

2. At the top of the new file, add a *context* statement providing a general description of the tests in the file. This will be a statement of the form `context("`⟨*some descriptive text*⟩`")`.

```
context('Test basic functioning of the steady-state module
    "solar_zenith_angle_in_degrees".')
```

3. Write one or more tests.

   Each test will be a statement of the form

```
test_that("⟨overall test description⟩", {
  ⟨one or more statements, including statements of the form expect_*(...)⟩
})
```

   More specifically, each test will generally set input parameter values for the module and then call `test_module`:

```
result <- test_module("⟨module name⟩", ⟨input parameter list⟩)
```

   The subsequent statements may then test various aspects of `result`.

   The heart of a `testthat` test are the *expectations*. These are assertions of some condition that should hold or some result that should be obtained from running the module code. They take the form of a function call to any one of a number of functions whose names begin with the string `expect_`. We will be primarily concerned with two such functions: `expect_equal(`⟨*actual value*⟩`,` ⟨*expected value*⟩`)`, which tests that its two arguments are equal (give or take some small amount, which can be specfied explicitely by supplying a third argument of the form `tolerance =` ⟨*some small number*⟩); and `expect_true(`⟨*some boolean condition*⟩`)`, which tests that some condition holds. More about `expect_` functions can be found in the section on expect functions in the book R Packages by Hadley Wickham and Jenny Bryan; the full complement of `expect_` functions is listed and documented at https://cran.r-project.org/web/packages/testthat/testthat.pdf.

   What should be tested? Here are some possibilities:

   (a) Sample values that produce recognizable results.

   From elementary trignometry, we know, for example, that the angle between $0^\circ$ and $180^\circ$ having a cosine of $0.5$ is $60^\circ$. We can test that this is the actual result obtained from our module with a simple test. We can write our test as follows:

```
test_that("When the cosine is 1/2, the angle is 60 degrees.", {
  input_parameters <- list(cosine_zenith_angle = 0.5)
  result = test_module("solar_zenith_angle_in_degrees", input_parameters)
```

presumably such tests could test the module functions independently of the rest of the BioCro code, with a minimum of set-up and tear-down code. But writing tests in R is simpler. Note, though, that this way of testing modules depends on the correct functioning of the R interface to the BioCro C++ library. If the R front end is buggy, then the module tests may fail for reasons not having anything to do with the underlying C++ module.

```
  expect_equal(result$zenith_angle_in_degrees, 60)
})
```

Since we will likely call `test_module` in each test, and with the same value for the module name each time, we can write a helper function to factor the repetitious code out of our tests; for example,

```
angle_from_cosine <- function(cza) {
  input_parameters <- list(cosine_zenith_angle = cza)
  result <- test_module("solar_zenith_angle_in_degrees", input_parameters)
  result$zenith_angle_in_degrees
}
```

Then our test becomes simply

```
test_that("When the cosine is 1/2, the angle is 60 degrees.", {
  angle <- angle_from_cosine(0.5)
  expect_equal(angle, 60)
})
```

(b) Edge cases.

It often makes sense to test module operation at the very limit of allowable values. For the cosine of an angle, these limits are plus and minus one.[31]

```
test_that("When the cosine is 1, the angle is 0 degrees", {
  angle <- angle_from_cosine(1)
  expect_equal(angle, 0)
})

test_that("When the cosine is -1, the angle is 180 degrees", {
  angle <- angle_from_cosine(-1)
  expect_equal(angle, 180)
})
```

(c) It's a good idea to test how the module behaves when given input that is out of bounds.

What should the module in our example do when given a value for the cosine that is greater than one or less than minus one? We'll assume it should return NaN ("not-a-number").

```
test_that("When the cosine is more than 1, the angle is not a number", {
  angle <- angle_from_cosine(1.000000001)
  expect_true(is.nan(angle))
})

test_that("When the cosine is less than -1, the angle is not a number", {
  angle <- angle_from_cosine(-1.000000001)
  expect_true(is.nan(angle))
})
```

(d) Lastly, if you discover a bug in your module, it's a good idea to write a test that demonstrates the bug—a test that will pass once the bug is fixed and ensures it stays fixed if later changes are made.

A listing of the complete test file is shown in appendix B.3.

[31] Note that zenith angles greater than $90°$ correspond to the sun being below the horizon.

### 2.3.2 Writing unit tests for a derivative module

Again, in writing our tests, we will make use of the R `test_module` function. Keep in mind that in the case of a derivative module, the output values represent the rate of change of the output parameters *per hour*. Here, we show how to test the heating degree days module.

1. Make a file for the test.

```
test-heating_degree_days.R
```

2. Add a context statement.

```
context('Test basic functioning of the derivative module "heating_degree_days".')
```

3. Write tests.

   A sample test file, with comments, for the `heating_degree_days` module is shown in appendix B.4.

### 2.3.3 Running the tests

Perhaps the easiest way to run all of the *testthat* tests is to use the *devtools* package: assuming you have an R session open in a directory inside the BioCro source tree and *devtools* has been loaded, simply do this:

```
test()
```

If you use RStudio, you needn't even type: choosing "Test Package" from the Build menu, or simply using the keyboard-shortcut "Ctrl/Cmd + Shift + T" will suffice. This method of running the tests has the added advantage that a test producing a segmentation fault won't crash the whole R session.[32]

`test()` will run all tests against the version of BioCro represented by the source code. To learn how to run the tests against an *installed* version, run the tests in only some of the files, or avoid installing *devtools*, consult the *readme* file `tests/README.md`.

[32] If you don't have the devtools package installed or haven't checked the "Use devtools package functions if available" checkbox in the build configuration pane, the RStudio "Test Package" command will still work (assuming the *testthat* package itself has been installed), but it won't do quite the same thing. Instead of simply running all of the `testthat` tests, it will run all of the `.R` files in the `tests` directory. Since one of these files, `testthat.R`, runs all of the *testthat* tests, this will essentially amount to the same thing unless there are other `.R` files in the `tests` directory.

# Appendix A Writing unit tests using Gro_solver

The section on writing unit tests was originally written in such a way that the tests called the `Gro_solver` function. This method of testing a module entails setting up a complete system and running a full simulation, albeit perhaps one with a very limited time span.

A much more convenient way to write tests is to use the `test_module` function. This is the method we demonstrate in section 2.3. By using `test_module`, we can test the basic functioning of a module while avoiding all of the many steps needed to set up a full system for running a simulation.

There may be cases, however, where we may still want the flexibility offered by being able to specify and run a full simulation. Some of the things we can do with a full-fledged system that we can't do just by calling `test_module` include

- testing how multiple modules work in conjunction with one another
- testing how the module chosen affects the evolution of a system over time
- testing how a module functions in the context of different solvers

Thus, it may be useful to present the steps required to write a test that sets up and runs a complete system; and so here, by way of some simple examples, we re-present the material from section 2.3, this time showing how to write tests using `Gro_solver` instead of `test_module`. As an added benefit, writing such tests will instruct a user in the rudiments of setting up and running a simulation in BioCro.

## A.1 Writing a unit test for a steady-state module using Gro_solver

For testing a steady-state module, it will suffice to set up a bare-bones system: the sequence of states can be a sequence of length one since we aren't interested in how the state changes over time. And the state variables need only include the input variables for the module being tested plus certain time-related variables.

1 & 2. Steps 1 and 2 are exactly as in section 2.3.1.

The third step is "new", as it involves setting up parameters for the `Gro_solver` function, something we don't need to do when using `test_module`:

3. Define parameters for the `Gro_solver` function. Assuming we call `Gro_solver` as

```
Gro_solver(initial_state, invariant_parameters, varying_parameters,
          steady_state_modules, derivative_modules, solver, verbose)
```

we have seven variables we need to define:

```
initial_state <- ...
invariant_parameters <- ...
varying_parameters <- ...
steady_state_modules <- ...
derivative_modules <- ...
solver <- ...
verbose <- ...
```

(The last two have default values of

```
list(type='Gro', output_step_size=1.0, adaptive_rel_error_tol=1e-4,
    adaptive_abs_error_tol=1e-4, adaptive_max_steps=200)
```

and FALSE, respectively. Since it doesn't matter what solver we use for testing a steady state module, and since in general tests shouldn't produce their own output, these are suitable values and may be omitted.)[33]

What values should be used to fill in the blanks? For the module lists, which are just vectors of names, the answer is clear:

```
steady_state_modules <- c("⟨name of module being tested⟩")
derivative_modules <- c()
```

These may be defined globally, outside of any test. The name of the module should exactly match the name as specified in the key of the entry for the module in `module_wrapper_creators`.

In our example, we use

```
steady_state_modules <- c("solar_zenith_angle_in_degrees")
derivative_modules <- c()
```

On the other hand, values for the module input parameters and for the initial state may potentially vary with each test. The keys, however, will in most cases be uniform across tests. For example, if the module inputs consist of invariant parameters $ip_1$, $ip_2$, ..., $ip_k$, varying parameters $vp_1$, $vp_2$, ..., $vp_m$, and initial state variables $isv_1$, $isv_2$, ..., $isv_n$, then we will be setting the corresponding function parameters to values of the following forms:

```
invariant_parameters <- list(ip_1 = ___, ip_2 = ___, ..., ip_k = ___)
varying_parameters <- list(vp_1 = ___, vp_2 = ___, ..., vp_m = ___)
initial_state <- list(isv_1 = ___, isv_2 = ___, ..., isv_n = ___)
```

As mentioned above, since we are setting up the most stripped-down of systems, these lists will be very short: we need to include a setting for each input parameter of the module being tested, and we need to include certain time-related settings. In particular, the invariant parameter list must include a setting for `timestep`, and the varying parameters must include either a setting for `time` or for both `doy` and `hour`. The values assigned to the time-related variables is immaterial for the purposes of the test except in the case that one of them is also an input to the module being tested. (The `solar_zenith_angle` module, for example, takes `time` as an input.) It is best, however, to assign values that might make sense in the simulation of a "real" system.

As for which list should include the module input variable values, this too is largely immaterial for the purposes of running the test. But again, to minimize confusion, it is best to include the setting in the list that would make sense in a real system. So, for example, if it is a parameter that normally varies over the course of time, then it doesn't make sense to assign it in the `invariant_parameters` list, and if it varies over time but not in a way that can be predicted from rates of change, then it shouldn't be assigned within the `initial_state`.

For our example, we use settings as follows.

```
invariant_parameters <- list(timestep = 1)
varying_parameters <- list(time = 1,
                           cosine_zenith_angle = ⟨some value that will vary from test to test⟩)
initial_state <- list()
```

4. Write one or more tests.

   As in section 2.3.1, each test will be a statement of the form

   ```
   test_that("⟨overall test description⟩", {
   ```

[33] There may be cases where we may want to temporarily set `verbose` to `TRUE` in order to debug a test we are writing. In this case, if we want to default the value of `solver`, we must use a named argument to set the `verbose` argument and write '`verbose=TRUE`' instead of just '`TRUE`' so that `TRUE` is not taken to be the value for `solver`.

```
  ⟨one or more statements, including statements of the form expect_*(...)⟩
})
```

However, instead of getting results to test by calling `test_module`, we call `Gro_solver` instead:

```
result = Gro_solver(initial_state, invariant_parameters, varying_parameters,
    steady_state_modules, derivative_modules)
```

As before, the various `expect_` functions are used to test `result`. In fact, once we make some global parameter settings and re-define our helper function `angle_from_cosine` to utilize `Gro_solver` instead of `test_module`, the tests themselves can remain exactly as in section 2.3.1.

Here are the precise settings we will use for the modules, invariant parameters, and initial state:

```
steady_state_modules <- c("solar_zenith_angle_in_degrees")
derivative_modules <- c()

initial_state <- list()
invariant_parameters <- list(timestep = 1)
```

The helper function `angle_from_cosine` becomes:[34]

```
angle_from_cosine <- function(cza) {
  varying_parameters <- list(time = 1, cosine_zenith_angle = cza)
  result = Gro_solver(initial_state, invariant_parameters, varying_parameters,
      steady_state_modules, derivative_modules)
  result$zenith_angle_in_degrees
}
```

A listing of the complete test file is shown in appendix B.5.

## A.2 Writing unit tests for a derivative module using Gro_solver

Again, it will suffice to set up a bare-bones system. But this time, we will set up a time sequence of length two or more since we want to see how the derivative affects how the state changes over time. For the most basic tests of module functionality, we need only use a sequence of two states. Again, the state variables need only include the input variables for the module being tested and the time-related variables.[35]

Here, we show how to test the heating degree days module using `Gro_solver`.

1 & 2. Steps 1 and 2 are exactly the same as in section 2.3.2.

Again, we need an extra step for setting up some `Gro_solver` parameters:

3. Define parameters for the `Gro_solver` function. Again, we will be assuming we call `Gro_solver` with

```
Gro_solver(initial_state, invariant_parameters, varying_parameters,
          steady_state_modules, derivative_modules, solver, verbose)
```

[34] We make no attempt here to coordinate the input values for `cosine_zenith_angle` and for `time`. While in general we prefer to use input values that might be associated with a realistic system, for the purposes of writing tests of a module, it may sometimes be more convenient not to do so.

[35] In the case of a derivative module where some of the inputs correspond to outputs of steady-state modules, we have two choices with regard to testing: (1) include, as part of the test, steady-state modules that produce these required inputs; (2) artificially include values for these inputs as part of the varying parameters. The latter option is the simplest and has the advantage of narrowing the scope of the test to the module we are most interested in testing. But there may be cases where we want to test how two or more modules work in conjunction with one another.

The assignments for the two module vectors are more or less the reverse of what they were for the steady-state module test:

```
steady_state_modules <- c()
derivative_modules <- c("⟨name of module being tested⟩")
```

For our example, this becomes

```
steady_state_modules <- c()
derivative_modules <- c("heating_degree_days")
```

As for `initial_state`, `invariant_parameters`, and `varying_parameters`, we may want to vary each of these from test to test. We may also want to run tests using various solvers and vary the timestep.

For our example, we'll write a global helper function that makes this easy.

```
calculate_heating_degree_days <- function(initial_value, time_vector,
    temperature_vector, threshold_temperature, timestep, solver) {
  initial_state <- list(heating_degree_days = initial_value)
  invariant_parameters <- list(base_temperature = threshold_temperature, timestep =
      timestep)
  varying_parameters <- list(time = time_vector, temp = temperature_vector)

  result <- Gro_solver(initial_state, invariant_parameters, varying_parameters,
                       steady_state_modules, derivative_modules, solver)

  result$heating_degree_days[length(time_vector)]
}
```

4. Write tests.

   A sample test file, with comments, for the `heating_degree_days` module is shown in appendix B.6.

# Appendix B Code listings

## B.1 Full listing of solar-zenith-angle-in-degrees module

(In this listing, we have expanded the class's Doxygen comment to demonstrate how LaTeX-style equations could be included, if desired. See section 2.2.3 for instructions on how to quickly and easily generate and display the resulting HTML and PDF forms of the Doxygen documentation.)

```
#ifndef SOLAR_ZENITH_ANGLE_IN_DEGREES_H
#define SOLAR_ZENITH_ANGLE_IN_DEGREES_H

#include <cmath> // for acos
#include "../constants.h" // for pi
#include "../modules.h"
#include "../state_map.h"

/**
 *  Given a value for the BioCro variable "cosine_zenith_angle",
 *  compute and output a value for the variable
 *  "zenith_angle_in_degrees" that corresponds to the angle having the
 *  given cosine.
 *
 *  The module allows negative cosine values, corresponding to
```

```
 *  positions of the sun below the horizon. If out-of-range values
 *  for the cosine are given, the output variable is set to NaN
 *  ("not-a-number").
 *
 *  Denoting the zenith angle by \f$\theta_s\f$ and its cosine by
 *  \f$x\f$, the formula used to compute \f$\theta_s\f$ (in degrees)
 *  from \f$x\f$ is
 *  \f[
 *      \theta_s =
       \begin{cases} \arccos(x) \cdot 180/\pi, & \text{if }-1 \le x \le 1; \\
                   \text{undefined,}       & \text{otherwise.}
       \end{cases}
 *  \f]
 *  where the usual range \f$ 0 \le x \le \pi \f$ for the \f$\arccos\f$
 *  function is used and undefined values are represented by NaN.
 *
 */
class solar_zenith_angle_in_degrees : public SteadyModule {

 public:
    solar_zenith_angle_in_degrees(const state_map* input_parameters, state_map*
        output_parameters)
       : SteadyModule{"solar_zenith_angle_in_degrees"},
         cosine_zenith_angle{get_input(input_parameters, "cosine_zenith_angle")},
         zenith_angle_in_degrees_op{get_op(output_parameters, "zenith_angle_in_degrees")}
       {}

    static string_vector get_inputs();
    static string_vector get_outputs();

 private:
    // References to input parameters:
    const double& cosine_zenith_angle;

    // Pointers to output parameters:
    double* zenith_angle_in_degrees_op;

    // Implement the pure virtual function do_operation():
    void do_operation() const override final;
};


string_vector solar_zenith_angle_in_degrees::get_inputs() {
    return {
        "cosine_zenith_angle"
    };
}


string_vector solar_zenith_angle_in_degrees::get_outputs() {
    return {
        "zenith_angle_in_degrees"
    };
}

void solar_zenith_angle_in_degrees::do_operation() const {
    double zenith_angle { acos(cosine_zenith_angle)
                          * 180 / math_constants::pi };
    update(zenith_angle_in_degrees_op, zenith_angle);
```

```
}

#endif
```

## B.2 Full listing of heating-degree-days module

```
#ifndef HEATING_DEGREE_DAYS_H
#define HEATING_DEGREE_DAYS_H

#include "../modules.h"
#include "../state_map.h"

class heating_degree_days : public DerivModule {


   public:
    heating_degree_days(
        const state_map*
            input_parameters,
        state_map*
            output_parameters
    )
    : DerivModule{"heating_degree_days"},
      temp{get_input(input_parameters, "temp")},
      base_temperature{get_input(input_parameters, "base_temperature")},
      heating_degree_days_op{get_op(output_parameters, "heating_degree_days")}
    {}

    static string_vector get_inputs();
    static string_vector get_outputs();

  private:
    // References to input parameters:
    const double& temp;
    const double& base_temperature;

    // Pointers to output parameters:
    double* heating_degree_days_op;

    // Implement the pure virtual function do_operation():
    void do_operation() const override final;
};

string_vector heating_degree_days::get_inputs() {
    return {
        "temp",
        "base_temperature"
      };
}

string_vector heating_degree_days::get_outputs() {
    return {
        "heating_degree_days"
    };
}

void heating_degree_days::do_operation() const {
```

```
    double temperature_deficit { base_temperature > temp ?
                                 (base_temperature - temp) : 0 };
    update(heating_degree_days_op, temperature_deficit / 24.0);
}


#endif
```

## B.3 Full listing of the unit-test file for solar-zenith-angle-in-degrees

```
context('Test basic functioning of the steady-state module "solar_zenith_angle_in_degrees".')

# Run the module using a cza as the the value of the cosine of the
# zenith angle and return the value of zenith_angle_in_degrees from
# the result.
angle_from_cosine <- function(cza) {
    input_parameters <- list(cosine_zenith_angle = cza)

    result <- test_module("solar_zenith_angle_in_degrees", input_parameters)

    result$zenith_angle_in_degrees
}


test_that("When the cosine is 1/2, the angle is 60 degrees.", {
    angle <- angle_from_cosine(0.5)
    expect_equal(angle, 60)
})

test_that("When the cosine is 1, the angle is 0 degrees", {
    angle <- angle_from_cosine(1)
    expect_equal(angle, 0)
})

test_that("When the cosine is -1, the angle is 180 degrees", {
    angle <- angle_from_cosine(-1)
    expect_equal(angle, 180)
})

test_that("When the cosine is more than 1, the angle is not a number", {
    angle <- angle_from_cosine(1.000000001)
    expect_true(is.nan(angle))
})

test_that("When the cosine is less than -1, the angle is not a number", {
    angle <- angle_from_cosine(-1.000000001)
    expect_true(is.nan(angle))
})
```

## B.4 Full listing of the unit-test file for heating-degree-days

```
context('Test basic functioning of the derivative module "heating_degree_days".')

## Given the (average) ambient temperature over an hour-long interval
## and given a threshold temperature, return the number of heating
## degree days that accumulate over the interval.
calculate_heating_degree_days <- function(ambient_temperature, threshold_temperature) {
```

```
    input_parameters <- list(temp = ambient_temperature, base_temperature =
        threshold_temperature)
    result <- test_module("heating_degree_days", input_parameters)
    result$heating_degree_days
}

threshold_temperature = 18.3 # ~65 deg F

test_that("If the ambient temperature is one degree below the threshold temperature, 1/24 of
    a heating degree day accumulates in each hour interval", {
    ambient_temperature = threshold_temperature - 1
    hdd <- calculate_heating_degree_days(ambient_temperature, threshold_temperature)
    expect_equal(hdd, 1/24)
})

test_that("When the ambient temperature is equal to the threshold temperature, no heating
    degree days accumulate", {
    ambient_temperature = threshold_temperature
    hdd <- calculate_heating_degree_days(ambient_temperature, threshold_temperature)
    expect_equal(hdd, 0)
})


test_that("When the ambient temperature is greater than the threshold temperature, no
    heating degree days accumulate", {
    number_of_samples <- 10
    temperature_surpluses <- sample(1:200, number_of_samples)/10 # test surpluses in the
        range of 0.1 to 20.0
    for (surplus in temperature_surpluses) {
        ambient_temperature = threshold_temperature + surplus
        hdd <- calculate_heating_degree_days(ambient_temperature, threshold_temperature)
        expect_equal(hdd, 0)
    }
})

test_that("When the ambient temperature is less than the threshold temperature, the heating
    degree day accumulation for the hour interval is 1/24 the difference", {
    number_of_samples <- 10
    temperature_deficits <- sample(1:500, number_of_samples)/10 # test deficits in the range
        of 0.1 to 50.0
    for (deficit in temperature_deficits) {
        ambient_temperature = threshold_temperature - deficit
        hdd <- calculate_heating_degree_days(ambient_temperature, threshold_temperature)
        expect_equal(hdd, deficit/24)
    }
})
```

## B.5 Full listing of the unit-test file for solar-zenith-angle-in-degrees—Gro_solver version

```
context('Test basic functioning of the steady-state module "solar_zenith_angle_in_degrees".')

steady_state_modules <- c("solar_zenith_angle_in_degrees")
derivative_modules <- c()

initial_state <- list()
invariant_parameters <- list(timestep = 1)
```

```
# Run the system using a cza as the the value of the cosine of the
# zenith angle and return the value of zenith_angle_in_degrees from
# the result.
angle_from_cosine <- function(cza) {
    # For these tests, the value of time is immaterial, but either
    # time or both doy and hour must be in the list of keys for
    # varying parameters.
    varying_parameters <- list(time = 1, cosine_zenith_angle = cza)

    result = Gro_solver(initial_state, invariant_parameters, varying_parameters,
        steady_state_modules, derivative_modules)

    result$zenith_angle_in_degrees
}


test_that("When the cosine is 1/2, the angle is 60 degrees.", {
    angle <- angle_from_cosine(0.5)
    expect_equal(angle, 60)
})

test_that("When the cosine is 1, the angle is 0 degrees", {
    angle <- angle_from_cosine(1)
    expect_equal(angle, 0)
})

test_that("When the cosine is -1, the angle is 180 degrees", {
    angle <- angle_from_cosine(-1)
    expect_equal(angle, 180)
})

test_that("When the cosine is more than 1, the angle is not a number", {
    angle <- angle_from_cosine(1.000000001)
    expect_true(is.nan(angle))
})

test_that("When the cosine is less than -1, the angle is not a number", {
    angle <- angle_from_cosine(-1.000000001)
    expect_true(is.nan(angle))
})
```

## B.6 Full listing of the unit-test file for heating-degree-days—Gro_solver version

```
context('Test basic functioning of the derivative module "heating_degree_days".')

steady_state_modules <- c()
derivative_modules <- c("heating_degree_days")

## Given an initial value for the number of heating degree days, a
## vector of equally-spaced time values (as fractional day-of-year
## values), a corresponding vector of temperatures, a threshold
## temperature value, a timestep value, and the specification of a
## solver, run the system and return the final heating-degree-days
## value.
calculate_heating_degree_days <- function(initial_value, time_vector, temperature_vector,
    threshold_temperature, timestep, solver) {
    initial_state <- list(heating_degree_days = initial_value)
```

```
    invariant_parameters <- list(base_temperature = threshold_temperature, timestep =
        timestep)
    varying_parameters <- list(time = time_vector, temp = temperature_vector)

    result <- Gro_solver(initial_state, invariant_parameters, varying_parameters,
                      steady_state_modules, derivative_modules, solver)

    result$heating_degree_days[length(time_vector)]
}

euler_solvers <- list(
    list(type='Gro_euler', output_step_size=1, adaptive_rel_error_tol=1e-4,
        adaptive_abs_error_tol=1e-4, adaptive_max_steps=200),
    list(type='Gro_euler_odeint', output_step_size=1, adaptive_rel_error_tol=1e-4,
        adaptive_abs_error_tol=1e-4, adaptive_max_steps=200)
)

other_solvers <- list(
    list(type='Gro_rk4', output_step_size=1, adaptive_rel_error_tol=1e-4,
        adaptive_abs_error_tol=1e-4, adaptive_max_steps=200),
    list(type='Gro_rkck54', output_step_size=1, adaptive_rel_error_tol=1e-4,
        adaptive_abs_error_tol=1e-4, adaptive_max_steps=200)
)

non_euler_solvers <- c(other_solvers, list(list(type='Gro_rsnbrk', output_step_size=1,
    adaptive_rel_error_tol=1e-4, adaptive_abs_error_tol=1e-4, adaptive_max_steps=200)))

well_behaved_solvers <- c(euler_solvers, other_solvers) # Rosenbrook solver doesn't work
    with time sequences of length less than 3.

all_solvers <- c(euler_solvers, non_euler_solvers)

test_that("A day at one degree below the threshold temperature adds 1 heating degree day", {
    ## test using 24-hour timesteps:
    for (i in 1:length(well_behaved_solvers)) {
        hdd <- calculate_heating_degree_days(
            0,              # start at zero heating-degree days
            c(1, 2),        # days-of-year 1 and 2
            c(19, 19),      # assume a constant ambient temperature of 19 degrees ...
            20,             # ... which is one degree below the threshold temperature
            24,             # the time interval is 24 hours--the time between successive days
            well_behaved_solvers[[i]]
        )
        expect_equal(hdd, 1.0)
    }
    ## test using 1-hour timesteps:
    for (i in 1:length(all_solvers)) {
        hdd <- calculate_heating_degree_days(
            0,                  # start at zero heating-degree days
            seq(1, 2, 1/24.0), # a sequence of times one-hour apart covering one day
            rep(19, 25),        # assume a constant ambient temperature of 19 degrees ...
            20,                 # ... which is one degree below the threshold temperature
            1,                  # the time interval is 1 hour
            all_solvers[[i]])
        expect_equal(hdd, 1.0)
    }
})

test_that("A random assortment of temperatures below the threshold
```

```
          temperature yields a number of heating degree days equal to the
          cumulative amount by which the temperatures are under the
          threshold temperature.", {
             days <- 30
             temperature_deficits <- sample(0:50, days)
             cumulative_temperature_deficit <- sum(temperature_deficits[1:(days - 1)]) #
                 Don't count the last day's temperature.

             temperatures <- 20 - temperature_deficits

             for (i in 1:length(euler_solvers)) {
                 hdd <- calculate_heating_degree_days(
                     0,                # start at zero heating-degree days
                     1:days,
                     temperatures,     # a random sequence of temperatures below the
                         threshold temperature
                     20,               # the threshold temperature
                     24,               # the time interval is 24 hours--the time between
                         successive days
                     euler_solvers[[i]])
                 expect_equal(hdd, cumulative_temperature_deficit)
             }

             ## For the non-Euler solvers, we have set a greater tolerance value than the
                 default:

             for (i in 1:length(non_euler_solvers)) {
                 hdd <- calculate_heating_degree_days(
                     0,                    # start at zero heating-degree days
                     1:days,
                     temperatures,         # a random sequence of temperatures below the
                         threshold temperature
                     20,                   # the threshold temperature
                     24,                   # the time interval is 24 hours--the time between
                         successive days
                     non_euler_solvers[[i]])
                 expect_equal(hdd, cumulative_temperature_deficit, tolerance = 0.03)
             }

         }
)

test_that("A random assortment of temperatures above the threshold
          temperature results in zero heating degree days.", {
             days <- 30
             temperature_surpluses <- sample(0:50, days)

             temperatures <- 20 + temperature_surpluses

             for (i in 1:length(all_solvers)) {
                 hdd <- calculate_heating_degree_days(
                     0,               # start at zero heating-degree days
                     1:days,
                     temperatures,    # a random sequence of temperatures above the threshold
                         temperature
                     20,              # the threshold temperature
                     24,              # the time interval is 24 hours--the time between
                         successive days
                     all_solvers[[i]])
```

```
                expect_equal(hdd, 0)
            }
        }
)
```