BioCro Workshop I: Using BioCro with the Standard Module Library

BioCro Development Team

February 9, 2026


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

Commit Hash: f66ccf7
Date: Mon, 9 Feb 2026 11:29:47 -0600
Branch:
```

Contents

1 About this Workshop

The main goal of this workshop is to demonstrate what can be done with the existing models available in BioCro, and especially to exhibit tools that can be used to gain more understanding. To achieve this, we will go through several examples showing how to use the most important BioCro functions, culminating with an example of parameterizing a model using measured data.

In this workshop, we will assume that you have a working R installation and are familiar with some basic R terminology, such as *vector*, *list*, and *data frame*. While some examples will use terms from mathematics, computer science, or plant physiology, it is not necessary to be fully familiar with these ideas in order to progress through the workshop. When possible, references to other resources will be provided so you can learn more about these topics if you'd like to.

This workshop was originally held on May 13, 2022 as part of the Crops in Silico Symposium. Since then, it has been updated following changes to BioCro syntax and other new developments. To use the workshop on your own, try running the code snippets on your machine, and complete the exercises to test your understanding.

Throughout this document, look out for the following types of boxes, which will provide instructions and other key pieces of information:

Code to run in R

This indicates a code snippet that should be run in R. Copy it from this document and paste it into the R terminal.

Code to (optionally) run in R

This indicates a code snippet that can optionally be run in R.

Expected output from R

This will come after a code snippet, and will show the expected output from the code. Use this to check whether you have successfully run the code.

Exercise

This will be a suggestion for a task you should be able to complete, and will often be based on one or more previous code snippets.

Learn more

This will be a suggestion for where to learn more about a particular topic that may only be covered briefly in this workshop.

2 Introduction

2.1 Essential information about BioCro

- BioCro is a software package for modular crop growth simulations.
- It can also be thought of as a general-purpose tool for defining and solving systems of ordinary differential equations (Section 2.2).
- We have published a paper describing BioCro in *in silico Plants* (Lochocki et al. 2022), and there is also a press release about the paper with some additional information.
- BioCro is written in `C++` but has a convenient `R` interface that we will be using today. R provides many possible avenues for inputting data and analyzing results.
- The BioCro R package is available on CRAN, so it can be installed from within R by typing `install.packages('BioCro')`.
- BioCro is free and open-source, with the full source code available via a public GitHub repository: `https://github.com/biocro/biocro`.
- Automated tests ensure that the code does not break following any changes made by the BioCro development team.
- All BioCro documentation can be accessed at the BioCro web site(`https://biocro.org`).

2.2 What is a system of differential equations?

When learning about BioCro, it is helpful to understand a few basic principles of ordinary differential equations.

First, a *derivative* is a mathematical quantity that expresses the instantaneous rate of change of one variable with respect to another. In BioCro, we almost exclusively deal with time rates of change; in other words, the instantaneous rate of change of X with respect to time. The time derivative of X is typically written as either $\frac{dX}{dt}$ (where t refers to time) or $\dot{X}$ (where the dot refers to a derivative with respect to time). For example, if M_{leaf} is the leaf mass, then $\frac{dM_{leaf}}{dt}$ and $\dot{M}_{leaf}$ are different ways of writing the instantaneous rate of change of leaf mass with respect to time.

An *ordinary differential equation* (ODE) is an equation that calculates the value of a derivative using a function of time and possibly other parameter values. For example, if the rate of leaf growth depends on the leaf mass and the incident light (Q_{in}), we could express this as an ordinary differential equation:

$$\frac{dM_{leaf}}{dt} = f\left(M_{leaf}, Q_{in}\right),$$

where f is a function.

A *system of ODEs* is simply a collection of related ODEs that are typically coupled to each other. For example, in a BioCro model, the rate of root and leaf growth may depend on the leaf mass, while the rate of water loss from the soil may depend on the root mass. A realistic crop model will be composed of many variables and ODEs.

If we know the initial values of all the variables in the system, then we can use the ODEs to calculate the instantaneous rates of change of each variable, and predict the values of the variables at a future time. This situation is called an *initial value problem* (IVPs), and many numerical algorithms have been developed for solving IVPs.

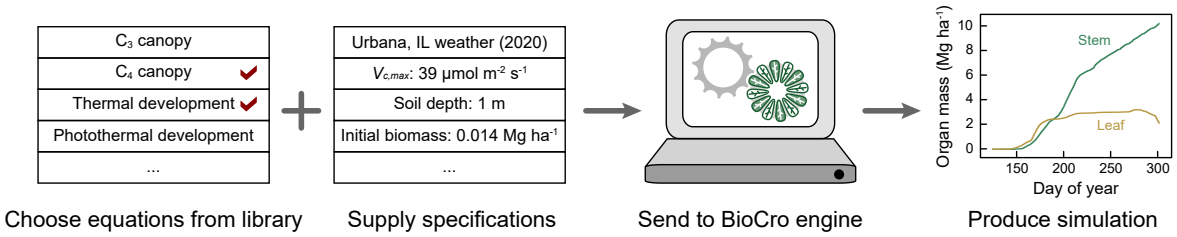
In many scientific fields, models are available for calculating the rates of change for key variables, making ODEs extremely useful for understanding observations and making predictions.

2.3 What do we mean by modular modeling?

When writing code for modeling crop growth using differential equations, a couple of problems commonly appear:

- Many crops share components, so code is duplicated in multiple places.
- There are different ways to model the same process, but there isn't a way to easily choose between alternate versions.

Named sets of equations provide a way to easily select and reuse pieces of models, and they are a central feature of BioCro that allows us to avoid these issues. In BioCro, we refer to these named sets of equations as *modules*. Because larger models are formed by combining modules together, we also sometimes refer to them as *model components*. Modules will be discussed in more detail later in the workshop.



2.4 How did we get here?

BioCro began its life in the early 1990s as a program called WIMOVAC (Humphries and Long 1995). Crops were added over the years and the software was rewritten in C, eventually becoming BioCro (Miguez et al. 2012).

Several years ago, Justin McGrath began modifying BioCro to reduce duplicated code and make it more flexible, resulting in the new version of BioCro we will be using today (Lochocki et al. 2022).

The primary developers in recent years include Justin McGrath, Ed Lochocki, Yufeng He, Scott Oswald, Scott Rohde, Deepak Jaiswal, Megan Matthews, Fernando Miguez, and Steve Long.

For more information about how the original code was made modular, see Section 9.

2.5 What does BioCro enable us to do?

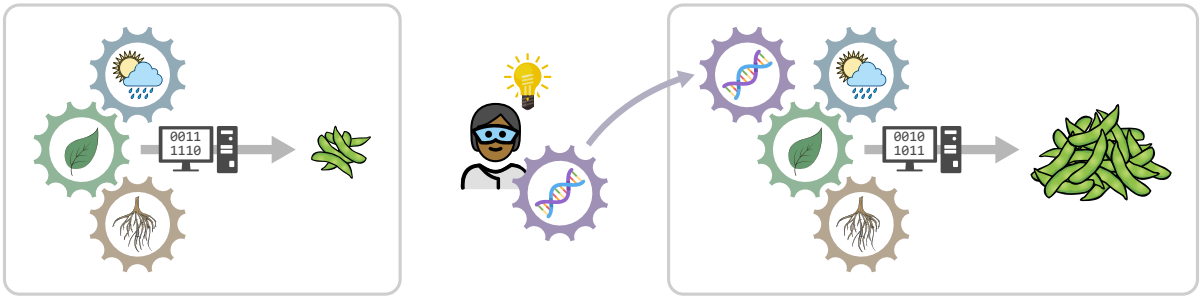
The latest versions of BioCro were created with a few important philosophical tenets in mind:

- Models are sets of equations.
- No equation is special.

This mindset has produced a flexible structure where model components can be added, removed, swapped, or redefined in a straightforward way. This makes BioCro particularly well-suited for operations like the following:

- **Sensitivity analysis**: Identify parameters that are most important for determining certain outputs.
- **Parameterization**: Find parameter values that best reproduce the available experimental data.
- **Quanatitative comparisons between alternate versions of model components**: For example, by quantifying which carbon allocation model best reproduces the available experimental data.
- **Sensor integration**: Replace calculation of a variable with measurements from a sensors.
- **Predictive power**: Quantify changes in accuracy as model components are updated or replaced.

- **Learning and teaching**: Examine model components on their own or in the context of a larger model.



2.6 What we'll be covering today

This workshop is focused on showing how BioCro's modular approach to modeling works in practice. We will begin by looking at individual model components and then move into complete crop models. We will also show a few key techniques that can be used to understand or alter BioCro models.

There are a few things we will not be covering today; these will be addressed in future workshops:

- We will not cover the process of adding new modules to BioCro.
- We will not cover the process of creating a personal module library, although basic instructions can be found at `https://github.com/biocro/skelBML`.
- Although we will briefly discuss the important processes included in a typical BioCro simulation, we will not take detailed look into the rationale for these model components, the assumptions underlying them, or the exact equations that they use.

Many workshop examples will be based on the Soybean-BioCro model, which was originally described in Matthews et al. 2022.

3 Installing and loading R packages

For this workshop, we will be using the BioCro R package, the `lattice` package (which provides tools for creating plots), the `dfoptim` package (which provides tools for optimization), and the `deSolve` package (which provides tools for solving differential equations). We will load them now before moving on to the examples.

Code to run in R

```
# Load required packages
library(BioCro)
library(lattice)
library(dfoptim)
library(deSolve)

# Check the BioCro version
expected_version  <- '3.3.1'
installed_version <- as.character(packageVersion('BioCro'))

if (compareVersion(expected_version, installed_version) != 0) {
  warning(
    'This script was written for BioCro version ', expected_version,
    ' but version ', installed_version,
    ' is installed; this may cause unexpected errors to occur.'
  )
}
```

Expected output from R

There shouldn't be any output from this code.

If you see a warning about the version of BioCro, or if any of the packages are not available, make sure the latest version of each package is installed using the following code.

Code to (optionally) run in R

```
# Make sure packages are installed
install.packages('BioCro')
install.packages('lattice')
install.packages('dfoptim')
install.packages('deSolve')
```

4 Examining a single module

4.1 Getting basic module properties

In BioCro, equations are grouped into *libraries* of *modules*.

- A module represents a set of related equations, while a module library contains a set of related modules.
- Modules come in two types: *differential* modules calculate the derivatives of their outputs with respect to time, while *direct* modules calculate their outputs directly.

Learn more

For more information about modules, please see Lochocki et al. 2022, as well as our articles titled *A Practical Guide to BioCro* and *BioCro as a Dynamical System*, available on the BioCro web site.

For each module, the variables associated with its equations can be thought of as its inputs and outputs.

Example: The C_3 assimilation module from the standard module library represents the Farquhar-von-Caemmerer-Berry (FvCB) model for C_3 photosynthesis coupled with the Ball-Berry equation for stomatal conductance. It needs values of light intensity, leaf temperature, Rubisco kinetic parameters, and several other variables. It uses several equations to calculate the CO_2 assimilation rate, stomatal conductance to H_2O, and internal CO_2 concentration from those inputs.

Learn more

To learn more about the FvCB model, see von Caemmerer 2000 and Lochocki and McGrath 2025. To learn more about the Ball-Berry model and how it can be coupled with the FvCB model, see our article titled *Using the Ball-Berry Model*, available on the BioCro web site.

We can obtain basic information about a module from using the `module_info` function, which requires a module name as an input.

- In BioCro, modules are identified by *fully-qualified* names that include the library name and the local name of the module within the library formatted like `library_name:local_name`.
- Currently, there is only one official BioCro module library—the standard module library—whose name is `BioCro`. However, many users have their own personal libraries, especially when working on new modules that should be kept private because they are not yet published.

Here we use `module_info` to obtain basic information about a very simple module from the standard module library.

Code to run in R

```
module_info('BioCro:total_biomass')
```

Expected output from R

```
##
##
## Module name:
##   total_biomass
##
## Module input quantities:
##   Grain
##   Leaf
##   LeafLitter
##   Rhizome
##   RhizomeLitter
##   Root
##   RootLitter
##   Shell
##   Stem
##   StemLitter
##
## Module output quantities:
##   total_intact_biomass
##   total_litter_biomass
##
## Module type (differential or direct):
##   direct
##
## Requires a fixed step size Euler ode_solver:
##   no
```

Exercise

Type `?module_info` in R to access the help page for this function, and then try changing the value of `module_info`'s optional input argument.

Note that the `?` command can be applied to all functions mentioned in this workshop, and the help pages are a key resource for understanding how to use each function.

4.2 Getting a list of all modules

The standard module library contains many modules.

To see them all, we can use the `get_all_modules` function, which requires the name of a module library.

Then, the `module_info` function can be used to get basic information about any of them.

Here we use `get_all_modules` to view a list of all modules in the standard module library.

Code to run in R

```
print(get_all_modules('BioCro'))
```

Expected output from R

The output is too long to reproduce here, but it should be a list of many module names, each prefixed by `BioCro`.

Exercise

Pick another module from the list, such as `BioCro:c3_assimilation`, and get its basic information.

4.3 Getting more information about a module

The module information available from within R is missing some important parts:

- The meaning of a module's inputs and outputs.
- The units for a module's inputs and outputs.
- A description of the module's purpose.
- The equations it actually uses.

This information can be found in the source code for each module, which is available online at the BioCro GitHub repository or via the annotated Doxygen documentation. For example:

- A short description of the `c3_assimilation` module can be found on its associated class overview page in the Doxygen documentation.
- The units for its inputs and outputs can be found as comments in it source code, available on its associated source code page in the Doxygen documentation (see the comments near `get_inputs()` and `get_outputs()`).
- The code implementation of its equations can also be found in the source code (see `do_operation()`).

The Doxygen documentation is useful, but can be overwhelming. Here is some advice that may be helpful when using it:

- To find a particular module, search for its local name (not its fully-qualified name) using the bar in the upper right corner. For example, search for `c3_assimilation`, but not `BioCro:c3_assimilation`. Generally, two or three results will come up. Results ending with `.h` or `.cpp` are source code pages, while results that do not have a file extension are class overview pages. When both `.h` and `.cpp` pages are available, the `.cpp` page is typically more useful.
- Stay focused on the three sections outlined above. The module description can be found on its class overview page. The inputs and outputs can be found in the `get_inputs()` and `get_outputs()` part of the source code page. The code implementation of equations can be found in the `do_operation()` part of the source code page.
- There are many other pieces of information available in the Doxygen documentation, and many other parts of the source code besides the sections highlighted above. Just ignore these when you are getting started. It is important to understand the other parts when creating modules, but not when getting basic information about a module.

Caveat: Fully documenting the modules is a long and time consuming process, so some modules have incomplete or nonexistent documentation at the moment.

4.4 Evaluating modules

4.4.1 Evaluating a module once

Analogous to evaluating an expression of code, we can evaluate a module's equations given a set of inputs.

This can be accomplished from `R` using the `evaluate_module` function, which requires the name of a module and a set of input values. It returns a list of the module's output values.

For example, we can run the `c3_assimilation` module using default soybean values for most of its inputs using the following command, which also specifies the values of a few module inputs that are not included in the default soybean values.

Code to run in R

```
evaluate_module(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    Qabs = 1800     # absorbed PPFD in micromol / m^2 / s
    StomataWS = 1   # 1 indicates no water stress, 0 indicates maximum water stress
    temp = 25       # air temperature in degrees C
    Tleaf = 27      # leaf temperature in degrees C
    rh = 0.7        # relative humidity (dimensionless)
    gbw = 2.0       # boundary layer conductance to H2O in mol / m^2 / s
  })
)
```

Expected output from R

```
## $Assim
## [1] 25.66557
##
## $Assim_check
## [1] -6.07514e-13
##
## $Assim_conductance
## [1] 103.1757
##
## $Ci
## [1] 279.906
##
## $Cs
## [1] 355.0091
##
## $GrossAssim
## [1] 32.67228
##
## $Gs
## [1] 0.5467807
##
## $iterations
## [1] 8
##
## $RHs
## [1] 0.7030636
##
## $RL
## [1] 1.45942
##
## $Rp
## [1] 5.547285
```

Exercise

Try changing some of the input values or running a different module.

4.4.2 Calculating and viewing a module response curve

In the previous example, a single value was provided for each of the module inputs, so each output also had a single value.

It is also possible to provide a sequence of input values and run the module for each of them.

This can be a useful way to see how one of a module's outputs depends on one of its inputs; the result from this type of calculation is typically called a *response curve*.

This can be done using the `module_response_curve` function, which requires us to specify inputs that should remain constant and inputs that should be varied.

For example, we can calculate a light response curve using the `c3_assimilation` module.

Code to run in R

```
light_response_curve <- module_response_curve(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    StomataWS = 1   # 1 indicates no water stress, 0 indicates maximum water stress
    temp = 25       # air temperature in degrees C
    Tleaf = 27      # leaf temperature in degrees C
    rh = 0.7        # relative humidity (dimensionless)
    gbw = 2.0       # boundary layer conductance to H2O in mol / m^2 / s
  }),
  data.frame(Qabs = seq(0, 2000, length.out = 101))
)
```

Expected output from R

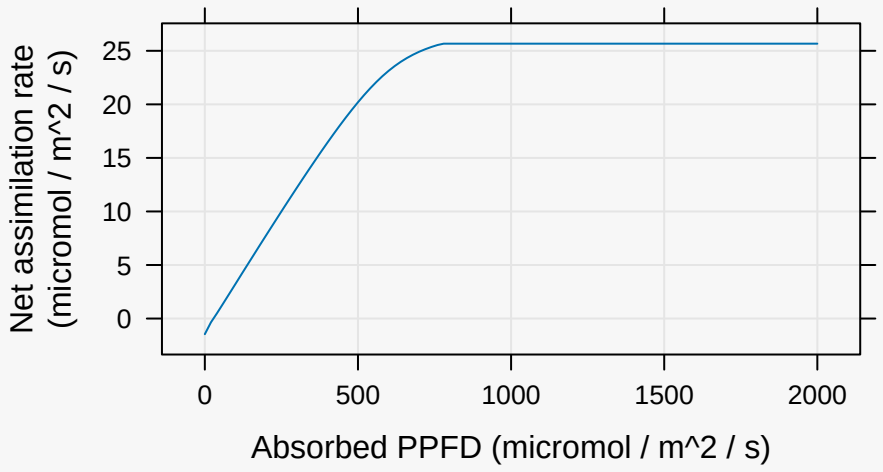
There shouldn't be any output from this code.

The return value is a data frame (an R structure equivalent to a table), and the response of net assimilation to absorbed light intensity can be plotted from it.

Code to run in R

```
light_response_curve_plot <- xyplot(
  Assim ~ Qabs,
  data = light_response_curve,
  type = 'l',
  grid = TRUE,
  xlab = 'Absorbed PPFD (micromol / m^2 / s)',
  ylab = 'Net assimilation rate\n(micromol / m^2 / s)'
)
print(light_response_curve_plot)
```

Expected output from R



The output from `module_response_curve` includes *all* of the module's outputs, so we could also plot the response of another output using the same data frame.

Exercise

Try plotting the response of stomatal conductance (`Gs`) to light intensity.

For another example demonstrating how to calculate multiple response curves at once, see Section 8.2.

5 Combining modules to form a larger model

5.1 Running a soybean growth model for one season

A key part of BioCro is the ability to link modules together in chains where the output from one module can be used as the input to another.

Example: the value of the `StomataWS` input to the `BioCro:c3_canopy` module could be calculated using the `BioCro:stomata_water_stress_linear` module since `StomataWS` is an output of that module.

This is the main way we can construct complex BioCro models from simpler model components.

If a model includes differential modules, we can solve the resulting set of coupled ordinary differential equations to find the model's state at specific time points.

The main BioCro function, `run_biocro`, does the work of combining the modules and integrating the rates of change for us.

For example, we can run a soybean growth model using weather data from Champaign, Illinois during 2002.

Code to run in R

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

Expected output from R

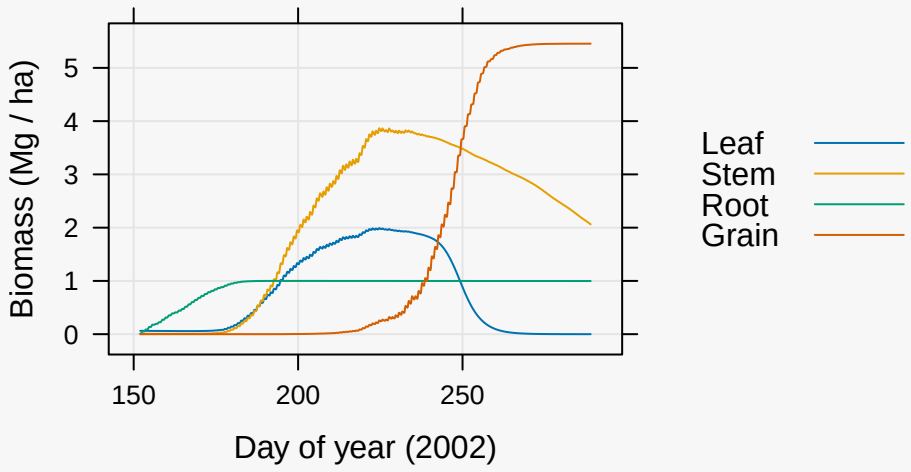
There shouldn't be any output from this code.

As with `module_response_curve`, the return value from `run_biocro` is a data frame. We can view it directly as a table, or plot some of its columns against each other. Here we plot the calculated biomass values against time.

Code to run in R

```
soybean_biomass_2002_plot <- xyplot(
  Leaf + Stem + Root + Grain ~ fractional_doy,
  data = soybean_result,
  type = 'l',
  grid = TRUE,
  auto.key = list(space = 'right'),
  xlab = 'Day of year (2002)',
  ylab = 'Biomass (Mg / ha)'
)
print(soybean_biomass_2002_plot)
```

Expected output from R



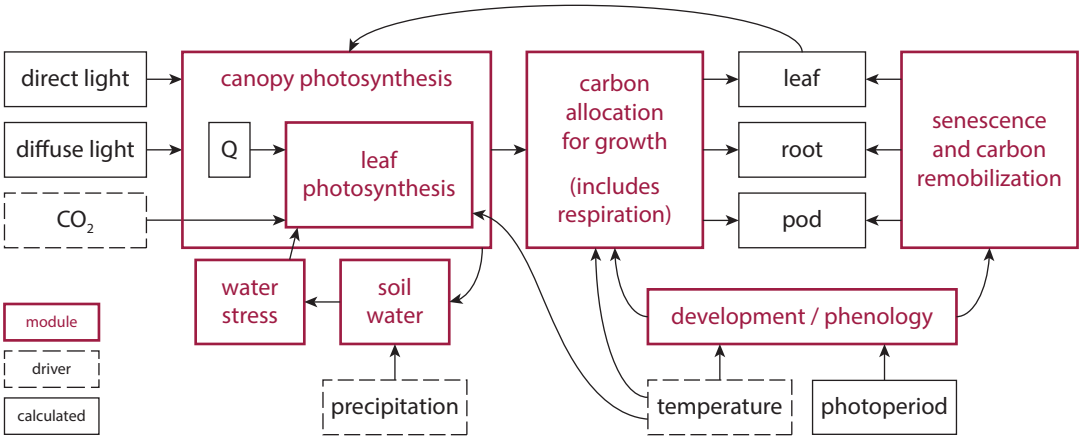
Exercise

Try plotting the soybean biomass against its developmental index (the `DVI` column) instead of time, or try plotting some other outputs like leaf area index (the `lai` column), canopy assimilation rate (the `canopy_assimilation_rate` column), or the degree of water stress (the `StomataWS` column).

5.2 Components of a typical BioCro crop growth model

Although the BioCro framework is a general-purpose tool for building and solving models, the standard module library is specialized for crop growth models that follow a similar organizational structure and contain many similar components.

In fact, it is possible to represent most BioCro crop models at a high level using a single diagram (subject to some necessary crop-specific modifications).



At the code level, each of these important processes is implemented via one or more modules.

The particulars may differ from crop to crop; see below for a list of all modules in Soybean-BioCro.

Code to run in R

```
cat('Direct modules used in the soybean model:\n')
cat(as.character(soybean$direct_modules), sep = '\n')
cat('\nDifferential modules used in the soybean model:\n')
cat(as.character(soybean$differential_modules), sep = '\n')
```

Expected output from R

```
## Direct modules used in the soybean model:
## BioCro:maintenance_respiration_calculator
## BioCro:senescence_coefficient_logistic
## BioCro:stefan_boltzmann_longwave
## BioCro:partitioning_coefficient_logistic
## BioCro:leaf_water_stress_exponential
## BioCro:sla_linear
## BioCro:parameter_calculator
## BioCro:height_from_lai
## BioCro:canopy_gbw_thornley
## BioCro:soil_evaporation
## BioCro:stomata_water_stress_linear
## BioCro:format_time
## BioCro:soybean_development_rate_calculator
## BioCro:solar_position_michalsky
## BioCro:shortwave_atmospheric_scattering
## BioCro:incident_shortwave_from_ground_par
## BioCro:ten_layer_canopy_properties
## BioCro:ten_layer_c3_canopy
## BioCro:ten_layer_canopy_integrator
## BioCro:carbon_assimilation_to_biomass
## BioCro:partitioning_growth_calculator
##
## Differential modules used in the soybean model:
## BioCro:senescence_logistic
## BioCro:maintenance_respiration
## BioCro:partitioning_growth
## BioCro:two_layer_soil_profile
## BioCro:development_index
## BioCro:thermal_time_linear
```

5.3 Brief overview of the Soybean-BioCro modules

A full description of Soybean-BioCro, including the assumptions underlying each component and the rationale for using them, would require at least one textbook and at least one workshop.

Some of this information can be found in the publication describing the model (Matthews et al. 2022). There are also several textbooks that can be helpful resources for understanding the many components involved in BioCro's crop growth simulations (Campbell and Norman 1998; Monteith and Unsworth 2013; Thornley and Johnson 1990).

Here, we will give a short overview of the major components of Soybean-BioCro, explaining how the modules relate to the conceptual diagram. Many details about the module equations have been omitted, but the module code and behavior can be examined using the methods described previously (Section 4).

- As the main driver of plant growth, photosynthesis plays a prominent role in any BioCro model, and many of the soybean modules are related to photosynthesis. Mechanistic models of photosynthesis calculate a carbon assimilation rate from inputs like the absorbed light intensity and intercellular CO_2 concentration, so they can only be applied at scales where these inputs are relatively constant—this is typically the leaf scale or smaller. So, in a crop growth model, leaf-level photosynthesis equations must be embedded in a canopy photosynthesis model where conditions like temperature, humidity, and light are allowed to vary within the canopy. In the soybean model, conditions in individual layers of the canopy are calculated using

the ten_layer_canopy_properties module, leaf-level assimilation and transpiration rates are calculated for each layer using the ten_layer_c3_canopy module, and then the leaf-level rates are added across layers by the ten_layer_canopy_integrator module to find the canopy-level rates.

- Mechanistic photosynthesis models respond to conditions specific to the leaf, such as its temperature and intercellular CO_2 concentration; however, these conditions are generally unknown beforehand since sensor data usually only includes properties of the ambient air surrounding the canopy. For this reason, it is necessary to include the the processes of heat and gas exchange between the leaf and its environment. Gas exchange is regulated by stomatal opening, which is typically modeled using empirical relationships; in BioCro, we use the Ball-Berry model. Heat exchange (and, ultimately, leaf temperature), is determined by conservation of energy; in BioCro, we use "the Penman-Monteith equation" to calculate leaf temperature, which implements energy conservation in an approximate way. These processes are included in the ten_layer_c3_canopy module along with photosynthesis.
- Light levels within the canopy strongly depend on environmental conditions outside the canopy, which are supplied via the weather data when running a simulation. The soybean model is designed to work with weather data sets where the incident sunlight is specified simply as the total intensity incident per unit of ground area just above the canopy. However, this is not enough information to determine light levels inside the canopy—it is critical to know how much of the light is direct vs. diffuse radiation, and for the direct light, it is critical to know the angle at which it hits the canopy. The solar_position_michalsky module calculates the angular position of the Sun given the latitude, longitude, and time, the shortwave_atmospheric_scattering module calculates the expected fractions of direct vs. diffuse light from the Sun's position, and the incident_shortwave_from_ground_par module calculates the actual intensities of direct and diffuse light just above the canopy.
- Once a canopy assimilation rate has been calculated, other modules decide how the assimilated carbon should be allocated for growth and maintenance respiration among the crop's tissues. Currently, BioCro crop models all use *partitioning* models, where fractions of the assimilated carbon are assigned to each tissue to be used for growth and respiration; this is accomplished by pairing the partitioning_growth module with one of the available "partitioning growth calculator" modules, which each implement slightly different rules for taking respiration and other factors into account. The soybean model uses the partitioning_growth_calculator module as its partitioning growth calculator.
- There are also several possible strategies for choosing the "partitioning coefficients," dimensionless fractions that determine the relative amounts of carbon devoted to each tissue in a partitioning growth model. In general, these coefficients depend on the developmental stage of the plant; as an extreme example, soybeans do not devote any carbon towards pod production until the flowering stage has been reached. The soybean model uses the partitioning_coefficient_logistic module to vary these coefficients continuously as the crop ages. The developmental stage of the crop is expressed as a dimensionless *development index* (DVI), which is determined from the temperature and photoperiod using the soybean_development_rate_calculator module. Most BioCro modules are intended to be applicable to multiple crops, but the developmental response to temperature and photoperiod is very specific to soybean.
- As a crop continues to age, it eventually begins to senesce. The soybean model uses the senescence_logistic module to accomplish this, which decreases the mass of each tissue at a rate proportional to its mass; in other words, it implements an exponential decay. Some of the lost mass is converted to litter, and some is remobilized for use by other organs such as the pod. The rate constants for senescence are calculated from soybean DVI by the senescence_coefficient_logistic module.
- Water availability plays a major role in crop growth. When water is scarce, plants respond by closing their stomata to reduce transpiration, which also reduces carbon assimilation. In the soybean model, this relationship is implemented by the stomata_water_stress_linear module, which determines a reduction factor for stomatal conductance from the soil water content. In turn, the soil water content is influenced by several factors: rainfall is the only source of soil water, while water available to the plant is lost due to canopy transpiration, evaporation from the soil surface, runoff, and the flow of water away from the surface to deeper soil layers. These processes are included in the soybean model via the soil_evaporation and two_layer_soil_profile modules.

- The `parameter_calculator` module is used to determine the canopy's leaf area index from the total leaf mass.
- The `thermal_time_linear` module is included for historical reasons and its output is not actually used by any other component of the model; it could be removed without any effect on the simulated values of biomass.

6 More analysis techniques

6.1 Optimization problems

We've now seen the essential BioCro functions, but have only just scratched the surface of what we can do with them.

R offers many possibilities in addition to the plotting tools that we have already used.

For example, we can solve optimization problems with functions like `optim`.

- The `optim` function is a general-purpose optimizer that minimizes the return value of another function (`fn`).
- Most optimizers, including `optim`, require that `fn` takes a numeric vector as an input and returns a single numeric value as its output.

We could use this to maximize an output like assimilation or to fit simulated biomass to measured data.

But, first we need to create a suitable `fn` from the other BioCro functions like `evaluate_module` and `run_biocro`. These functions have too many inputs and outputs, so they need to be modified.

As a general strategy, we can take the following two steps to create a suitable `fn`:

- Identify inputs that we want to vary, and fix the values of all other inputs; this technique is known in computer programming as *partial application*.
- Choose a method for extracting one important value to be optimized; this step is more open-ended and will depend on the particular situation.

The BioCro package provides functions to help with partial application (`partial_evaluate_module` and `partial_run_biocro`), and the following examples will demonstrate how they can be used in the context of optimization.

6.2 Maximizing soybean assimilation

Let's say we want to maximize soybean assimilation rate by varying the leaf temperature; in other words, we want to find the temperature that produces the highest soybean assimilation rate for a set of otherwise fixed conditions.

In this case, we could use `optim` to maximize the value of a function `fn` that takes a value of leaf temperature as an input and returns a value of the net assimilation rate.

Here we show how to generate a suitable function from the `BioCro:c3_assimilation` module. The first step is to indentify leaf temperature as an independent input and fix the other module input values.

Code to run in R

```
c3_tleaf <- partial_evaluate_module(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    Qabs = 1800     # absorbed PPFD in micromol / m^2 / s
    StomataWS = 1   # 1 indicates no water stress, 0 indicates maximum water stress
    temp = 25       # air temperature in degrees C
    rh = 0.7        # relative humidity (dimensionless)
    gbw = 2.0       # boundary layer conductance to H2O in mol / m^2 / s
  }),
  'Tleaf'           # specify the inputs that should not be fixed
)

# We can run the module using a leaf temperature of 27 degrees C
str(c3_tleaf(27))
```

Expected output from R

```
## List of 2
##  $ inputs :List of 41
##   ..$ Catm                          : num 373
##   ..$ Gs_min                        : num 0.001
##   ..$ Gstar_Ea                      : num 37830
##   ..$ Gstar_c                       : num 19
##   ..$ Jmax_Ea                       : num 43540
##   ..$ Jmax_at_25                    : num 195
##   ..$ Jmax_c                        : num 17.6
##   ..$ Kc_Ea                         : num 79430
##   ..$ Kc_c                          : num 38
##   ..$ Ko_Ea                         : num 36380
##   ..$ Ko_c                          : num 20.3
##   ..$ O2                            : num 210
##   ..$ Qabs                          : num 1800
##   ..$ RL_Ea                         : num 46390
##   ..$ RL_at_25                      : num 1.28
##   ..$ RL_c                          : num 18.7
##   ..$ StomataWS                     : num 1
##   ..$ Tleaf                         : num 27
##   ..$ Tp_Ha                         : num 62990
##   ..$ Tp_Hd                         : num 182140
##   ..$ Tp_S                          : num 588
##   ..$ Tp_at_25                      : num 13
##   ..$ Tp_c                          : num 19.8
##   ..$ Vcmax_Ea                      : num 65330
##   ..$ Vcmax_at_25                   : num 110
##   ..$ Vcmax_c                       : num 26.4
##   ..$ atmospheric_pressure          : num 101325
##   ..$ b0                            : num 0.008
##   ..$ b1                            : num 10.6
##   ..$ beta_PSII                     : num 0.5
##   ..$ electrons_per_carboxylation: num 4.5
##   ..$ electrons_per_oxygenation  : num 5.25
##   ..$ gbw                           : num 2
##   ..$ phi_PSII_0                    : num 0.352
##   ..$ phi_PSII_1                    : num 0.022
##   ..$ phi_PSII_2                    : num -0.00034
##   ..$ rh                            : num 0.7
##   ..$ temp                          : num 25
##   ..$ theta_0                       : num 0.76
##   ..$ theta_1                       : num 0.018
##   ..$ theta_2                       : num -0.00037
##  $ outputs:List of 11
##   ..$ Assim            : num 25.7
##   ..$ Assim_check      : num -6.08e-13
##   ..$ Assim_conductance: num 103
##   ..$ Ci               : num 280
##   ..$ Cs               : num 355
##   ..$ GrossAssim       : num 32.7
##   ..$ Gs               : num 0.547
##   ..$ iterations       : num 8
##   ..$ RHs              : num 0.703
##   ..$ RL               : num 1.46
##   ..$ Rp               : num 5.55
```

The new `c3_tleaf` function accepts the required input, but it returns much more information than we need. Here we create a simple "wrapper" for this function that extracts only the net assimilation output.

Code to run in R

```
c3_assim_from_tleaf <- function(x) {c3_tleaf(x)$outputs$Assim}

# We can run the module using a leaf temperature of 27 degrees C, only returning
# the Assim output
c3_assim_from_tleaf(27)
```

Expected output from R

```
## [1] 25.66557
```

Now we can pass this function to `optim`, with one small modification: `optim` actually *minimizes* the value of the function passed to it, so if we want to *maximize* net assimilation, we need to rephrase the problem as minimizing the negative of the value of net assimilation.

Code to run in R

```
optim_result <- optim(
  25,                                    # initial guess for optimal temperature
  function(x) {-c3_assim_from_tleaf(x)}, # rephrase to a minimization
  method = 'Brent',                      # works well for 1D optimization
  lower = 0,                             # lower limit of temperatures to try
  upper = 50                             # upper limit of temperatures to try
)

# Print the optimal Tleaf value
optim_result$par
```

Expected output from R

```
## [1] 27.42779
```

For a more advanced example where net assimilation is maximized under multiple conditions to calculate an optimal temperature curve, see Section 8.3.

6.3 Parameterizing a model using measured data

One of the most important parts of developing a model is parameterizing it by comparing its predictions against measured data.

An optimizer can be used to accomplish this goal by finding the optimal values of parameters that produce the best agreement with the measurements.

In order to do this, the main tasks are to:

- Obtain measurements of quantities that can be predicted by the model.
- Identify which of the model parameters should be varied to create the best match with the measured data.

- Define an *error metric* that quantifies the degree of agreement between the predictions and the observations; typically this will be something like the χ^2 statistic. Smaller values of the error metric should indicate a closer agreement.
- Create a function that accepts a single vector (the values of the selected parameters) and returns a single number (the resulting value of the error metric).

Once these tasks have been performed, the resulting function can then be passed to an optimizer, which will find the values that minimize the error.

Here we show how this parameterization can be accomplished using BioCro. In this example:

- We have a basic model for Miscanthus growth.
- We have Miscanthus leaf area index (LAI) observed at multiple times during the season and above-ground dry yield at the end of the season.
- Many parameters affect both of these values, but we will fit only the initial specific leaf area (`iSp`) and the "specific leaf area thermal time decay" (`Sp_thermal_time_decay`), which reduces the specific leaf area as the crop ages.
- We will use χ^2 for the error metric; this is a useful statistic when fitting observations of quantities that may have very different magnitudes due to differences in units.
- We will use one of the derivative-free optimization algorithms available from the `dfoptim` package, since it tends to converge quickly in these applications.

The basic Miscanthus model is included with the BioCro R package, as is the weather data. The observed values of LAI and above-ground dry yield will be defined below using R commands. In a more realistic situation, the model definition, weather data, and observations may be stored in text or CSV files and subsequently loaded into the R workspace.

Code to run in R

```
# Here we create a data frame with the Miscanthus observations.
# - time: The time of the observation following BioCro conventions, expressed as
#   the number of hours since the start of the year
# - lai: The leaf area index (dimensionless)
# - above_ground_dry_yield: Total stem and leaf mass in Mg / ha
yield <- utils::read.table(
  textConnection('
    time  lai     above_ground_dry_yield
    4224  0.0445  NA
    4296  0.0035  NA
    4416  0.0104  NA
    4536  0.0568  NA
    4608  0.0878  NA
    4704  0.2725  NA
    4824  2.1064  NA
    4896  2.7614  NA
    4992  3.899   NA
    5064  3.4318  NA
    5160  4.4921  NA
    5232  4.8522  NA
    5304  5.1393  NA
    5400  4.9133  NA
    5472  4.7485  NA
    6120  NA      19.35101369
  '),
  header = TRUE
)

# View the result
str(yield)

# Now we create a corresponding weather data set
min_doy <- min(floor(yield$time / 24)) - 1
max_doy <- max(ceiling(yield$time / 24)) + 1

miscanthus_weather <- weather[['2016']]

miscanthus_weather <-
  with(miscanthus_weather, {miscanthus_weather[doy >= min_doy & doy <= max_doy, ]})
```

Expected output from R

```
## 'data.frame': 16 obs. of  3 variables:
##  $ time                  : int  4224 4296 4416 4536 4608 4704 4824 4896 4992 5064 ...
##  $ lai                   : num  0.0445 0.0035 0.0104 0.0568 0.0878 ...
##  $ above_ground_dry_yield: num  NA NA NA NA NA NA NA NA NA NA ...
```

Next, we'll create a function that runs a miscanthus growth simulation using the weather data but accepts only `iSp` and `Sp_thermal_time_decay` as inputs; this can be accomplished using `partial_run_biocro`. When comparing to the measured data, we'll need the above-ground dry yield; this isn't included in the typical simulation

output, but can be calculated afterwards.

Code to run in R

```
par_to_optimize <- c('iSp', 'Sp_thermal_time_decay')

partial_function <- function(x) {
  f = with(miscanthus_x_giganteus, {partial_run_biocro(
    initial_values,
    parameters,
    miscanthus_weather,
    direct_modules,
    differential_modules,
    ode_solver,
    par_to_optimize
  )})
  r = f(x)
  r = within(r, {above_ground_dry_yield = Leaf + Stem})
  r
}
```

Expected output from R

There shouldn't be any output from this code.

Now `partial_function` accepts a vector of the two values, but it still returns all of the model output rather than a single value to minimize. We can use it to calculate χ^2 by wrapping another function around it.

Code to run in R

```
to_minimize <- function(x) {
  # Get results from the model and then pull out times at which there are
  # observed data. Use tryCatch to ensure that bad parameters do not completely
  # prevent the optimization from running.
  r <- tryCatch(
    {partial_function(x)},
    error = function(e) {data.frame()}
  )
  r_at_obs <- r[r$time %in% yield$time, ]

  # We'll used the chi-squared statistic. It is the sum of squared differences
  # divided by predicted values. Using this gives more equal weight to each
  # variable you're predicting. Otherwise, values that are big because of unit
  # differences will have more weight.
  obs_lai <- yield$lai
  pred_lai <- r_at_obs$lai
  ss1 <- sum(((obs_lai - pred_lai)^2 / pred_lai), na.rm = TRUE)

  obs_yield <- yield$above_ground_dry_yield
  pred_yield <- r_at_obs$above_ground_dry_yield
  ss2 <- sum(((obs_yield - pred_yield)^2 / pred_yield), na.rm = TRUE)
  ss <- ss1 + ss2

  # Some parameters cause the model to fail completely. Add a large penalty in
  # these cases. This will make the optimizer move away from unrealistic input
  # values. Here, `!is.finite` will find Inf, NaN, and NA values.
  if (nrow(r) < 1 || any(!is.finite(pred_lai)) || any(!is.finite(pred_yield))) {
    ss <- ss + 1000
  }
  ss
}

# Test the function with default values
print(to_minimize(unlist(miscanthus_x_giganteus$parameters[par_to_optimize])))
```

Expected output from R

```
## [1] 9.246989
```

Now we can perform the optimization; we'll also wrap the call to the optimizer in `system.time` so we can see how long it takes to run.

Code to run in R

```
system.time({
  miscanthus_optim_result <- nmkb(
    par = unlist(miscanthus_x_giganteus$parameters[par_to_optimize]),
    fn = to_minimize,
    lower = c(1e-3, -0.1),
    upper = c(10, 0.1)
  )
})

# See the original and optimized parameter values
cat('Original values:\n')
print(miscanthus_x_giganteus$parameters[par_to_optimize])
cat('Optimized values:\n')
print(setNames(miscanthus_optim_result$par, par_to_optimize))
```

Expected output from R

```
##    user  system elapsed
##  22.663   0.012  22.683
## Original values:
## $iSp
## [1] 1.7
##
## $Sp_thermal_time_decay
## [1] 0
## Optimized values:
##                   iSp Sp_thermal_time_decay
##           0.008709778          -0.002516946
```

Now we can check the result. In these plots, simulated values are shown as lines and observed values are shown as solid circles.

Code to run in R

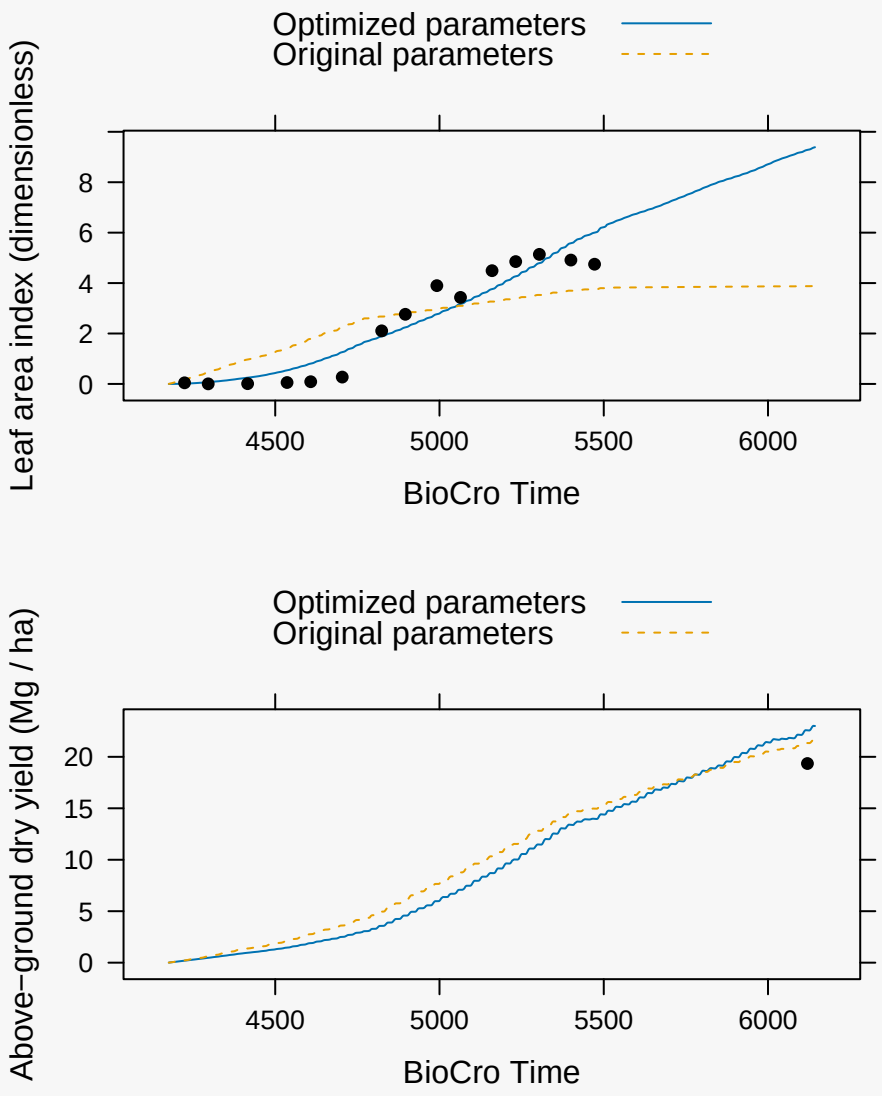
```
original_res <- partial_function(miscanthus_x_giganteus$parameters[par_to_optimize])
optimized_res <- partial_function(miscanthus_optim_result$par)

full_res <- rbind(
  within(original_res, {par_type = 'Original parameters'}),
  within(optimized_res, {par_type = 'Optimized parameters'})
)

xyplot(
  lai ~ time,
  group = par_type,
  data = full_res,
  type = 'l',
  xlab = 'BioCro Time',
  ylab = 'Leaf area index (dimensionless)',
  par.settings = list(superpose.line = list(lty = c(1, 2))),
  auto.key = list(space = 'top'),
  panel = function(...) {
    panel.xyplot(...)
    panel.points(yield$lai ~ yield$time, pch = 16, col = 'black')
  }
)

xyplot(
  above_ground_dry_yield ~ time,
  group = par_type,
  data = full_res,
  type = 'l',
  xlab = 'BioCro Time',
  ylab = 'Above-ground dry yield (Mg / ha)',
  par.settings = list(superpose.line = list(lty = c(1, 2))),
  auto.key = list(space = 'top'),
  panel = function(...) {
    panel.xyplot(...)
    panel.points(yield$above_ground_dry_yield ~ yield$time, pch = 16, col = 'black')
  }
)
```

Expected output from R



This optimization may seem complicated, but it is actually much simpler than a realistic model parameterization, which may include many more observations and parameters. A more reliable and useful objective function would also include more checks for error conditions.

A full example demonstrating one approach to parameterizing BioCro's soybean model can be found online at the documentation web site for the `BioCroValidation` R package: https://biocro.org/BioCroValidation/articles/parameterizing_soybean_biocro.html.

6.4 What comes next

That's the end of the structured part of the workshop! With any remaining time that might be left, you can:

- Participate in the soybean challenge (Section 7).

- Try out some of the additional examples (Section 8).
- Try working on your own modeling ideas.
- Ask any other questions you might have.
- Learn more about BioCro's approach to modular modeling (Section 9).
- Read one of the articles on the BioCro web site.

7 The soybean challenge

One way to test your BioCro knowledge is to try to maximize soybean yield. As in Section 5.1, we can run a soybean simulation for 2002.

Code to run in R

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

Expected output from R

There shouldn't be any output from this code.

Then, the final pod biomass can be extracted.

Code to run in R

```
soybean_result$Grain[nrow(soybean_result)]
```

Expected output from R

```
## [1] 5.454286
```

Exercise

Your task is to increase that value!

For example, you could try increasing a photosynthesis parameter like $V_{c,max}$, which is represented in the soybean model as `Vcmax_at_25`.

Code to run in R

```
soybean_result_new <- run_biocro(
  soybean$initial_values,
  within(soybean$parameters, {Vcmax_at_25 = 200}),
  soybean_weather$'2002',
  soybean$direct_modules,
  soybean$differential_modules,
  soybean$ode_solver
)

soybean_result_new$Grain[nrow(soybean_result)]
```

Expected output from R

```
## [1] 5.512024
```

Any approach is allowed! If you think you have a way to get a super high yield and would like to share it, email your code to eloch@illinois.edu and jmcgrath@illinois.edu. Interesting approaches will be showcased on the BioCro website (with your permission).

8 Additional examples

8.1 Swapping a model component

In Section 5.1, we ran a default Soybean-BioCro simulation for weather data from 2002.

Code to run in R

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

Expected output from R

There shouldn't be any output from this code.

In this simulation, photosynthesis was included via the mechanistic Farquhar-von-Caemmerer-Berry (FvCB) model. Alternatively, it would be possible to use a simpler photosynthesis model based on radiation use efficiency (RUE). In the RUE model, the gross assimilation rate is directly proportional to the incident light intensity. The standard BioCro module library contains a RUE canopy photosynthesis module, so it is simple to run an alternate soybean simulation using RUE photosynthesis in place of the FvCB model. In addition to switching a module, we will also need to provide a value for the efficiency, which is called `alpha_rue`.

Code to run in R

```
soybean_result_rue <- run_biocro(
  soybean$initial_values,
  within(soybean$parameters, {alpha_rue = 0.05}),
  soybean_weather$'2002',
  within(soybean$direct_modules, {
    canopy_photosynthesis = 'BioCro:ten_layer_rue_canopy'
  }),
  soybean$differential_modules,
  soybean$ode_solver
)
```

Expected output from R

There shouldn't be any output from this code.

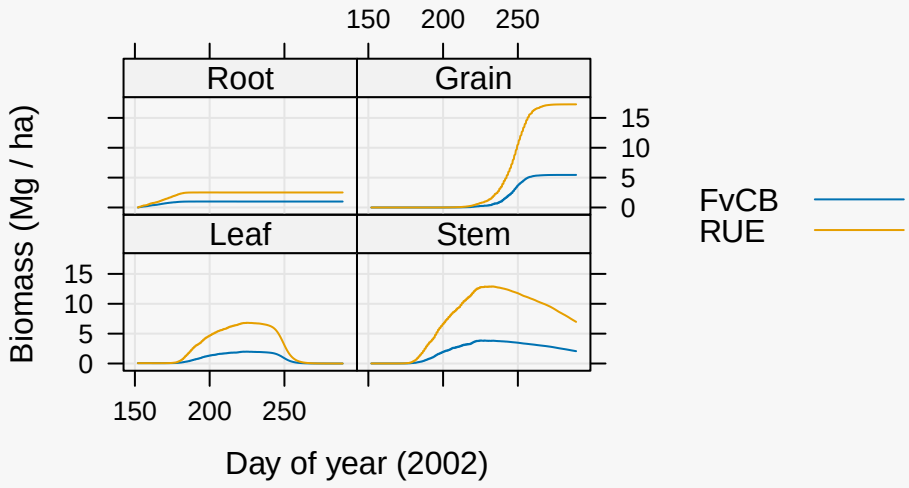
Predicted biomass values from the two versions can be compared visually:

Code to run in R

```
biomass_col <- c('fractional_doy', 'Leaf', 'Stem', 'Root', 'Grain')
soybean_comp <- rbind(
  within(soybean_result[biomass_col], {photo_method = 'FvCB'}),
  within(soybean_result_rue[biomass_col], {photo_method = 'RUE'})
)

xyplot(
  Leaf + Stem + Root + Grain ~ fractional_doy,
  group = photo_method,
  data = soybean_comp,
  type = 'l',
  auto.key = list(space = 'right'),
  grid = TRUE,
  xlab = 'Day of year (2002)',
  ylab = 'Biomass (Mg / ha)'
)
```

Expected output from R



With this value of `alpha_rue`, the agreement between the model predictions is not so great. However, an optimization procedure similar to the ones in Section 6.3 could be used to find the value of `alpha_rue` that produces the best agreement. Afterwards, the response of the two different soybean models to important factors like temperature and CO_2 can be compared using response curves (as in Section 4.4.2) and sensitivity analysis (as in Sections 8.4 and 8.4). This type of analysis can reveal scenarios where the model behavior is expected to differ. For an example of how this could be done, see Lochocki et al. 2022 and the *Quantitative Comparison Between Two Photosynthesis Models* article.

8.2 Simultaneously calculating and plotting multiple response curves

When calling `module_response_curve`, it is possible to vary multiple inputs at one time, which allows us to calculate multiple response curves with one command.

For example, we can calculate the temperature response of light-saturated net assimilation at several values of relative humidity (RH). Here, the leaf temperature and humidity values are independent of each other, so we use the `expand.grid` function to form a data frame of all possible combinations of their values. Then, we use the `group` option in `xyplot` to separately plot the light response curves for each value of relative humidity. We also use the `paste` function to generate an informative plot title from the information contained in the response curve data frame.

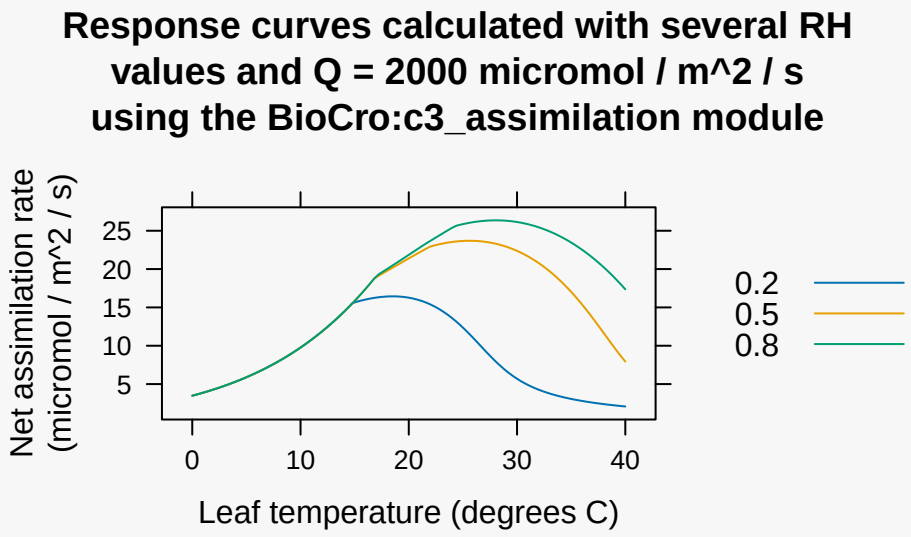
Code to run in R

```
temperature_response_curve <- module_response_curve(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {Qabs = 2000; StomataWS = 1; temp = 25; gbw = 2.0}),
  expand.grid(
    Tleaf = seq(from = 0, to = 40, length.out = 101),
    rh = c(0.2, 0.5, 0.8)
  )
)

caption <- paste(
  'Response curves calculated with several RH\nvalues and Q =',
  unique(temperature_response_curve$Qabs),
  'micromol / m^2 / s\nusing the',
  unique(temperature_response_curve$module_name),
  'module'
)

temperature_response_curve_plot <- xyplot(
  Assim ~ Tleaf,
  group = rh,
  data = temperature_response_curve,
  auto.key = list(space = 'right'),
  type = 'l',
  xlab = 'Leaf temperature (degrees C)',
  ylab = 'Net assimilation rate\n(micromol / m^2 / s)',
  main = caption
)
print(temperature_response_curve_plot)
```

Expected output from R



Exercise

Try changing the value of the absorbed photosynthetically active photon flux density `Qabs` and notice how the plot caption is automatically updated to reflect the new value.

8.3 Maximizing net assimilation under multiple conditions

An interesting feature of the response curves in Section 8.2 is that for each value of RH, there is an optimal value of leaf temperature that maximizes the net assimilation rate. We can use a combination of BioCro tools and built-in R tools to determine the optimal temperature for a given RH value. In this example, we use the `partial_evaluate_module` BioCro function to hold all module inputs except `Tleaf` and `RH` constant. Then we ask an optimizer to find the value of `Tleaf` that maximizes the module's net assimilation output for a sequence of humidities. Using this process, we generate a curve we can use to visualize how the optimal temperature for C_3 photosynthesis depends on ambient humidity, according to this model.

Code to run in R

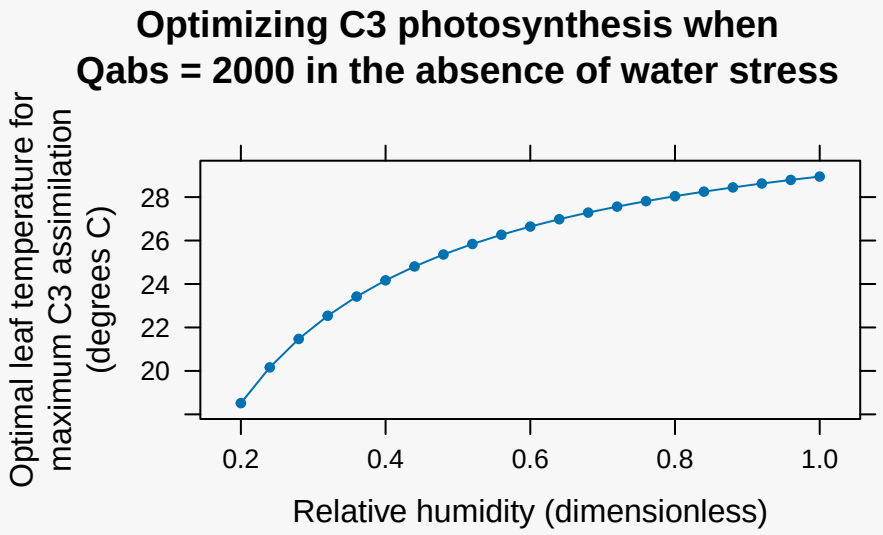
```
c3_tleaf_rh <- partial_evaluate_module(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {Qabs = 2000; StomataWS = 1; temp = 25; gbw = 2.0}),
  c('Tleaf', 'rh')
)

rh_seq <- seq(0.2, 1, length.out = 21)

optim_temp <- sapply(rh_seq, function(rh) {
  optim_result_rh <- optim(
    25,
    function(tl) {-c3_tleaf_rh(c(tl, rh))$outputs$Assim},
    method = 'Brent',
    lower = 0,
    upper = 50
  )
  optim_result_rh$par
})

optim_temp_plot <- xyplot(
  optim_temp ~ rh_seq,
  type = 'b',
  pch = 20,
  xlab = 'Relative humidity (dimensionless)',
  ylab = 'Optimal leaf temperature for\nmaximum C3 assimilation\n(degrees C)',
  main = 'Optimizing C3 photosynthesis when\nQabs = 2000 in the absence of water stress'
)
print(optim_temp_plot)
```

Expected output from R



It is well-known that the optimal temperature for photosynthesis depends on the ambient CO_2 concentration (Long 1991). This example shows that the optimal temperature also changes with humidity levels, shifting to lower temperatures in drier conditions.

Exercise

Try finding the optimal LAI that maximizes net soybean canopy assimilation for fixed environmental conditions. For this, the `BioCro:c3_canopy` module would be required instead of the `BioCro:c3_assimilation` module.

8.4 Calculating module sensitivity coefficients

Sensitivity analysis can be a powerful way to understand and demonstrate the behavior of a model component or a model as a whole. Essentially, we choose a particular output that we're interested in (such as net assimilation rate) and calculate changes in that variable that occur in response to small changes in an input. This technique can reveal which model inputs have the biggest impact on its important outputs; in turn, this knowledge can be used to guide future investigations.

There are several ways to calculate sensitivity coefficients during sensitivity analysis. Let's say we have a function $f(x)$ that calculates a value of y from a value of x, and we want to calculate the sensitivity of y to x at a value of x_0. The simplest option would be to choose a small x step size (δx) and compute the sensitivity as

$$s_1 = f(x_0 + \delta x) - f(x_0). \tag{1}$$

In other words, s_1 is the change in y caused by an increase in x from x_0 to $x_0 + \delta x$. A disadvantage of this approach is that s_1 strongly depends on the value of δx. So we could also try

$$s_2 = s_1/\delta x. \tag{2}$$

Now we can interpret s_2 as the change in y per change in x. Note that as δx approaches 0, s_2 approaches the derivative df/dx evaluated at x_0. In other words, for values of δx that are sufficiently small, s_2 is independent of δx. Another option is to calculate fractional changes rather than absolute changes, i.e.,

$$s_3 = \frac{s_1/f(x_0)}{\delta x/x_0} = \frac{s_2}{f(x_0)/x_0} = s_2\frac{x_0}{f(x_0)}. \tag{3}$$

We can interpret s_3 as the relative change in y per relative change in x. As δx approaches 0, s_3 approaches $(df/dx)(x_0/f(x_0))$, where the derivative is evaluated at x_0. Again, for values of δx that are sufficiently small, s_3 is independent of δx. Note that this approach is not viable if x_0 is zero because s_3 is undefined in that case. (If df/dx approaches infinity as x_0 approaches 0, then s_3 may have a finite value that could be calculated using L'Hôpital's rule; however, this situation is difficult to deal with using numerical methods.)

Also note that there are two strategies commonly employed for choosing δx. One is to simply specify an absolute value. The other is to specify a relative step size where $\delta x = \epsilon x_0$ for a small value of ϵ.

Here we define a sensitivity helping function that requires $f(x)$, x_0, and either δx or ϵ as input arguments, and returns the values of s_1, s_2, and s_3 as a data frame. If `delta_x` is `NULL`, then a relative step based on ϵ will be used for the calculations.

Code to run in R

```
sensitivity <- function(FUN, x0, ep = 1e-6, delta_x = NULL) {
  if (is.null(delta_x)) {
    delta_x <- x0 * ep
  }

  x1 <- x0 + delta_x
  y0 <- FUN(x0)
  y1 <- FUN(x1)
  s1 <- y1 - y0
  s2 <- s1 / delta_x
  s3 <- s2 / (y0 / x0)

  return(data.frame(
    x0 = x0,
    x1 = x1,
    y0 = y0,
    y1 = y1,
    s1 = s1,
    s2 = s2,
    s3 = s3
  ))
}
```

Expected output from R

There shouldn't be any output from this code.

Now, let's calculate sensitivity coefficients for the `c3_assimilation` module, treating the ambient CO_2 concentration C_a as the independent variable and the net assimilation as the dependent variable. We'll also provide an option to specify the absorbed light level. First, we'll need to create a function f such that $f(CO_2, Q)$ = net assimilation. We can specify the independent variables using `partial_evaluate_module`, and then create another function wrapper later that selects just the net assimilation output.

Code to run in R

```
co2_func <- partial_evaluate_module(
  'BioCro:c3_assimilation',
  within(soybean$parameters, {
    StomataWS = 1
    temp = 25
    Tleaf = 27
    rh = 0.7
    gbw = 2.0
  }),
  c('Catm', 'Qabs')
)
```

Expected output from R

There shouldn't be any output from this code.

Now we will define a range of absorbed light values to use and calculate the sensitivity coefficients for each one using a base level of C_a = 400 ppm.

Code to run in R

```
qabs_seq <- seq(0, 2000, length.out = 101)

sens_result <- do.call(rbind, lapply(qabs_seq, function(Qabs) {
  sens <- sensitivity(
    function(catm) {co2_func(c(catm, Qabs))$outputs$Assim},
    400
  )
  sens$Qabs <- Qabs  # Also include the absorbed light level in the result
  return(sens)
}))
```

Expected output from R

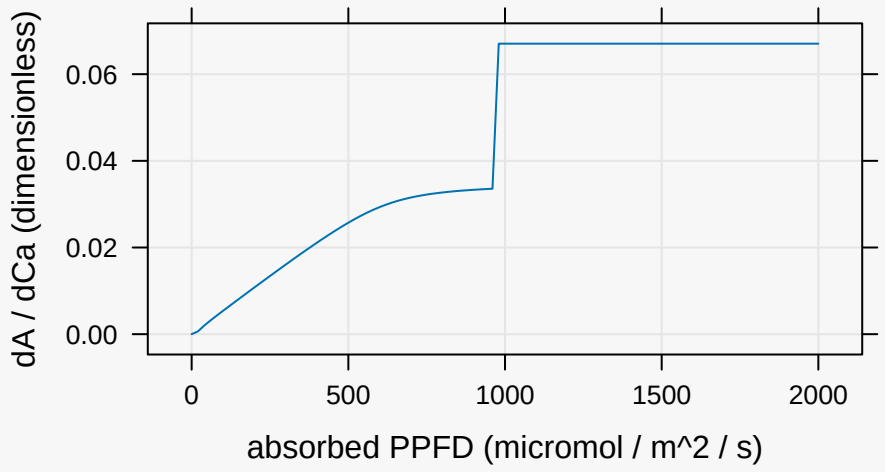
There shouldn't be any output from this code.

Now we can see the sensitivity of net assimilation to ambient CO_2 concentration at different levels of absorbed light intensity.

Code to run in R

```
sensitivity_plot <- xyplot(
  s2 ~ Qabs,
  data = sens_result,
  type = 'l',
  grid = TRUE,
  xlab = 'absorbed PPFD (micromol / m^2 / s)',
  ylab = 'dA / dCa (dimensionless)'
)
print(sensitivity_plot)
```

Expected output from R



From this plot, we can learn several things. One observation is that the sensitivity coefficient is non-negative for all values of absorbed light; this indicates additional CO_2 never causes a decrease in soybean assimilation. Another observation is that that soybean assimilation is most sensitive to increases in ambient CO_2 levels when the absorbed light levels are high. This agrees with our intuition; CO_2 assimilation is limited by light availability at low light levels, so additional CO_2 cannot be used for assimilation.

Exercise

Use the `BioCro:c4_assimilation` module to calculate a similar sensitivity curve for miscanthus (using default values from `miscanthus_x_giganteus_parameters`).

This will reveal some differences between the response of C_3 and C_4 photosynthesis to changes in atmospheric CO_2 levels.

8.5 Calculating simulation sensitivity coefficients

It is possible to calculate the sensitivity of simulation outputs like final pod biomass to input parameters like C_a following the same approach used in Section 8.4. The only difference would be that `partial_run_biocro` will be used to construct $f(x)$, rather than `partial_evaluate_module`.

As a simple example, we will calculate the sensitivity of final pod biomass in 2002 to the value of `alphaLeaf`. Here we will use partial application, and will extract just the final seed mass from a full simulation result, similar to the strategy used in Section 6.2. These techniques will create a function that can be passed to the sensitivity calculator defined in Section 8.4.

Code to run in R

```
# Use partial application to fix all inputs except alphaLeaf
aleaf_func <- with(soybean, {partial_run_biocro(
  initial_values,
  parameters,
  soybean_weather$'2002',
  direct_modules,
  differential_modules,
  ode_solver,
  'alphaLeaf'
)})

# Helper function that returns just the final seed mass
final_pod_func <- function(alphaLeaf) {
  sim_result <- aleaf_func(alphaLeaf)
  sim_result$Grain[nrow(sim_result)]
}

# Calculate sensitivity
sensitivity(final_pod_func, soybean$parameters$alphaLeaf)
```

Expected output from R

```
##         x0       x1       y0       y1            s1         s2        s3
## 1 23.36771 23.36774 5.454286 5.454281 -5.600647e-06 -0.2396746 -1.026834
```

An interesting observation here is that the sensitivity coefficients are negative, indicating that increases in `alphaLeaf` cause decreases in yield. `alphaLeaf` controls the crop's level of leaf investment, so we have learned that additional leaf investment is detrimental to soybean yield; this agrees with previous results indicating that soybean canopies are larger than necessary for optimal yields (Srinivasan, Kumar, and Long 2017).

Exercise

Try calculating the sensitivity coefficient for the response of soybean yield to changes in atmospheric CO_2 concentration.

See Lochocki et al. 2022 for a more detailed example of calculating and using simulation sensitivity coefficients.

9 Understanding BioCro's approach to modular modeling

9.1 Three levels of computational modeling

In general, there are several different ways to think about a computational model as we move from abstract to concrete representations. Defining a model using computer code—the most concrete representation—is necessary for numerically solving it, but there are several complications associated with modeling at the code level. BioCro is designed to help mitigate these issues. So, to better understand BioCro, it is helpful to understand the three essential ways to think about a model: at the conceptual level, at the equation level, and at the code level. In this section, we will illustrate these levels using a very simple plant growth model.

9.1.1 The concept level

At the conceptual level, we often represent models using diagrams like the one in Figure 1, where the model is shown as a collection of major components (boxes) and the connections between them (arrows). This particular diagram shows that the model can be divided into two conceptually distinct parts: leaf photosynthesis and carbon partitioning. The photosynthesis component uses the absorbed sunlight level and the leaf mass to determine the overall rate of mass gain due to carbon assimilation, and the partitioning component determines the separate rates of root and leaf mass gain from the overall rate. This type of diagram is essential when building a model because it allows us to identify the important processes, inputs, and outputs that we want to include in the model, as well as the relationships between them. However, it doesn't let us actually make quantitative calculations or predictions.

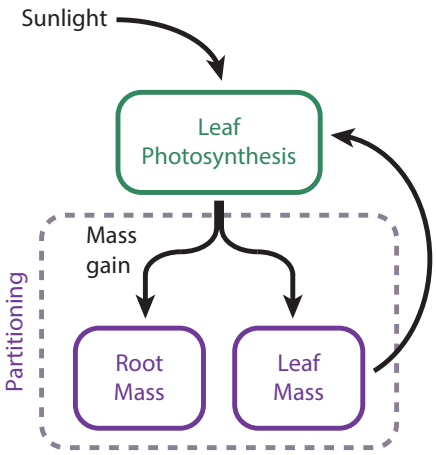


Figure 1: A conceptual flow diagram of a simple plant growth model where the absorbed sunlight and total leaf mass are used to determine the rate of carbon assimilation, and then the assimilated carbon is partitioned into leaf and root compartments where it determines the rates of mass gain. This is an informal style of diagram, but more specialized ones called *Forrester diagrams* more clearly specify different types of model components.

9.1.2 The equation level

In order to make calculations, we first need to associate equations with the boxes and lines in the conceptual diagram. Typically there are multiple possibilities for the equations to use; our choices will depend on the information we have available to us, as well as our goals for how we want to use the model. Here we will use some simple equations to represent the model.

For photosynthesis, we will calculate the rate of carbon assimilation per unit leaf area (A; mol m^{-2} s^{-1}) and the overall rate of mass gain (G; kg s^{-1}) from the absorbed photosynthetically-active photon flux density (Q; mol

m^{-2} s^{-1}) as follows:

$$A = Q \cdot \alpha_{RUE} \tag{4a}$$

$$G = A \cdot M_{leaf} \cdot SLA \cdot C_{conversion} \tag{4b}$$

Here, α_{RUE} (mol CO_2 (mol photon)$^{-1}$) is the radiation use efficiency, M_{leaf} is the leaf mass (kg), SLA (m^2 kg^{-1}) is the specific leaf area, and $C_{conversion}$ (kg mol^{-1}) is the amount of mass gained by the plant per mol of assimilated carbon.

For carbon partitioning, we will calculate the rates of leaf and root mass gain (dM_{leaf}/dt and dM_{root}/dt, respectively, where M_{root} is the root mass) as follows:

$$\frac{dM_{leaf}}{dt} = G \cdot f_{leaf} \tag{5a}$$

$$\frac{dM_{root}}{dt} = G \cdot f_{root} \tag{5b}$$

Here f_{leaf} and f_{root} are the fractions of carbon allocated to the leaf and root, respectively. (To ensure that mass is conserved, the sum of f_{leaf} and f_{root} should be 1.)

These equations represent the essential processes included in the model, but they don't specify values for the parameters (like SLA and α_{RUE}) or the initial values of the leaf and root masses. So, there are a few additional equations required to fully describe the model. We can define the parameters and initial values using simple equations as follows:

$$\alpha_{RUE} = 0.07 \tag{6a}$$

$$SLA = 25 \tag{6b}$$

$$f_{leaf} = 0.2 \tag{6c}$$

$$f_{root} = 0.8 \tag{6d}$$

$$C_{conversion} = 0.03 \tag{6e}$$

$$M_{leaf}(0) = 1 \tag{7a}$$

$$M_{root}(0) = 1 \tag{7b}$$

We would like to consider the absorbed sunlight Q as varying with time, so we have specified its value at 1-second intervals throughout the course of one day (Table 1). The values follow a half-period sinusoidal light profile throughout the day; this is not realistic, but it is sufficient for this simple model. They are calculated according to the following equation, where the time t is expressed in seconds:

$$Q(t) = \sin\left(\frac{t}{3600 \cdot 12} \cdot \pi\right) \cdot 2000 \times 10^{-6}. \tag{8}$$

Comparing Equations 4 and 5 with Figure 1, we can recognize the essential features of the conceptual diagram within the equations. For example, the absorbed sunlight Q and the leaf mass M_{leaf} are both inputs to the photosynthesis equations (Equation 4), while the total rate of mass gain (G) is an input to the partitioning equations (Equation 5). So, taken together, Equations 4, 5, 6, and 7, along with Table 1, define the model in a more concrete way than the conceptual diagram. However, we are still not able to readily solve the model or use it to make predictions.

t	Q
0	0
1	7.2×10^{-8}
2	1.5×10^{-7}
⋮	⋮
86400	0

Table 1: Values of $Q(t)$ specified at 1-second intervals over the course of one day, as calculated using Equation 8.

9.1.3 The code level

To finally make calculations, we need to translate these equations into a format that can be understood by a computer. We can think of this as the most concrete and specialized way to represent a model: the code level. For example, we can define and solve the model in the `R` environment with the help of the `deSolve` library (Figure 2).

```
f <- function(t, y, p) {
  with(p, with(as.list(y), {
    A = Q(t) * alpha_rue           # mol / m^2 / s
    G = A * Leaf * SLA * C_conv  # kg
    dLeaf = G * f_leaf           # kg / s
    dRoot = G * f_root           # kg / s
    return(list(c(dLeaf, dRoot)))
  }))
}

initial_values <- c(
  Leaf = 1,  # kg
  Root = 1   # kg
)

params <- list(
  alpha_rue = 0.07,      # mol / mol
  SLA = 25,              # m^2 / kg
  f_leaf = 0.2,          # kg / ks
  f_root = 0.8,          # kg / kg
  C_conv = 0.03,         # kg / mol
  Q = function(time) {  # mol / m^2 / s
    sin(time / 3600 / 12 * pi * 2000e-6)
  }
)

time <- seq(0, 3600 * 12)  # s
solution <- lsodes(initial_values, time, f, params)
```

Figure 2: `R` code capable of solving the model defined in Sections 9.1.1 and 9.1.2.

It is clear that all the equations have made an appearance in this code, but there are also several associated commands related to computer operations. The requirement to add these commands can be a barrier to using a computational model, since writing them requires specialized knowledge beyond just the mathematical form of the equations.

Another important thing to notice is that in this code, we have lost most of the flexibility that existed at the conceptual level. For example, at the conceptual level (Figure 1), we specify that there is a photosynthesis component, but we don't specify which one, and we are free to mentally replace one model with another. In the code, however, we have clearly specified that our photosynthesis model has a linear response to light, and it is much more difficult to switch to a different model. (Perhaps in this case it wouldn't be too much work, but more realistic models are typically much more complicated, and the existing code is not always organized in a way that clearly indicates which lines are related to which model component.)

9.2 Making computer code more modular and flexible

The main purpose of BioCro is to mitigate both issues with the code representation of a model that were discussed in Section 9.1.3— the requirement for specialized computer programming knowledge and the lack of flexibility— by (1) reducing the amount of "extra stuff" that must be written when translating equations to the code level and (2) retaining the flexibility of the concept level. In BioCro, this is accomplished by breaking up a model's equations into reusable groups that we call *modules*; this terminology was chosen to emphasize that BioCro takes a *modular* approach to modeling. (In typical computer science terms, one might say that BioCro takes a highly *encapsulated* approach to modeling.) To understand what this means, let's look at another short example showing how a typical piece of computer code can be made more modular. This time, we will consider two hypothetical pieces of code that represent models of soybean and miscanthus growth (Figure 3).

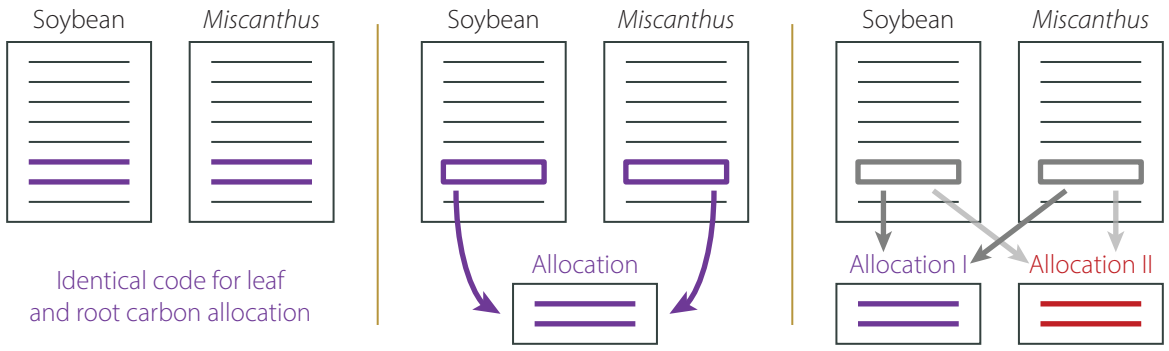


Figure 3: Diagram showing how models of soybean and miscanthus growth can be made more modular. *Left*: In the original versions, both pieces of code contain identical sections for modeling carbon allocation. *Center*: To reduce duplication, the code has been modified so that both crop models now use a separate function to implement the carbon allocation component. *Right*: To make the code even more flexible, the crop models now allow the user to specify a carbon allocation function, which could be selected from multiple options.

9.2.1 Identifying distinct concepts

The key to making code more modular and flexible is to identify and separate logically distinct concepts whenever possible. For example, let's say we notice that the two growth models are using the same equations for allocating carbon to the leaf and stem—perhaps both models use Equation 5 but with different parameters (see the left part of Figure 3). Noticing overlaps like this one are not always easy because the code may not be written in exactly the same way. For example, the `R` code for soybean might look like this:

```
leaf_mass_gain_rate <- total_mass_gain_rate * 0.2
root_mass_gain_rate <- total_mass_gain_rate * 0.8
```

while the `R` code used in the miscanthus version might look like this:

```
leaf_factor <- 0.3
dleaf_dt <- mass_gain * leaf_factor
droot_dt <- mass_gain * (1 - leaf_factor)
```

Thus, it can be time-consuming and difficult to identify cases of overlap.

9.2.2 Separating disctinct components

Nevertheless, once an area of overlap has been detected, we can now recognize that this carbon allocation concept is logically distinct from the rest of the code and could be separated out. One way to do this would be to write a dedicated function for carbon allocation, and have each crop model use this function (see the center part of Figure 3). For example, that function could look something like this:

```
# This function partitions the crop's total rate of mass gain into its leaf and
# root compartments.
leaf_root_partitioning <- function(
  total_mass_gain_rate, # kg / s
  leaf_factor           # dimensionless
)
{
  return(list(
    leaf_mass_gain_rate = total_mass_gain_rate * leaf_factor,       # kg / s
    root_mass_gain_rate = total_mass_gain_rate * (1 - leaf_factor) # kg / s
  ))
}
```

There are two main advantages to this approach. One is that if the allocation model needs to be updated (for example, to add another component such as the stem), it only needs to be modified in one place. The other is that the model is now easier to understand because the allocation component can easily be examined on its own.

9.2.3 Becoming even more flexible

This level of modularity is common in modeling, but it is possible to go further by recognizing that the carbon allocation code used in these models is just one of many possible ways to model carbon allocation. In a more flexible design, the code for the crop growth models could have an input argument that specifies the particular allocation function to use. Then a modeler using the code would be free to choose between multiple options if they are available (see the right part of Figure 3).

By repeating this process, it would be possible to develop a very flexible type of crop growth model, where a user is able to specify which photosynthesis equations to use, which carbon allocation equations to use, or any other model components that might have existed in the original versions of the models. But why stop there? We could also separate out the parameters, initial values, and drivers from the code that's used to combine the equations and run the model. We could even stop requiring specific model components (e.g. photosynthesis or carbon allocation) and allow the user to include any equations they would like to use. The end result is an extremely flexible framework for defining and running models of nearly any type. It could be used for simulating soybean or miscanthus growth, or even something completely different like a gene network.

It's important to realize that although there are numerous benefits to writing modular code this way, it also makes other aspects of the code more difficult. For example, if the user can specify *any* allocation model, it's now unclear which parameters would need to be specified by the user, since different models might require different parameters. Thus, the parameter specification needs to be very flexible, and the allocation model must be able to communicate which inputs it requires. These kinds of issues prevent code like this from being written very often.

9.3 The BioCro framework

In Section 9.2.3, we discussed the benefits of using fully modular code, but recognized out that writing it can be difficult. However, a key feature of modular and flexible code is that it only needs to be written once, and then it can be used for many different modeling purposes. This is exactly what has been done with BioCro—beginning

with several crop models that had been written separately, all distinct components were identified and separated, greatly reducing duplicated code and resulting in an extremely flexible modeling framework.

In BioCro, a user can choose which model components they wish to use and supply any other required specifications. Then, the BioCro framework takes care of combining the equations and running the simulation (Section 2.5). The framework is now quite complex, but most users will never need to look at its code directly.

Returning to the simple model we discussed before, we can see how it looks in BioCro (Figure 4) and compare this version to the earlier one (Figure 2). The most important thing to notice is that instead of directly specifying the equations for photosynthesis and carbon partitioning, we can now refer to pre-defined sets of equations by their names - in this case, `BioCro:example_model_mass_gain` and `BioCro:example_model_partitioning`. For this simple model, BioCro doesn't actually save a lot of typing, but for more realistic models with hundreds of equations, it is vastly faster and easier to define the model using BioCro modules instead of doing it manually. (Of course, this requires that the modules exist; writing them requires a bit of extra work, but the effort pays off because they can be reused for many purposes.)

```
time <- seq(0, 3600 * 12)                    # s
Q <- sin(time / 3600 / 12 * pi * 2000e-6)    # mol / m^2 / s

solution_biocro <- run_biocro(
    parameters = list(
        alpha_rue = 0.07,   # mol / mol
        SLA = 25,           # m^2 / kg
        f_leaf = 0.2,       # kg / ks
        f_root = 0.8,       # kg / kg
        C_conv = 0.03,      # kg / mol
        timestep = 1
    ),
    initial_values = list(
        Leaf = 1,   # kg
        Root = 1    # kg
    ),
    drivers = data.frame(time = time, Q = Q),
    direct_module_names = 'BioCro:example_model_mass_gain',
    differential_module_names = 'BioCro:example_model_partitioning'
)
```

Figure 4: `R` code using the BioCro package to solve the model defined in Sections 9.1.1 and 9.1.2.

Besides shortening the code, BioCro allows us to think of the model at the conceptual level—where model components are combined to form a larger model— even though we are working at the coding level required to actually run a simulation. This ability provides several important benefits to BioCro users, such as the following:

- Defining a model is completely separate from solving it, allowing users to focus on biology instead of computer science.
- Model components can easily be swapped for alternative versions to better match the available experimental inputs, to take advantage of new developments, or to compare alternative components.
- BioCro modules can be developed privately so users can withhold access to model components until after publication.

References

Campbell, Gaylon S. and John Norman (1998). *An Introduction to Environmental Biophysics*. 2nd ed. New York: Springer-Verlag (cit. on p. 18).

Humphries, S. W. and S. P. Long (1995). "WIMOVAC: a software package for modelling the dynamics of plant leaf and canopy photosynthesis". In: *Bioinformatics* 11.4, pp. 361–371. DOI: 10.1093/bioinformatics/11.4.361 (cit. on p. 5).

Lochocki, Edward B and Justin M McGrath (2025). "Widely Used Variants of the Farquhar-von-Caemmerer-Berry Model Can Cause Errors in Parameter Estimation". In: *in silico Plants*, diaf014. DOI: 10.1093/insilicoplants/diaf014 (cit. on p. 8).

Lochocki, Edward B et al. (2022). "BioCro II: a Software Package for Modular Crop Growth Simulations". In: *in silico Plants*. DOI: 10.1093/insilicoplants/diac003 (cit. on pp. 4, 5, 8, 36, 44).

Long, S. P. (1991). "Modification of the response of photosynthetic productivity to rising temperature by atmospheric CO2 concentrations: Has its importance been underestimated?" In: *Plant, Cell & Environment* 14.8, pp. 729–739. DOI: 10.1111/j.1365-3040.1991.tb01439.x (cit. on p. 40).

Matthews, Megan L et al. (2022). "Soybean-BioCro: a semi-mechanistic model of soybean growth". In: *in silico Plants* 4.1. DOI: 10.1093/insilicoplants/diab032 (cit. on pp. 6, 18).

Miguez, Fernando E. et al. (2012). "Modeling spatial and dynamic variation in growth, yield, and yield stability of the bioenergy crops Miscanthus × giganteus and Panicum virgatum across the conterminous United States". In: *GCB Bioenergy* 4.5, pp. 509–520. DOI: 10.1111/j.1757-1707.2011.01150.x (cit. on p. 5).

Monteith, John and Mike Unsworth (2013). *Principles of environmental physics: plants, animals, and the atmosphere*. Academic Press (cit. on p. 18).

Srinivasan, Venkatraman, Praveen Kumar, and Stephen P. Long (2017). "Decreasing, not increasing, leaf area will raise crop yields under global atmospheric change". In: *Global Change Biology* 23.4, pp. 1626–1635. DOI: 10.1111/gcb.13526 (cit. on p. 44).

Thornley, John H. M. and I. R. Johnson (1990). *Plant and Crop Modelling: A Mathematical Approach to Plant and Crop Physiology* (cit. on p. 18).

von Caemmerer, S. (2000). *Biochemical Models of Leaf Photosynthesis*. CSIRO Publishing. DOI: 10.1071/9780643103405 (cit. on p. 8).