A Practical Guide to BioCro

BioCro Development Team

December 15, 2025

```
This document was generated from the version of BioCro specified as follows:

Commit Hash: 0848588
Date: Mon, 15 Dec 2025 12:33:00 -0600
Branch:
```

Contents

1 Getting Started

This vignette provides a demonstration of a few important BioCro functions and some R basics. We will be using functions from the `BioCro` and `lattice` packages, so we must make sure they are both loaded:

```
library(BioCro)
library(lattice)
```

If these packages are not installed on your machine, installation instructions for BioCro are available at its GitHub repository web page, while the `lattice` package can be installed by typing the following in R:

```
install.packages('lattice')
```

While reading this guide, please keep in mind that R has a built-in help system that can provide documentation for any command, including those in the BioCro library. This help system can be accessed using the `?` command. For example:

```
# Access documentation for a BioCro function when the package is loaded
?run_biocro

# Access documentation for a BioCro data set, even if the package is not loaded
?BioCro::soybean

# Access documentation for a base R function
?list

# Access documentation for an R operator, which must be quoted using ', `, or "
?`<-`
```

New R users may also find it useful to consult the resources available in Section 7.2.

2 Running a BioCro Simulation

BioCro's main purpose is to allow a user to easily simulate the growth of a crop throughout a growing season. To demonstrate this ability, we will run BioCro's soybean growth model using weather data corresponding to the 2002 growing season in Champaign, Illinois. In this section, we discuss the R functions and input arguments required to accomplish this goal. Along the way, we will introduce the most important R data types for BioCro analysis: lists, data frames, and vectors. The discussion of these objects presented in this vignette is intentionally basic and rudimentary; many guides are available online that discuss them in more detail (Section 7.2).

2.1 Calling `run_biocro`

To run a BioCro simulation, we use the `run_biocro` function, which builds a set of model equations from its input arguments and solves the model over the specified time period. The BioCro package includes pre-set collections of values that can be passed as input arguments to `run_biocro` when simulating soybean growth:

```
soybean_result <- run_biocro(
  soybean$initial_values,
  soybean$parameters,
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules,
  soybean$ode_solver
)
```

(See Section 7.1.1 for an alternate way to write this command.)

This command begins to illustrate the power of BioCro: with one function call, we are able to run a complex crop growth simulation including a mechanistic model for C_3 photosynthesis coupled to equations for leaf energy balance and stomatal opening, photothermal soybean development, soil water dynamics, and many other important processes. All of this was accomplished using the standard BioCro module library, and we didn't need to write a single equation.

The following subsections discuss the input arguments to `run_biocro`, explaining how they work together to define and run a simulation, how to specify them in R, and how to use pre-set options available in the BioCro package. In each case, we will also show how they relate to an important element of the BioCro philosophy: that *models are sets of equations*. Specifically, we will emphasize that each input argument to `run_biocro` specifies equations that help to determine the time evolution (or lack thereof) for one or more members x_i of the model's state $\mathbf{x}$, which is taken to comprise all quantities involved in the model (Equations 3, 4, 5, 6, and 7).

Although this might seem like a merely philosophical point, it has considerable practical implications regarding the design and use of the BioCro framework. In particular, it enables full modularity—which can be viewed as simply swapping one equation for another—allowing for great flexibility in model design and analysis.

2.2 Choosing the modules

Background

To run a BioCro simulation, the user must specify collections of *modules* to use. A module represents one or more related equations that model some aspect of plant biology. Each module M has *input quantities* $\mathbf{x}_{M_{\text{in}}}$ and *output quantities* $\mathbf{x}_{M_{\text{out}}}$; when a module is run, its outputs are determined from its inputs according to its equations. Modules come in two types: *differential modules* calculate terms of the time derivatives of their output quantities, while *direct modules* directly calculate the values of their output quantities. In other words, a direct module defines a function $\mathbf{f}_M$ that calculates its outputs from its inputs according to the following equation:

$$\mathbf{x}_{M_{\text{out}}} = \mathbf{f}_M\left(\mathbf{x}_{M_{\text{in}}}, t\right). \tag{1}$$

Likewise, a differential module defines a function $\mathbf{g}_M$ that calculates derivatives of its outputs from its inputs according to the following equation:

$$\frac{d\mathbf{x}_{M_{\text{out}}}}{dt} = \mathbf{g}_M\left(\mathbf{x}_{M_{\text{in}}}, t\right). \tag{2}$$

During a simulation, modules are able to form chains where the output of one module is used as the input to another. For example, a direct module could calculate partitioning coefficients based on a plant's physiological age and a differential module could subseqently use those parititioning coefficients to distribute assimilated carbon to different organs. In order for these module chains to behave as expected, the collection of modules must be run in a particular order to ensure that the values of each module's inputs are known before it attempts to calculate its outputs. When a simulation is run, BioCro automatically determines a suitable ordering for the modules (Section 2.6). Any module inputs that are not calculated by other modules must be provided as parameters (Section 2.3) or drivers (Section 2.4).

Since direct modules calculate instantaneous values of quantities, at most one direct module can have a particular quantity as an output. For example, multiple direct modules are able to calculate the canopy assimilation rate, but only one such module can be used in a given simulation. Before running a simulation, BioCro checks for any overlap between the outputs of the direct modules (Section 2.6). On the other hand, since differential modules calculate terms of a derivative, multiple differential modules can have a particular quantity as an output; in this case, the module outputs will be added together to determine the overall derivative. For example, one differential module may calculate a positive rate of leaf carbon gain due to assimilation while another calculates a negative rate of leaf carbon loss due to senescence; both may be used in a given simulation.

Within the context of a simulation, the inputs and outputs of all modules are part of the state $\mathbf{x}$. In this situation, we can say that the collection of direct modules defines a collection of functions where each function f_i calculates the value of an individual state element x_i from a subset of the state $\mathbf{x}^i$ according to the following equation:

$$x_i = f_i\left(\mathbf{x}^i, t\right). \tag{3}$$

Each f_i is defined by one module, and one module may define multiple f_i. Likewise, we can also say that the collection of differential modules defines a collection of functions where each function g_i calculates the derivative of an individual state element x_i from a subset of the state $\mathbf{x}^i$ according to the following equation:

$$\frac{dx_i}{dt} = g_i\left(\mathbf{x}^i, t\right). \tag{4}$$

Each g_i can be defined by more than one module, and one module may define terms of multiple g_i.

R implemetation

BioCro modules are organized into groups called *module libraries* (Section 5); each library has a name, and each module within it has a local name, allowing individual modules to be identified by specially-formatted strings of the type `library_name:local_module_name`, where the library name (`library_name`) and local module name (`local_module_name`) are separated by a colon (`:`); we refer to these strings as *fully-qualified module names*. Collections of modules (which may come from multiple libraries) can be specified in R using the `c` command, which creates an R vector from its input arguments. For example, if we wish to use a module called `Module_1` from a library called `libA` and a module called `Module_2` from a library called `libB`, we could write the following:

```
modules <- c(
    'libA:Module_1',
    'libB:Module_2'
)
```

Elements of a vector can be accessed using an integer index; for example, we can retrieve or modify the first element of `modules` as in the following example, where we replace `Module_1` with `Module_3` from the same library:

```
print(modules[1])

## [1] "libA:Module_1"

modules[1] <- 'libA:Module_3'
print(modules)

## [1] "libA:Module_3" "libB:Module_2"
```

Sometimes it may be convenient to specify a module's role within the collection. In this case, it is helpful to provide names for one or more of the modules in a collection. Here we demonstrate how to accomplish this with an R *list*:

```
differential_modules <- list(
   'BioCro:partitioning_growth',
   thermal_time_module = 'BioCro:thermal_time_linear'
)
```

Here, `thermal_time_module` is the name of an element of the list `differential_modules`; it is *not* the name of a module. Its purpose is to allow us to access that list element by a meaningful identifier (`thermal_time_module`) rather than its numbered position within the list. Now, if we want to use a different equation for calculating thermal time, we can easily switch it via a command like:

```
differential_modules$thermal_time_module <- 'BioCro:thermal_time_trilinear'
```

Swapping one module for an alternate version can be very powerful when evaluating the differences between multiple modeling strategies, and accomplishing this "plug-and-play modularity" with ease is one of BioCro's most important abilities. For an example of how the performance of two modules can be compared in the context of a soybean simulation, see the *Quantitative Comparison Between Two Photosynthesis Models* vignette.

It may be tedious to repeatedly specify the same library name multiple times in a set of fully-qualified module names. Doing so can be avoided by using the `module_paste` function, which automatically prepends a library name to a vector or list of local module names, forming a set of fully-qualified module names. For example, we can create the same list of differential module names using `module_paste` as follows:

```
differential_modules <- module_paste('BioCro', list(
  'partitioning_growth',
  thermal_time_module = 'thermal_time_linear'
))
```

Pre-set options available in the BioCro package

In Section 2.1, we supplied pre-defined lists of modules when calling the `run_biocro` function: `soybean$direct_modules` and `soybean$differential_modules`. As discussed in that section, these modules define a large set of equations that represent many important processes involved in the growth of a crop and its interactions with its environment. The contents of these lists can be printed to the R console using the `str` command, which nicely shows the *str*ucture of the object being printed:

```
str(soybean$differential_modules)

## List of 6
##  $ senescence  : chr "BioCro:senescence_logistic"
##  $             : chr "BioCro:maintenance_respiration"
##  $             : chr "BioCro:partitioning_growth"
##  $ soil_profile: chr "BioCro:two_layer_soil_profile"
##  $             : chr "BioCro:development_index"
##  $ thermal_time: chr "BioCro:thermal_time_linear"
```

To learn about specific modules, see Section 6.2.

2.3 Choosing the parameters and initial values

Background

To run a BioCro simulation, a user must supply the values of *parameters* (quantities whose values are taken to be constant during the course of a simulation) and the *initial values* of the *differential quantities* (quantities whose evolution with time is determined by differential equations). In general, the sets of required parameters and initial values will be determined by the modules and drivers that a user has chosen to use (Sections 2.2 and 2.4, respectively).

Specifying the parameters and initial values can be thought of as defining simple equations of the form

$$x_i = p_i \tag{5}$$

and

$$x_i(t_0) = x_{i0}, \tag{6}$$

where x_i is an individual state element, p_i is a constant parameter value and x_{i0} is the value of a differential quantity at the start of the simulation (where $t = t_0$).

R implementation

Both the parameters and initial values must be specified using lists of named elements. When specifying collections of modules (Section 2.2), names are optional; for parameters and initial values, names are required since they are used to uniquely identify each quantity. For example, if we have two parameters called `parameter_1` and `parameter_2` whose values are 2.3 and 8.9, respectively, we can specify them using a list as follows:

```
parameters <- list(
    parameter_1 = 2.3,
    parameter_2 = 8.9
)
```

Here the syntax of defining a list—where a value is assigned to a name using an equals sign—is a reminder that parameter specifications can be thought of as representing simple individual equations like Equations 5 and 6.

Pre-set options available in the BioCro package

In Section 2.1, we supplied pre-defined lists of parameters and initial values when calling the `run_biocro` function: `soybean$parameters` and `soybean$initial_values`. These lists include quantities that characterize soybean physiological processes (such as $v_{\mathrm{c,max}}$), specifics related to field conditions (such as the initial seed mass per unit ground area), and other important categories of quantities. To learn about specific quantities, see Section 6.1.

2.4 Choosing the drivers

Background

To run a BioCro simulation, a user must supply the *drivers*, which are quantities whose values are taken to be known beforehand at a set of discrete time points. In most BioCro simulations, the drivers represent weather data obtained from sensors at a particular location. Within the BioCro framework, interpolation is used to create a continuous function of time for each quantity defined in the drivers. In this sense, each driver can be viewed as following an equation of the form

$$x_i = d_i(t), \tag{7}$$

where where x_i is an individual state element and $d_i(t)$ is a function determined by interpolation from the driver's discrete values.

R implementation

The drivers must be specified as a *data frame*, an R data structure that conceptually resembles a table. In the case of the drivers, each column in the data frame represents one quantity, and each row represents one time point. For example, we could define a simple set of drivers that specifies the air temperature (`temp`) throughout one day at 3-hour intervals as follows:

```
hour <- seq(0, 23, 3)
temp <- 20 + 8 * sin((hour / 24) * pi)^2 # we use "vector arithmetic" to form ‘temp‘
drivers <- data.frame(
    time = hour, # BioCro requires the drivers to have ‘time‘ variable.
    temp = temp
)
```

(Typically, BioCro drivers would not be determined from an analytic equation; here we do this for convenience when providing an example of constructing a simple data frame.)

The contents of a data frame can be viewed in several ways (Section 3); here we simply print to the R terminal using the `print` command:

```
print(drivers)

##   time     temp
## 1    0 20.00000
## 2    3 21.17157
## 3    6 24.00000
## 4    9 26.82843
## 5   12 28.00000
## 6   15 26.82843
## 7   18 24.00000
## 8   21 21.17157
```

Pre-set options available in the BioCro package

In Section 2.1, we supplied a pre-defined data frame of drivers when calling the `run_biocro` function: `soybean_weather$'2002'`. Assembling drivers that represent weather data in a particular location can be a complicated process because different weather data sets or weather sensors produce different outputs and use different formats, precluding a general approach to obtaining and processing weather data. For convenience, the BioCro package includes some pre-processed weather data from Champaign, Illinois which can be used to replicate the analysis of a few published papers and serve illustrative purposes.

2.5 Choosing the differential equation solver

Background

To run a BioCro simulation, a user must specify an algorithm to use for solving the set of coupled ordinary differential equations (ODEs) defined by the other input arguments to `run_biocro`; in BioCro, we refer to such an algorithm as an *ode_solver*. In general, numerical ODE integration methods require a function $\mathbf{G}(\mathbf{x}_{\text{differential}}, t)$ that calculates derivatives $d\mathbf{x}_{\text{differential}}/dt$ from values of the time and a set of quantities $\mathbf{x}_{\text{differential}}$. Along with a set of initial values and a time domain, the algorithm determines the time evolution of $\mathbf{x}_{\text{differential}}$. In BioCro, such a function $\mathbf{G}$ can be defined by taking the following steps:

1. Determine the values of the parameters using Equation 5.
2. Determine the values of the drivers using Equation 7.
3. Determine the values of the direct module outputs using Equation 3.
4. Determine the derivatives of the differential quantities using Equation 4.

Once the time evolution of the differential quantities has been determined by the ODE solver, the time evolution of the state as a whole can be determined using steps 1-3 above. This process of defining $\mathbf{G}$, passing it to a solver, and then determining the entire state's time evolution is automatically handled by the BioCro framework.

R implementation

An ODE solver is specified using a list with a particular format, where the list must contain the following named elements:

- `type`: The name of the algorithm.
- `output_step_size`: The time interval to be used in the output of `run_biocro`, specified as a multiplicative factor relative to the time step used in the drivers; for example, if the drivers have an hourly time step and the `output_step_size` is 0.5, the simulation result will have a half-hourly interval.

- `adaptive_rel_error_tol`: A relative error tolerance to be used with adaptive step-size ODE solvers.
- `adaptive_abs_error_tol`: An absolute error tolerance to be used with adaptive step-size ODE solvers.
- `adaptive_max_steps`: The maximum number of attempts allowed when an adaptive step-size ODE solver tries to find a new step size.

Pre-set options available in the BioCro package

In Section 2.1, we supplied a pre-defined ODE solver when calling the `run_biocro` function: `soybean$ode_solver`. Its contents can be viewed as with any other list:

```
str(soybean$ode_solver)

## List of 5
##  $ type                  : chr "boost_rkck54"
##  $ output_step_size      : num 1
##  $ adaptive_rel_error_tol: num 1e-04
##  $ adaptive_abs_error_tol: num 1e-04
##  $ adaptive_max_steps    : num 200
```

Here the `boost_rkck54` ODE solver is used. This is the `Boost` library's implementation of an adaptive step-size 5^{th} order Runge-Kutta solver with 4^{th} order Cash-Karp error estimation. This is an excellent solver for any situation where an Euler solver is not required and when the system of equations is not stiff. For stiff systems, the `boost_rosenbrock` solver, which is the `Boost` library's implementation of an adaptive step-size 4^{th} order Rosenbrock solver, may be more appropriate. A full list of the available types can be obtained using the `get_all_ode_solvers()` function.

2.6 Validating `run_biocro` input arguments

Background

Before running a simulation, it is essential to check whether the supplied modules, parameters, initial values, and drivers are able to specify a well-defined system that can in principle be solved. Four conditions are sufficient to ensure the validity of a model in this sense:

1. The input quantities of each module must be *defined*; here, a quantity is said to be defined if it is a parameter (Section 2.3), a driver (Section 2.4), or if it is the output of a module (Section 2.2).
2. No quantity can be defined more than once. For example, a quantity cannot be defined as a parameter and as the output of a direct module; similarly, a quantity cannot be defined by multiple direct modules.
3. Each differential module output must have a corresponding initial value (Section 2.3).
4. There must exist an ordering of the modules such that a module requiring a particular direct module output occurs later than the module providing that output; in other words, there must be no circular dependencies among the direct modules.

R implementation

The `run_biocro` function checks the supplied arguments to ensure that these four conditions are satisfied and provides helpful feedback to the user if a problem is detected. For example, if one or more of the differential quantities is missing an initial value, or if a parameter is missing, an error will occur:

```
soybean_result <- run_biocro(
  within(soybean$initial_values, rm(Leaf)),          # remove the initial 'Leaf' value
  within(soybean$parameters, rm(leaf_reflectance)), # remove 'leaf_reflectance'
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules,
  soybean$ode_solver
)

## Warning in rm(leaf_reflectance):  object 'leaf_reflectance' not found
## Error in as.data.frame(.Call(R_run_biocro, initial_values, parameters, :  Caught exception
in R_run_biocro:  Thrown by dynamical_system::dynamical_system:  the supplied inputs cannot
form a valid dynamical system
##
##
## [pass] No quantities were defined multiple times in the inputs
##
## [fail] The following module inputs were not defined:
##  Leaf from the 'maintenance_respiration_calculator' module
##  Leaf from the 'parameter_calculator' module
##  Leaf from the 'senescence_logistic' module
##  Leaf from the 'partitioning_growth' module
##
## [fail] The following differential module outputs were not part of the initial values:
##  Leaf from the 'senescence_logistic' module
##  Leaf from the 'maintenance_respiration' module
##  Leaf from the 'partitioning_growth' module
##
## [pass] There are no cyclic dependencies among the direct modules.
```

Additionally, error messages will be generated if the name of a module or the type of the `ode_solver` cannot be found in the corresponding libraries:

```
soybean_result <- run_biocro(
  soybean$initial_values,
  soybean$parameters,
  soybean_weather$'2002',
  append(soybean$direct_modules, 'BioCro:nonexistent_module'), # add a nonexistent module
  soybean$differential_modules,
  soybean$ode_solver
)

## Error in stats::setNames(.Call(R_module_creators, module_names), names(module_names)):  Caught
exception in R_module_creators:  "nonexistent_module" was given as a module name, but no module
with that name could be found.
```

If the modules are not provided in a suitable order for evaluation, they will be automatically reordered.

There is also a related function (`validate_dynamical_system_inputs`) which takes the same input arguments as `run_biocro` (except `ode_solver`) and checks them for validity without attempting to run the simulation. It returns a boolean indicating validity and prints additional information to the R terminal, such as a list of parameters that are not used by any modules (since such parameters could in principle be removed without affecting the simulation) and a suitable ordering for the direct modules (if reordering is required).

```
# This code is not evaluated here since it produces a large amount of text
valid <- validate_dynamical_system_inputs(
  soybean$initial_values,
  soybean$parameters,
  soybean_weather$'2002',
  rev(soybean$direct_modules), # Reverse the order of the direct modules
  soybean$differential_modules
)
```

3 Viewing and saving the results of a BioCro simulation

The output from the call to `run_biocro` in Section 2.1 is returned as a data frame, an R data structure that was introduced in Section 2.4. In this case, each column represents one of the quantities whose value is determined during the simulation, and each row represents a time point. By default, all quantities except the parameters are included in the output of `run_biocro`, so this data frame also includes the drivers.

Since the `run_biocro` function returns its result in a commonly used R structure, there are many options available for interacting with the data. In the following sections, we demonstrate how to access numerical values from the data frame, how to create plots from its columns, and how to save or export the data frame for future analysis.

3.1 Accessing numerical values

Several options exist for viewing the numerical contents of a data frame. We have already mentioned the `str` and `print` commands in Section 2.4; here we provide a few additional methods.

We can view the results formatted as a table in a new window in the R environment:

```
View(soybean_result)
```

We can store the column names in a vector for later use or viewing:

```
soybean_model_outputs <- colnames(soybean_result)
```

The individual columns can be accessed the same way as the named list elements discussed in Section 2.2. For example, if we want to see the `doy` column, which represents the day of the year (DOY), we could type the following:

```
str(soybean_result$doy)

##  num [1:3288] 152 152 152 152 152 152 152 152 152 152 ...
```

It's also possible to view a subset of the data frame limited to just some columns:

```
str(soybean_result[c('doy', 'hour', 'Leaf')])

## 'data.frame': 3288 obs. of  3 variables:
##  $ doy : num  152 152 152 152 152 152 152 152 152 152 ...
##  $ hour: num  0 1 2 3 4 5 6 7 8 9 ...
##  $ Leaf: num  0.0631 0.0631 0.0631 0.0631 0.0631 ...
```

A similar technique can be used to limit which rows are displayed:

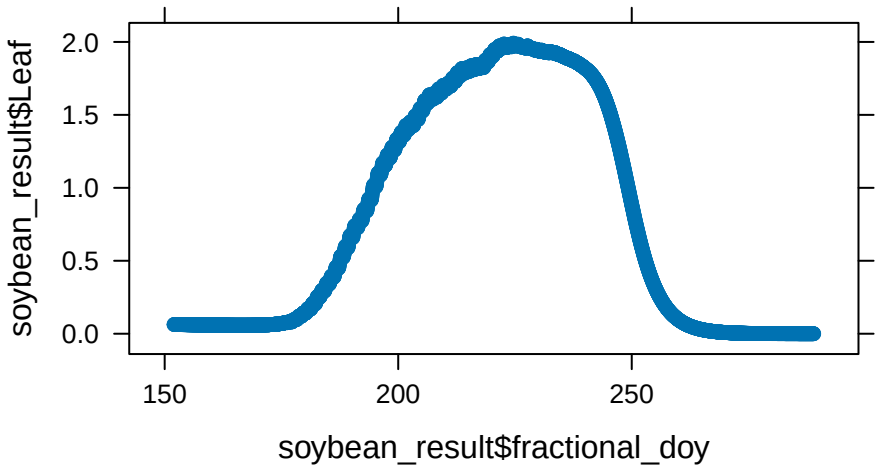
```
str(soybean_result[round(soybean_result$doy) == 250, c('doy', 'hour', 'Leaf')])

## 'data.frame': 24 obs. of  3 variables:
##  $ doy : num  250 250 250 250 250 250 250 250 250 250 ...
##  $ hour: num  0 1 2 3 4 5 6 7 8 9 ...
##  $ Leaf: num  0.885 0.878 0.872 0.866 0.86 ...
```

3.2 Plotting the results of a simulation

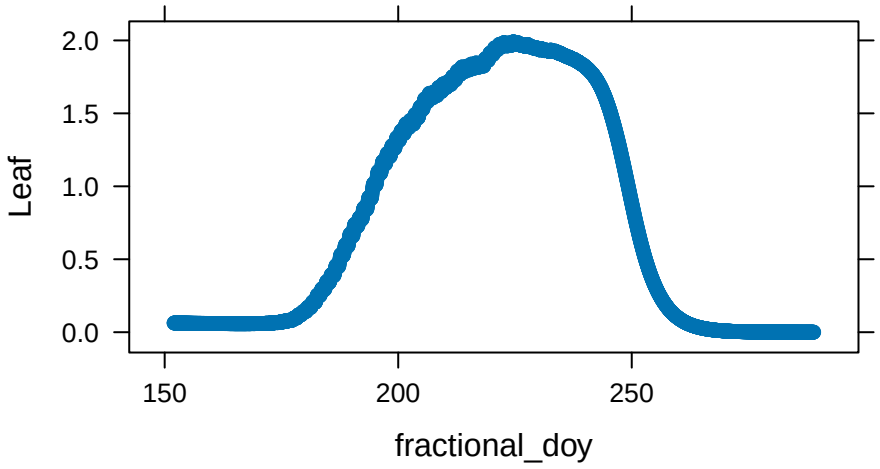
Viewing the numeric results directly from a data frame can be useful at times, but it's often more useful to plot one or more columns from the result against an independent variable such as the fractional day of year. We can do this with the `xyplot` function from the `lattice` package, which must be loaded into the R workspace (Section 1). This function produces an R *trellis* object which can be viewed with the `print` function.

```
soybean_plot_v1 <- xyplot(
  soybean_result$Leaf ~ soybean_result$fractional_doy
)
print(soybean_plot_v1)
```



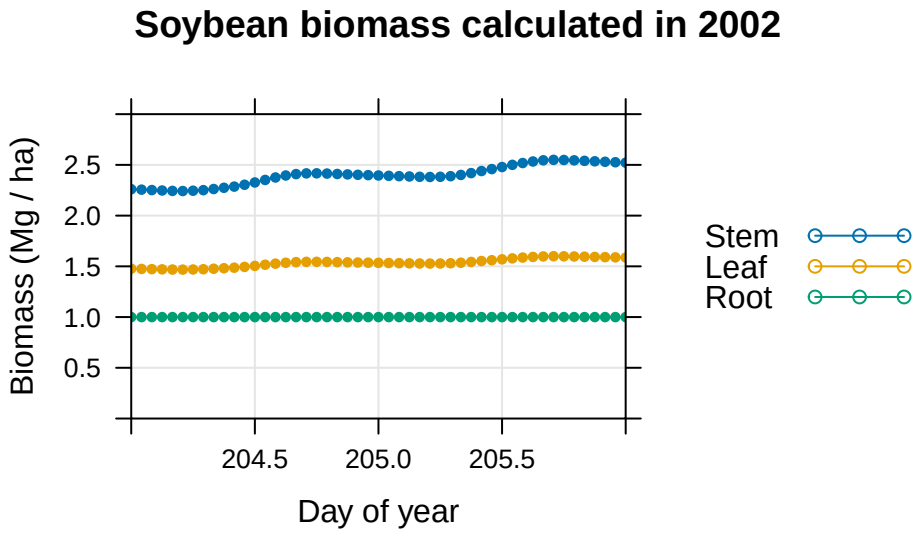
We can avoid repeating the data frame name each time we specify a column by using `xyplot`'s data argument. Notice the effect this has on the axis labels:

```
soybean_plot_v2 <- xyplot(
  Leaf ~ fractional_doy,
  data = soybean_result
)
print(soybean_plot_v2)
```



Here is a more advanced example where we plot multiple organ masses on the y axis, provide axis labels, specify axis ranges, add a legend, etc. Notice that we use two-element vectors to specify the axis limits and a list to specify the legend properties. See Section 6.1 for a discussion about the units used in this figure.

```
soybean_plot_v3 = xyplot(
  Stem + Leaf + Root ~ fractional_doy,          # Specify multiple data series using `+`
  data = soybean_result,                        # Plot data from `soybean_result`
  type = 'b',                                   # Plot using both points and a line (use
                                                # 'l' for just a line or 'p' for points)
  pch = 20,                                     # Use a small solid circle for the points
  ylab = 'Biomass (Mg / ha)',                   # Y label
  xlab = 'Day of year',                         # X label
  auto.key = list(space = 'right'),             # Add a legend on the right side
  grid = TRUE,                                  # Add horizontal and vertical lines
  main = 'Soybean biomass calculated in 2002', # Add a main title
  xlim = c(204, 206),                           # Specify the X axis limits
  ylim = c(0, 3)                                # Specify the Y axis limits
)
print(soybean_plot_v3)
```



3.3 Exporting or saving the results of a simulation

For any serious analysis, it is critical to store either the results themselves or the code that produced them. R offers clear routes for both options:

- R code can be saved to a file with a `.R` extension; this forms a *script* that can be run later using R's `source` command. Saving code that you've been entering directly into the R terminal as a script ensures that you can repeat or modify a previous analysis.
- Any R objects can be saved as R data files using the `save` function; for example, the `soybean_result` data frame calculated in Section 2.1 can be saved to a file with the following command: `save(soybean_result, file=file.choose())`. Here the filename is chosen interactively using the `file.choose` function, and it should have a `.rda`, `.RData`, or `.rdata` extension.
- R data frames can also be saved in plain-text format as comma-separated value (CSV) files using R's `write.csv` function: for example, `write.csv(soybean_result, file=file.choose(), row.names=FALSE)`. Here the filename should have a `.csv` extension. Now the data can be opened with many other pieces of software for analysis or plotting.

4 Visualizing module behavior by calculating a response curve

To understand how a module works, it can be helpful to visualize how one of its output quantities depends on one of its input quantities across a reasonable range—a plot typically called a *response curve*. As an example, here we will calculate the response of the soybean net CO_2 assimilation rate (A_n) to the absorbed photosynthetically active photon flux density (PPFD; Q_{abs}) according to the Farquhar-von-Caemmerer-Berry model for C_3 photosynthesis, which is available in the standard BioCro module library as the `c3_assimilation` module. (This module also uses the Ball-Berry model for stomatal conductance, iteratively solving for consistent solutions to the photosynthesis and conductance equations.)

The first step towards calculating a response curve is to check the module's inputs and outputs, which can be done using the `module_info` function, as discussed in Section 6.2:

```
module_info('BioCro:c3_assimilation')

##
##
```

```
## Module name:
##   c3_assimilation
##
## Module input quantities:
##   atmospheric_pressure
##   b0
##   b1
##   beta_PSII
##   Catm
##   electrons_per_carboxylation
##   electrons_per_oxygenation
##   gbw
##   Gs_min
##   Gstar_c
##   Gstar_Ea
##   Jmax_at_25
##   Jmax_c
##   Jmax_Ea
##   Kc_c
##   Kc_Ea
##   Ko_c
##   Ko_Ea
##   O2
##   phi_PSII_0
##   phi_PSII_1
##   phi_PSII_2
##   Qabs
##   rh
##   RL_at_25
##   RL_c
##   RL_Ea
##   StomataWS
##   temp
##   theta_0
##   theta_1
##   theta_2
##   Tleaf
##   Tp_at_25
##   Tp_c
##   Tp_Ha
##   Tp_Hd
##   Tp_S
##   Vcmax_at_25
##   Vcmax_c
##   Vcmax_Ea
##
## Module output quantities:
##   Assim
##   Assim_check
##   Assim_conductance
##   Ci
##   Cs
##   GrossAssim
```

```
##   Gs
##   RHs
##   RL
##   Rp
##   iterations
##
## Module type (differential or direct):
##   direct
##
## Requires a fixed step size Euler ode_solver:
##   no
```

Here, the `Qabs` input represents the absorbed PPFD Q_{abs}, while the `Assim` output represents the net assimilation rate A_n. So we will be varying `Qabs` while keeping the other input quantities fixed. The values of several other input quantities are specified in `soybean$parameters`. For others, we will need to make a choice based on reasonable field conditions.

To run the module, we will need the `evaluate_module` function, which requires a fully-qualified module name and a named list of input quantities. It attempts to run the module, returning its outputs as a named list. If any of the module's inputs are not provided, a helpful error message will be returned. So we can try passing the `soybean$parameters` list as an input to the `c3_assimilation` module; this will help identify the remaining inputs that we need to define.

```
outputs <- evaluate_module('BioCro:c3_assimilation', soybean$parameters)
```

```
## Error:  The 'BioCro:c3_assimilation' module requires 'Qabs' as an input quantity
##   The 'BioCro:c3_assimilation' module requires 'StomataWS' as an input quantity
##   The 'BioCro:c3_assimilation' module requires 'Tleaf' as an input quantity
##   The 'BioCro:c3_assimilation' module requires 'gbw' as an input quantity
##   The 'BioCro:c3_assimilation' module requires 'rh' as an input quantity
##   The 'BioCro:c3_assimilation' module requires 'temp' as an input quantity
```

Here, as expected, an error occurs because several input quantities are not defined. We can choose values for these and add them to the parameter list:

```
outputs <- evaluate_module(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    rh = 0.7       # dimensionless
    Qabs = 1800    # micromol / m^2 / s
    Tleaf = 27     # degrees C
    gbw = 1.2      # mol / m^2 / s
    StomataWS = 1 # dimensionless; 1 indicates no water stress
    temp = 25      # degrees C
  })
)
```

Choosing reasonable input parameters is sometimes a difficult process, but insight can often be gained by examining weather data, crop parameter lists, the outputs of a BioCro simulation, or published literature. Critical information about particular quantities, such as their meaning and units, can be obtained using the strategies in Section 6.

Since we now have a complete set of inputs, we are ready to calculate a response curve by varying Q and recording corresponding values of A_n. One way to accomplish this goal would be to repeatedly call `evaluate_module` within a `for` loop or using a function like `sapply`. However, since calculating a response curve is a common operation, the BioCro package includes a convenience function for doing so:

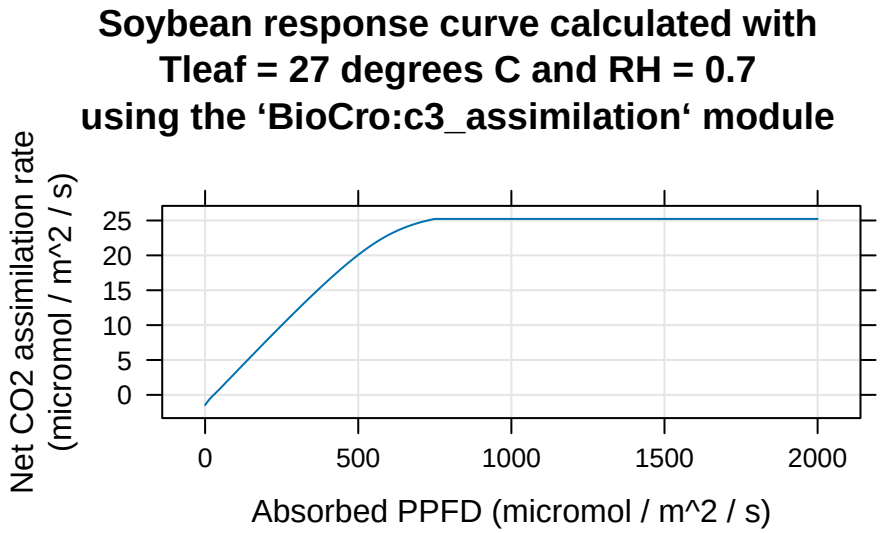
`module_response_curve`. This function requires a fully-qualified module name, a list of fixed input quantities, and a data frame specifying the values of any input quantities that should vary. It returns a data frame with each of the module's inputs and outputs as columns, where each row represents a different set of inputs. This is a convenient format for plotting or saving the results, as discussed in Section 3.

For this example, a response curve can be calculated and viewed as follows, where we generate a plot caption from the information stored in the response curve data frame:

```
rc <- module_response_curve(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    rh = 0.7
    Tleaf = 27
    gbw = 1.2
    StomataWS = 1
    temp = 25
  }),
  data.frame(Qabs = seq(from = 0, to = 2000, length.out = 501))
)

caption <- paste0(
  'Soybean response curve calculated with\nTleaf = ', unique(rc$Tleaf),
  ' degrees C and RH = ', unique(rc$rh), '\nusing the ‘',
  unique(rc$module_name), '‘ module'
)

xyplot(
  Assim ~ Qabs,
  data = rc,
  type = 'l',
  xlab = 'Absorbed PPFD (micromol / m^2 / s)',
  ylab = 'Net CO2 assimilation rate\n(micromol / m^2 / s)',
  main = caption,
  grid = TRUE
)
```



(Note that since the data frame returned by `module_response_curve` includes *all* of the module's outputs,

we could also use `rc` to create plots showing how the stomatal conductance for water (`Gs`) or the intercellular CO_2 concentration (`Ci`) respond to the incident PPFD.)

From this figure, we can learn a few interesting things about the response of the carbon assimilation rate to the amount of incoming light. First, when the light is low, the carbon consumed by respiratory processes exceeds the carbon gained through photosynthetic assimilation, leading to a negative value of the net assimilation rate. An important consequence of this is that although plants take in CO_2 during the day, they exhale CO_2 at night. As light intensity increases, the initial response is linear, although it begins to plateau at higher light intensities; in other words, the additional light energy does not cause a large increase in assimilation when the light intensity is already high.

Although this information is encoded in the equations used to calculate the net assimilation rate, these insights are difficult to obtain by merely viewing the equations, and only become apparent when calculating a response curve using realistic values for environmental conditions and photosynthetic parameters. This example illustrates a very basic response curve calculation; a more advanced analysis could include multiple curves calculated for different values of air temperature or relative humidity. Thus, the ability to quickly and easily calculate response curves from individual model components is another key ability of BioCro, which complements the benefits of modularity that were discussed in Sections 2.1 and 2.2.

A response curve can also be the first step towards a full sensitivity analysis, where numerical derivatives of an output quantity with respect to an input quantity can be calculated. See the *Quantitative Comparison Between Two Photosynthesis Models* vignette for a more detailed analysis including some of these ideas.

5 Module libraries

Currently, there is only one module library available, and it comes packaged as part of the BioCro R package. This "standard library" is named `BioCro`, so all fully-qualified module names currently take the form `BioCro:module_name`. Later, the BioCro R package will only contain functions related to the core framework (such as `run_biocro`), and each module library (including the standard library) will exist as a separate R package, where the name of the module library is set to be the name of its R package. Module libraries will allow researchers to develop their own modules privately and facilitate the creation of modules that can be used for other purposes besides crop growth simulations. Stay tuned for new developments!

6 Getting basic information about quantities and modules

6.1 Quantities

In the final plot in Section 3.2, we specified that the units of the `Stem`, `Leaf`, and `Root` quantities are each Mg / ha (megagrams per hectare). You may be wondering: where did this information come from?

Essential information for each quantity—including its units, modules that use it as an input, and modules that calculate it as an output—can be found at the quantity documentation web site: https://biocro.org/BioCro-documentation/quantity_docs/quantities.html. Visiting this site is the easiest way to find units.

The quantity documentation web site is automatically generated from the module source code. If the code is not formatted properly, the units may not be detected for a particular quantity. In that case, units can be found by directly viewing the code. Here we will demonstrate how to do this for `Leaf`. First, we will need to locate a module which uses `Leaf` as an input or an output, and then we will view its source code.

An appropriate module can be located using the quantity documentation web site or through R commands. To use R commands, we begin by using the `get_all_quantities` function, which is part of the BioCro framework. This function requires the name of a module library as its input argument, and it returns a data frame with three columns: `quantity_name`, `quantity_type`, and `module_name`. Each row in this data frame represents an input or output of one of the modules from the specified module library.

Once we get information about all the BioCro quantities, we can take a subset of them: we just want the rows where the quantity name is `Leaf`. This will give us some possible modules to choose from. In this example, we use the `cat` command with `sep = '\n'` to ensure that each vector element is printed on its own line:

```
all_quantities <- get_all_quantities('BioCro')
leaf_quantity_subset <- all_quantities[all_quantities$quantity_name == 'Leaf', ]
leaf_modules <- unique(leaf_quantity_subset$module_name)
cat(leaf_modules, sep = '\n')

## BioCro:biomass_leaf_n_limitation
## BioCro:example_model_mass_gain
## BioCro:example_model_partitioning
## BioCro:maintenance_respiration
## BioCro:maintenance_respiration_calculator
## BioCro:parameter_calculator
## BioCro:partitioning_growth
## BioCro:senescence_logistic
## BioCro:thermal_time_and_frost_senescence
## BioCro:thermal_time_senescence
## BioCro:total_biomass
```

Now we can see there are several modules that have `Leaf` as an input or output. Let's choose one of them: the `total_biomass` module. We can find its source code in `src/module_library/total_biomass.h`. (See Section 6.3 for more information about how to access the source code.) This is a C++ header file that defines the module. Looking through the code, we can find the units for `Leaf` specified in a comment (Listing 1).

Listing 1: Section of code from `src/module_library/total_biomass.h`.

```
string_vector total_biomass::get_inputs()
{
    return {
        "Grain",           // Mg / ha
        "Leaf",            // Mg / ha
        "LeafLitter",      // Mg / ha
        "Rhizome",         // Mg / ha
        "RhizomeLitter",   // Mg / ha
        "Root",            // Mg / ha
        "RootLitter",      // Mg / ha
        "Shell",           // Mg / ha
        "Stem",            // Mg / ha
        "StemLitter"       // Mg / ha
    };
}
```

6.2 Modules

Basic information about a module—its input quantities, output quantities, and type—can be obtained from within R using the `module_info` function, which prints module info to the R terminal and also (optionally) returns it as a list. Printing to the terminal can be disabled by setting the `verbose` argument to `FALSE`. For an example of printing module info to the terminal, see Section 4; here we demonstrate how to silently store the module info in a list:

```
info <- module_info('BioCro:total_biomass', verbose = FALSE)
str(info)

## List of 6
##  $ module_name           : chr "total_biomass"
##  $ inputs                : chr [1:10] "Grain" "Leaf" "LeafLitter" "Rhizome" ...
```

```
##  $ outputs               : chr [1:2] "total_intact_biomass" "total_litter_biomass"
##  $ type                  : chr "direct"
##  $ euler_requirement     : chr "does not require a fixed-step Euler ode_solver"
##  $ creation_error_message: chr "none"
```

Notice that BioCro modules can be designated as requiring a fixed-step Euler solver; this is to accommodate discrete dynamical systems, which can be represented in BioCro provided that the ODE solver only takes steps of a specified size. An error will occur if `run_biocro` is called with a non-Euler ODE solver when one or more of its modules requires an Euler ODE solver.

A list of all modules available within a particular module library can be obtained with the `get_all_modules` function, and detailed information about an individual module can be obtained by viewing its source code or the associated Doxygen documentation; see Section 6.3.

6.3 Accessing the source code

There are several options for viewing BioCro's source code:

- BioCro is available as a public GitHub repository, so all of its source code can be viewed via a web browser at https://github.com/biocro/biocro.
- The source code can also be downloaded from the GitHub repository and viewed locally via a user's preferred methods.
- Doxygen documentation of all the code or various subsets of the code can be viewed at the BioCro documentation website https://biocro.org.

Many users will only be interested in the source code for the modules, which is located in the `src/module_library` directory. Most of the time, the source code for a module called `module_name` will be contained in a file called `src/module_library/module_name.h` or `src/module_library/module_name.cpp`. However, there are a few cases where the file name does not match the module name. In this case, the code can be quickly located by searching for the module's name within the directory using `grep` or another search method; alternatively, each module is present as a class within the Doxygen documentation (see Section 6.3.1).

6.3.1 Viewing module code in the Doxygen documentation

Looking directly at the code is the most complete way to learn about a module, but it might not always be the best way, since the module's equations are surrounded by boilerplate code required by the C++ framework. As an alternative to reading the code itself, most modules have detailed descriptions written as Doxygen-style comments that include some background information, the equations used by the module, and references to published literature that describe the equations and concepts used in the module. Nicely-formatted versions of these description comments are available at the BioCro documentation website, as mentioned above.

For example, part of the Doxygen-style comment for the `thermal_time_linear` module is shown in Listing 2. In the nicely-formatted version, this is rendered as shown in Figure 1.

Listing 2: Snippet of Doxygen-style comment from the `thermal_time_linear` module.

```
/**
 * This module implements the most basic model, which is discussed in many places,
 * e.g. section 2.7 of Campbell & Norman (1998). In this model, `DR` is determined
 * from the air temperature `T` according to:
 *
 * | DR            | T range        |
 * | :-----------: | :------------: |
 * | `0`           | `T <= T_base`  |
 * | `T - T_base`  | `T_base < T`   |
 *
 * Thermal time has units of `degrees C * day` and the development rate, as written
 * here, has units of `degrees C * day / day = degrees C`. This is a common formulation,
```

```
 * reflecting the fact that average daily temperatures are often used to calculate the
 * increase in thermal time during an entire day.
 *
 * This model is based on the observation that once the air temperature exceeds a
 * threshold, development begins to proceed linearly. However, it is known that this
 * trend cannot continue indefinitely and this model tends to overestimate development
 * at high temperatures.
 */
```

This module implements the most basic model, which is discussed in many places, e.g. section 2.7 of Campbell & Norman (1998). In this model, `DR` is determined from the air temperature `T` according to:

DR	T range
0	`T <= T_base`
`T - T_base`	`T_base < T`

Thermal time has units of `degrees C * day` and the development rate, as written here, has units of `degrees C * day / day = degrees C`. This is a common formulation, reflecting the fact that average daily temperatures are often used to calculate the increase in thermal time during an entire day.

This model is based on the observation that once the air temperature exceeds a threshold, development begins to proceed linearly. However, it is known that this trend cannot continue indefinitely and this model tends to overestimate development at high temperatures.

Figure 1: Nicely-formatted version of the comment in Listing 2, as rendered by Doxygen.

The Doxygen documentation also includes the code itself in addition to the nicely-formatted comments.

Besides the nice formatting, there is another advantage to viewing the Doxygen documentation: linking between files and classes. There are links to the source code from the module class description page even when the file name doesn't match the module class name. For example, the `ten_layer_rue_canopy` class page includes a link to the source code file `multilayer_rue_canopy.h` where it is defined.

7 Tips and tricks

7.1 Writing more efficient R code

7.1.1 Reducing duplication with the `with` command

In Section 2 and several other places in this document, elements of pre-defined crop model definition lists like `soybean` were passed to BioCro functions like `run_biocro`. These commands often involve repeating the crop model name multiple times, such as when the `soybean_result` data frame was calculated in Section 2.1. In that command, `soybean$` appears five times.

There is an alternative way to formulate these commands where the R function `with` can be used to reduce the duplicated typing. For example, the soybean simulation in Section 2.1 can be run as follows:

```
soybean_result <- with(soybean, {run_biocro(
  initial_values,
```

```
  parameters,
  soybean_weather$'2002',
  direct_modules,
  differential_modules,
  ode_solver
)})
```

Besides shortening the code, using `with` also makes it easy to modify a command to simulate the growth of another crop. Such a switch may only require one change to the command, where the single instance of the name of a crop is replaced by another. (Additional changes may be required if the drivers need to be different for the new crop.) For more examples, see the help page for `crop_model_definitions` by typing `?crop_model_definitions` in R.

7.1.2 Modifying lists on-the-fly with `within` and `append`

Sometimes it may be useful to make a small change to one of the pre-set model definition components; for example, the value of a parameter might need to be changed, or an additional module may be required. In these situations, `within` and `append` can be very handy.

The `within` command allows us to modify the named elements of a list or vector, returning a new one. For example:

```
# Create a small list
original_list <- list(a = 1, b = 2, c = 3)

# Create a new list from the original one by removing the `a` element and
# changing the value of the `c` element
new_list <- within(original_list, {
  rm(a)
  c = 4
})

# We don't need to actually store the new list; instead we can pass it directly
# to another function. Here we perform the same operations (but separate them
# with `;` instead of writing them on separate lines) and pass the result
# directly to `str` without storing it as a named object.
str(within(original_list, {rm(a); c = 4}))

## List of 2
##  $ b: num 2
##  $ c: num 4
```

The `append` command allows us to add an unnamed element to a list or vector, returning a new one. For example:

```
str(append(original_list, 5))

## List of 4
##  $ a: num 1
##  $ b: num 2
##  $ c: num 3
##  $  : num 5
```

Both of these commands are used throughout this document; for examples, see Section 2.6.

7.2 Additional R commands and resources

There are a few R commands that may be useful but weren't specifically mentioned elsewhere in this guide:

- At any time, it's possible to see all the objects that have been created during an R session by using the `ls` command: `ls()`.
- If you want to clear out your session, you can delete all the objects with the `rm` command as follows: `rm(list=ls())`.
- It is possible to check a variable's type using the `class` command, for example: `class(c('doy', 'hour', 'Leaf'))` will return `character`, meaning that `c('doy', 'hour', 'Leaf')` produces a character vector. On the other hand, `class(soybean$parameters)` will return `list`.

New users might find it useful to consult online guides to R, rather than relying R's help system. The following topics would be particularly helpful to anyone using BioCro to run simulations or investigate module behavior:

- **Data structures**: This topic refers to basic object types like lists, vectors, and data frames. We recommend https://adv-r.hadley.nz/vectors-chap.html.
- **Subsetting**: This topic refers to selecting subsets of R data frames and matrices. We recommend https://www.statmethods.net/management/subset.html and https://adv-r.hadley.nz/subsetting.html.
- **Vector arithmetic**: This topic refers to performing arithmetic operations on each member of one vector, forming a new vector. We recommend https://pubs.wsb.wisc.edu/academics/analytics-using-r/vector-math.html and https://www.r-tutor.com/r-introduction/vector/vector-arithmetics.
- **Apply-type functions**: This topic refers to several options for applying a function to each member of a vector or list. We recommend https://www.guru99.com/r-apply-sapply-tapply.html.
- **Lattice graphics**: This topic refers to generating plots using the `lattice` package, which includes many other plotting tools besides `xyplot`. We recommend https://www.statmethods.net/advgraphs/trellis.html for an overview and https://homerhanumat.github.io/tigerstats/xyplot.html for `xyplot`.
- **Scripts**: The R help file for the `source` command (available by typing `?source`) has a great explanation for how the command works and how it differs from entering code directly into the console. Online guides are available with advice about best practices for writing R scripts, such as this one: https://swcarpentry.github.io/r-novice-inflammation/06-best-practices-R/.

8 Final remarks

So now we've gone through two important examples: running a full simulation for a crop and calculating response curves for a module using different parameter values. We've also demonstrated how the results can be visualized and and saved. Along the way, we also discussed a few important data types in R.

It's important to note that the techniques shown here can be extended to other crops and modules. For an in-depth example, see the *Quantitative Comparison Between Two Photosynthesis Models* vignette.

To test your understanding, you could try to complete the following tasks:

- Run the soybean simulation a few times with several different values of a key parameter (such as `Vcmax_at_25`, which represents the maximum rate of carboxylation $v_{c,max}$) and compare plots of the time evolution of the `Grain` quantity (representing the pod mass per unit area) in each of the scenarios. (See Figure 2 for a possible solution.)
- Calculate a temperature response curve for soybean assimilation, in analogy to the light reponse curve we calculated in Section 4. (See Figure 3 for a possible solution.)

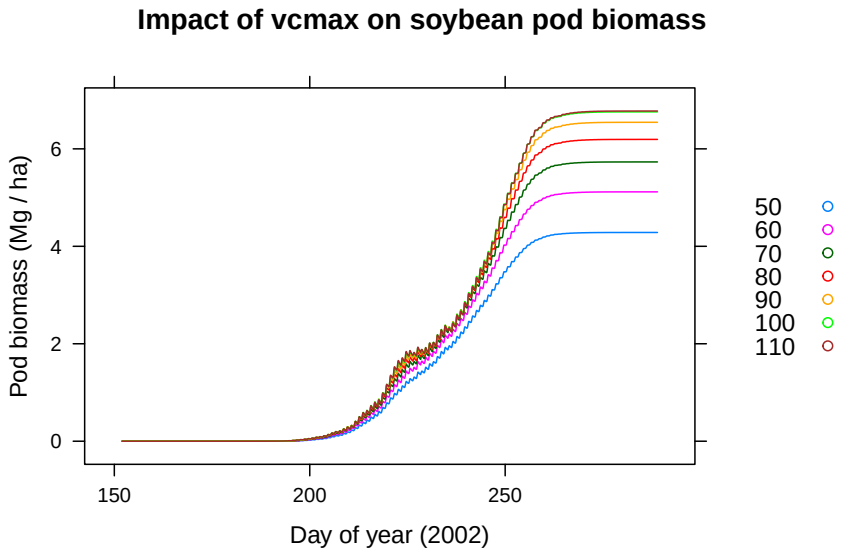


Figure 2: Soybean pod biomass throughout 2002 simulated with different values of $v_{c,max}$.

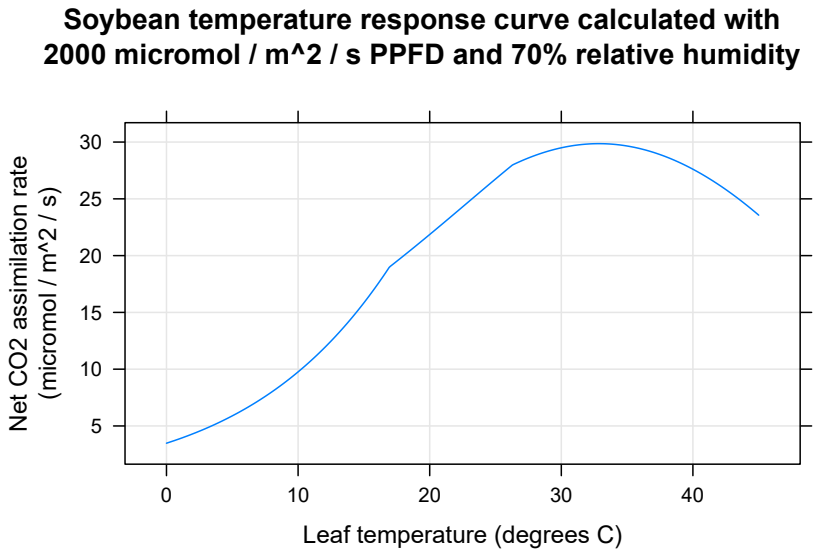


Figure 3: Response of soybean net assimilation to leaf temperature.

Good luck, and have fun exploring plant growth models using BioCro!