Draft version January 3, 2019
Preprint typeset using LaTeX style AASTeX6 v. 1.0

cis_interface: A PYTHON PACKAGE FOR INTEGRATING COMPUTATIONAL MODELS ACROSS LANGUAGES AND SCALES

Meagan Lang[1]

[1]National Center for Supercomputing Application, University of Illinois, Urbana-Champaign, IL email: langmm@illinois.edu

ABSTRACT

Thousands of computational models have been created within the plant biology community in the past two decades that have the potential to be combined into complex integration networks capable of capturing more complex biological processes than possible with isolated models. However, the technological barriers introduced by differences in language and data formats has slowed this progress. We present cis_interface, a Python package for running integration networks with connections between models across languages and scales. cis_interface coordinates parallel execution of models in Python, C, C++, and Matlab on Linux, Mac OS, and Windows operating systems, and handles communication in a number of data formats common to computational plant modeling. cis_interface is designed to be user-friendly and can be accessed at https://github.com/cropsinsilico/cis_interface.

1. INTRODUCTION

Plant biologists have produced a wealth of computational models to describe the biological processes governing plant growth and development, covering scales from atomistic to global. Although usually developed independently to address questions specific to an organ, species, or growth process, computational plant models can also have applications beyond their original scope. Many biological processes are the same across different species of plants, allowing models developed for one species to be adapted for another by modifying input parameters. In addition, computational models for different scales, organs, or processes are often related via biological dependencies. With advances in computational power, it should be possible to link biologically-related models to create complex integration networks capable of capturing the response of entire plants, fields, or regions. However, independently developed computational models often have compatibility issues resulting from differences in programming language or data format that make such collaborations difficult.

Consider the task of integrating a root growth model with a shoot growth model. The root growth model is written in C (see Listing 1) and can be expressed as

$$R_{t+1} = R_t \times r_r \times dt + R_t. \tag{1}$$

The inputs to the root growth model are

- R_t: the root mass at time t
- r_r: the relative root growth rate
- dt: the time step

and the output of the root growth model is R_{t+1}, the root mass at the next time step. The shoot growth model is written in Python (see Listing 2) and can be expressed as

$$S_{t+1} = S_t \times r_s \times dt + S_t - (R_{t+1} - R_t). \tag{2}$$

The inputs to the shoot model are

- S_t: the shoot mass at time t
- r_s: the relative shoot growth rate

- dt: the time step
- R_t: the root mass at time t
- R_{t+1}: the root mass at time $t+1$

and the output of the shoot growth model is S_{t+1}, the shoot mass at the next time step. While it is possible to run both models in isolation if all of the appropriate input variables are provided, the value of R_{t+1} calculated by the root growth model could be used as input to the shoot growth model. However, because the two models are written in different languages, the two models cannot be directly integrated. To integrate the models a scientist must either "manually" integrate them by running one model and then the other or translate one of the models into the language of the other so that it can be called directly. For a tightly coupled set of models such as these where one model depends directly on the result from the other at the current time step, manual integration is very inefficient for more than a handful of time steps even when done using a script. Although translating one or both of these simple toy models would be relatively straight forward, actual plant models are much more complex and could contain thousands of lines of code that would be time consuming to translate and result in unnecessary replication of model algorithms.

We present an Open Source Python package, `cis_interface`, for creating and running integration networks by connecting existing computational models written in different programming languages. `cis_interface` was developed as part of the Crops *in Silico* (Marshall-Colon et al. 2017) initiative to build a complete crop *in silico* from the level of the genes to the level of the field. `cis_interface` is available on Github[1] with full documentation[2] and can be installed from PyPI[3] via `pip install cis_interface` or conda-forge[4] via `conda -c conda-forge install cis_interface`. Although developed for the purpose of integrating plant biology models, `cis_interface` can be applied to any situation requiring coordination between models in the supported languages. §2 provides background information on existing efforts for cross-language model integration, §3 describes the methods used by the `cis_interface` package to integrate models, §5 presents several tests demonstrating the performance of message passing between integrated models, and §6 summarizes the features of the `cis_interface` package, describes a few use cases, and outlines areas of ongoing and future development.

2. BACKGROUND

Computational plant models are usually written with a very specific research question in mind (e.g. describing a specific metabolic pathway in C4 photosynthesis, Wang et al. 2014). The resulting model codes are often highly tuned for this one purpose and are written specifically for scientists within an isolated group of collaborators. As a result, it is unlikely that models produced by two independent groups will be directly compatible in terms of language, data format, or units (Marshall-Colon et al. 2017).

There are models written in Matlab (Zhu et al. 2013; Wang et al. 2014), Python (Pradal et al. 2009), C++ (Merks et al. 2011; Postma et al. 2017), Microsoft Excel (Sharkey 2016), VisualBasic (Humphries & Long 1995; Hall & Minchin 2013), R (Wang et al. 2015), Java (Song et al. 2013; Kappas et al. 2013), Fortran (Goudriaan & Laar 1994), and several domain specific languages (e.g. SBML, Hucka et al. 2003). The variety of data formats used by these codes is just as diverse. While some specific subfields have settled on standards for things like microarray gene expression data (e.g. MINiML, GEO 2017), many models use unique data formats that may or may not include metadata such as the data types, field names, or units. For example, while LPy (Boudon et al. 2012) and Houdini FX (Houdini FX 2018) both allow users to specify plant structures via L-systems, they differ slightly in their syntax and so are not directly compatible. Or, as another example, 3D canopy structures can be generated in any unit and standard 3D geometry formats like Ply (Turk 1994) and Obj (obj 1994) do not include units in their metadata. As a result, if an LPy model produces a canopy in Ply format with units of centimeters, the photosynthetic photon flux density (PPFD) output by a ray tracer like fastTracer (Song et al. 2013) (which expects a geometry in units of meters) would be incorrect.

As a result, integrating two biological models like the root and shoot models from §1 requires that the following questions be addressed first:

1. Orchestration: How will models be executed?
2. Communication: How will information be passed from one model to the next across languages?
3. Translation: How will data output by one model be translated into a format understood by the next model?

[1] https://github.com/cropsinsilico/cis_interface
[2] https://cropsinsilico.github.io/cis_interface/
[3] https://pypi.org/project/cis_interface/
[4] https://github.com/conda-forge/cis_interface-feedstock

These three questions can be addressed through manual integration (e.g. running the root model, converting the output root mass into the units expected by the shoot model, and then running the shoot model). However, manual integration is 1) computationally inefficient when many iterations are necessary, such as during parameter searches or steady state convergence tests, 2) unique to every integration, requiring the production of new scripts, and 3) complex when more than three models are being integrated. Within both computational biology and the larger scientific computing community, groups have developed around different solutions to the problem of connecting models. These solutions can generally be grouped into two categories: domain specific languages and frameworks.

2.1. *Domain Specific Language (DSL)*

One solution is to agree upon a language. The questions of orchestration, communication, and translation become trivial when models are all written in the same language. Beyond selecting a single programming language, many communities have developed dedicated domain specific languages (DSLs) for model representation. For example, the Systems Biology Markup Language (SBML Hucka et al. 2003) is an XML-based format for representing models of biological processes. Scientists can compose their models using SBML markup and then run their model using software designed to parse SBML files. Other DSLs for biological models include LPy (Boudon et al. 2012), CellML (Cuellar et al. 2003), FLAME (Coakley et al. 2012), and BioPAX (Demir et al. 2010). Dedicated model DSLs improve the reusability of models written in the DSL and have the advantage of often implicitly handling data transformation. However, they provide no help for existing models that are not written in the DSL or those that cannot be expressed within the constraints of the DSL. Furthermore, learning a new DSL, in addition to a programming language, can be daunting.

2.2. *Workflow/Data Flow/Framework*

Another approach is a workflow/data flow/framework tool for models that are written in the same programming language or can be wrapped using an intermediary (e.g. Cython, Behnel et al. 2011). For example, there are many generic tools in Python for coordinating the execution of tasks in parallel or serial (e.g. Babuji et al. 2018; cel 2018; lui 2018). While these tools are powerful for organizing complex work flows, it is the user's onus to make sure that the components they connect are compatible and handle any data transformation that might be necessary, e.g. unit conversion or field selection.

There are also domain specific frameworks for connecting biological models. For example, OpenAlea (Pradal et al. 2015) allows users to compose networks from different components including plant models, analysis tools, and visualization tools. Domain specific frameworks are more intuitive and ultimately more flexible than DSLs in the types of operations they allow models to perform since they have access to the full power of a programming language. However, they are stricter in several respects: 1) the model must be written in (or exposable to), the language of the framework and 2) the model must be written in a way that is aware of the framework and/or the format of models with which it will interact. Like DSLs, frameworks also provided limited support for models written without the framework in mind.

cis_interface is an example of a framework that overcomes these issues by exposing simple and easily accessible interfaces in the language of the model (See §3.2.6) that permit messages to be passed between the model processes as they run in parallel (See §3.2). As a result, models writers only need knowledge of the language in which their model is written.

3. METHODS

cis_interface was designed to address the questions from §2 while being:

- **Easy to use.** Require as little modification to the model source code as possible and only in the language of the model itself.
- **Efficient.** Allow models to run in parallel with asynchronous communication that doesn't block model execution when a message is sent, but has yet to be received.
- **Flexible.** Provide the same interface to the user, regardless of the communication mechanism being used or the platform the model is being executed on.

To accomplish these objectives, cis_interface is built upon several well-established open-source software packages along with tools developed explicitly for cis_interface. cis_interface currently support models written in Python, Matlab, C, and C++ with additional Domain Specific Language (DSL) support for LPy models (Boudon et al. 2012). Support for additional languages is planned for future development (See §6.3.1). Statement on novelty

3.1. *Orchestration*

cis_interface is executed via a command line interface (CLI), cisrun with specification files (§3.1.1) as input. Based on the information contained in the specification files, cis_interface establishes a network of asynchronous communication channels (§3.2), and launches the models on new processes (§3.1.2).

3.1.1. *Specification Files*

Users specify information about models and integration networks via declarative YAML files (Ben-Kiki et al. 2009). The YAML file format was selected because it is human readable and there are many existing tools for parsing YAML formats in different programming languages (e.g. Simonov 2006; pyy 2006; jsy 2011). The declarative format allows user to specify exactly what they want to do, without describing how it should be done. While the information about models and integration networks can be contained in a single YAML file, the information can naturally be split between two or more files, one (or more) containing information about the model(s) and one containing the connections comprising the integration network. This separation is advantageous because the model YAML can be re-used, unchanged, in conjunction with other integration networks.

Model YAMLs include information about the location of the model source code, the language the model is written in, how the model should be run, and any input or output variables including their data type and units. Integration networks are specified by declaring the connections between models and connections are declared by pairing an output variable from one model with the input variable of another model. The cis_interface CLI sets up the necessary communication mechanisms to then direct data from one model to the next in the specified pattern. Models can have as many input and/or output variables as is desired and connections between models are specified by references to the input/output variables associated with each model. In addition, input and output variables can also be connected to files. This format allows users the flexibility to create complex integration networks and test models in isolation before running the entire integration network.

3.1.2. *Model Execution*

cis_interface launches each model in an integration on its own process, allowing models to complete independent operations in parallel and complete tasks more quickly. For example, the root and shoot growth models from §1 require 2.42 s and 4.06 s respectively to run for 100 time steps in isolation. If these two models were manually integrated in serial via the command line, the integration would require a total of 6.48 s. However, the same integration requires only 4.52 s using cis_interface, a speed-up of 1.43. The exact speedup provided by parallel integration using cis_interface depends on how independent the models are and how equally distributed the work is between the models. If the models are fully independent, the speedup is limited by the time required to execute the slowest model. In the example, this would be the shoot growth model which required almost twice the time to execute as the root model. If the models are dependent on one-another, the models will have to wait for messages to be passed and the speedup will not be as great (see §**??** for additional discussion of parallelism and the speedups achievable using cis_interface). Fix these numbers for most recent version of the code.

While every model is executed on a new process, how the model is handled depends on the language it is written in. Models written in interpreted languages (Python and Matlab) are executed on the command line with the interpreter. In the case of Matlab, where a significant amount of time is required to start the Matlab interpreter (See §5.2), Matlab shared engines are used to execute Matlab models. Matlab shared engines are Matlab instances that other processes can submit Matlab code to for execution. While shared engines also require the same amount of time to start as a standard instance of the Matlab interpreter, they can be started in advance and then reused.

For the compiled languages (C and C++) there are a few options. The user can compile the model themselves, provided they link against the appropriate cis_interface header library. Alternatively, users can provide the location of the model source code and let one of several cis_interface drivers handle the compilation, including linking against the appropriate cis_interface interface library. cis_interface also has support for compiling models using Make (Stallman et al. 2004) and CMake (Martin & Hoffman 2006) for models that already have a Makefile or CMakeLists.txt. To use these tools to compile a model, lines are added to the recipe in order to allow linking against cis_interface.

Once the models are running, cis_interface uses threads on the master process to monitor the progress of the model and report back model output (e.g. log messages printed to stdout or stderr) and any status changes. If a model issues any errors, the master process will shut down any model processes that are still running and close any connections, discarding any unprocessed messages. If a model completes without any errors, the master process will cleanup any connections that are no longer required after waiting for all messages to be processed. The master process will only complete once an error is encountered or all model processes have completed.

The algorithms for a generic model driver and the master thread can be found in Algorithms 1 & 2 respectively.

Algorithm 1 Generic model driver class. This algorithm does not represent the actual implementation of any one of the languages supported by cis_interface. Each of the language-specific drivers will have slightly different methods for compilation (turning the source code into an executable), launching the model process, and monitoring the model's status via output and error codes. However, this algorithm does provided the general behavior pattern upon which every model driver is based.

```
class ModelDriver($name, args, yml$)
    $errors \leftarrow []$
    $executable \leftarrow$ Compile(args)
    $process \leftarrow$ ModelProcess on a new process
    $thread \leftarrow$ ModelThread on a new thread

    method ModelProcess()
        Executable
    end method

    method ModelThread()
        while IsAlive(ModelProcess) do
            if HasOutput(ModelProcess) then
                Print GetOutput(ModelProcess)
            else
                Sleep
            end if
        end while
        if HasErrors(ModelProcess) then
            $errors \leftarrow$ GetErrors(ModelProcess)
        end if
    end method

    method Terminate()
        Kill(ModelProcess)
    end method

    method Stop()
        while not $errors$ do
            Sleep
        end while
        Terminate
    end method

end class
```

Algorithm 2 Basic steps executed on the master thread to create, start, and monitor model & connection drivers. This pseudocode is not an exhaustive reproduction of the entire implementation, but represents the broad strokes needed to understand the purpose and behavior of the master thread. The algorithm for the ModelDriver and ConnectionDriver can be found in Algorithms 1 & 6 respectively.

```
$models \leftarrow$ Parse YAML files
$inputChannels \leftarrow Map()$
$outputChannels \leftarrow Map()$
                                                        ▷ Create Drivers
for all $minmodels$ do
    for all $iinm.inputs$ do
        if not $i.isFile$ then
            $inputAddress \leftarrow i.address$
        else
            $inputAddress \leftarrow$ 'generate'
        end if
        $i.driver \leftarrow$ ConnectionDriver($inputAddress$, 'generate', $i$)
        $inputChannels[i.name] = i$
    end for
end for
for all $minmodels$ do
```

Algorithm 3 Continuation of Algorithm 2

```
    for all $o \in m.outputs$ do
        if $o.isFile$ then
            $outputAddress \leftarrow o.address$
        else
            if $o.args \in inpurChannels$ then
                $outputAddress \leftarrow inputChannels[o.args]$
            else
                Error
            end if
        end if
        $o.driver \leftarrow$ ConnectionDriver($`generate', outputAddress, o$)
        $outputChannels[o.name] = o$
    end for
end for
for all $minmodels$ do
    $m.driver \leftarrow$ ModelDriver($m.name, m.args, m$)
end for
                                                        ▷ Start Drivers
for all $minmodels$ do
    for all $iinm.inputs$ do
        $i.driver$.Start
    end for
end for
for all $minmodels$ do
    for all $o \in m.outputs$ do
        $o.driver$.Start
    end for
end for
for all $minmodels$ do
    $m.driver$.Start
end for
                                                        ▷ Monitor Drivers
$errorFlag \leftarrow false$
while not $errorFlag$ do
    for all $minmodels$ do
        if $m.driver.errors$ then
            $errorFlag \leftarrow true$
            break
        end if
    end for
end while
                                                        ▷ Stop Drivers
if $errorFlag$ then
    $allErrors \leftarrow []$
    for all $minmodels$ do
        $m$.Terminate
        Append $allErrors$, $m.errors$
    end for
    Error($allErrors$)
else
    for all $minmodels$ do
        $m$.Stop
    end for
end if
```

make sure this is split correctly across pages

3.2. *Communication*

In order for integrations to be possible, models must be able to send and receive information to and from other models written in different languages via some communication mechanism. Communication within `cis_interface` integration networks was designed to be flexible in terms of the languages, platforms, and data types available. To accomplish this, `cis_interface` leverages three different tools for communication, System V IPC Queues (§3.2.1), ZeroMQ (§3.2.2), and RabbitMQ (§3.2.3), which each have their own strengths and weaknesses. The particular communication mechanism used by `cis_interface` is determined by the platform, available libraries, and integration strategy. However, regardless of the communication mechanisms, the user will always use the same interface in the language of their model (See §3.2.6).

cis_interface includes its own implementation of asynchronous communication via each of the communication mechanisms that are supported. While there are existing tools for asynchronous communication using each of these tools, using a tool developed specifically for cis_interface allows for a more uniform treatment of the different communication mechanisms, greater control over how threads are managed (e.g. killing a thread when messages cannot be interpreted), and a more seamless coordination with other cis_interface processes (e.g. enforcing dependencies on other model and connection threads).

3.2.1. *System V IPC Queues*

The first communication mechanism used by cis_interface was System V interprocess communication (IPC) message queues (Rusling 1999) on Posix (Linux and Mac OS X) systems. IPC message queues allow messages to be passed between models running on separate processes on the same machine. While IPC message queues are simple, fast (See §5.1), and are built into most Posix operating systems, they do not work in all situations. IPC queues are not natively supported by Windows operating systems and do not allow communication between remote processes. In addition, IPC queues also have relatively low default message size limits on Mac OS X systems (2048 bytes). Once the queue is full or if the message is larger than the limit, any process attempting to send an additional message will block until a sufficient number of messages has been removed from the queue to accommodate the new message. For messages larger than the limit, the sending process will block indefinitely. This can be handled by splitting large messages into multiple smaller messages (See §3.2.5), however, the time required to send a message increases with the number of message it must be broken into (See §5.1). These limits make sending large messages relatively inefficient when compared with other communications mechanisms. As a result, IPC queues are used by cis_interface as a fallback on Posix systems if none of the other supported communication libraries have not been installed.

3.2.2. *ZeroMQ*

The preferred communication mechanism used by cis_interface are ZeroMQ sockets (ZMQ Akgul 2013). ZMQ provides broker-less communication via a number of protocols and patterns with bindings in a wide variety of languages that can be installed on Posix and Windows operating systems. ZMQ was adopted by cis_interface in order to allow support on Windows and for future target languages (See §6.3) that could not be accomplished using IPC queues. In addition, while ZMQ allows interprocess communication via IPC, ZMQ also supports protocols for distributed communication via an Internet Protocol (IP) network. While cis_interface does not currently support using these protocols for distributed integration networks, this is a future development plan that has been prepared for.

3.2.3. *RabbitMQ*

While broker-less communication like ZMQ is light weight and fast, it is not as fault tolerant as brokered messaging systems that confirm message delivery and can resend dropped messages. Resilience to dropped messages, while not as necessary for integrations running entirely on a local machine, will be more important for integrations running on distributed resources with less reliable connections. As a result, cis_interface includes support for brokered communication via RabbitMQ (RMQ RMQ 2007) that will be used during future development to allow integrations to run on distributed resources or include remote models run as services (§6.3.2). Due to the slower message speed, cis_interface does not currently use RMQ for communication unless explicitly specified by the user in their integration network YAML.

3.2.4. *Asynchronous Message Passing*

The same asynchronous communication strategy is used with each of the communication mechanisms supported by cis_interface. Models do not block on sending messages to output channels; the model is free to continue working on its task while an *output driver* waits for the message to be routed and received on a separate master process thread. Similarly, an *input driver* continuously pings input channels on another thread, moving received messages into a buffer queue so that they are ready and waiting for the receiving model when it asks for input. These drivers work together to move message along from one model to the next like a conveyor belt. As a result, models in complex integration networks are not affected by the rate at which dependent model consume their output and the speedup offered by running the models in parallel is improved. Figure 1 describes the general flow of messages, Algorithms 4 & 5 describe the asynchronous output & input channel procedures, and Algorithm 6 describes the procedure followed by input & output connection drivers.

1. A model sends a message in the form of a native data object via a language-specific API to one of the output channels declared in the model YAML. The output channel interface encodes the message and sends it.

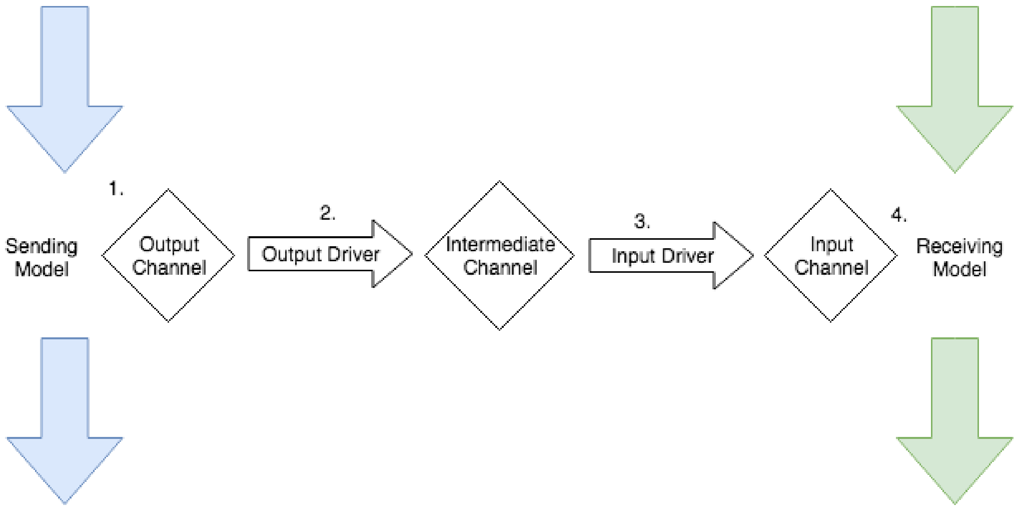

Figure 1: Diagram of how messages are passed asynchronously using input/output drivers and an intermediate channel.

2. An output connection driver (written in Python) runs in a separate thread on the master process, listening to the model output channel. When the model sends a message, the output connection driver checks that the message is in the expected format and then forwards it to an intermediate channel. The intermediate channel is used as a buffer for support on distributed architectures (e.g. if one model is running on a remote machine). In these cases, the intermediate channel will connect to a RMQ broker using security credentials.

3. An input connection driver (also written in Python) runs in a separate thread, listening to the intermediate channel. When a message is received, it is then forwarded to the input channel of the receiving model as specified in the integration network YAML.

4. The receiving model receives the message in the form of an analogous native data object. Interface receive calls can be either blocking or non-blocking, but are blocking by default.

Algorithm 4 Asynchronous communication class for sending messages to an output channel. The procedures Open, Close, Send, & IsOpen are procedures that are implemented by external communication packages and different slightly from one communication mechanism to the next.

```
class AsyncSender(address)
    backlog ← []
    x ← Open(address)
    Run BacklogThread on a new thread

    method BacklogThread()
        while IsOpen(x) do
            if length(backlog) > 0 then
                flag ← Send(x, backlog[0])
                if flag then
                    Remove backlog[0]
                else
                    Close(x)
                end if
            else
                Sleep
            end if
        end while
    end method

    method SendBacklog(msg)
        if IsOpen(x) then
            Append backlog, msg
        end if
        Return IsOpen(x)
    end method

    method Close()
        Close(x)
        backlog ← []
    end method

end class
```

Algorithm 5 Asynchronous communication class for receiving messages from an input channel. The procedures Open, Close, Recv, IsOpen, & IsMessageWaiting are procedures that are implemented by external communication packages and different slightly from one communication mechanism to the next.

```
class AsyncReceiver($address$)
    $backlog \leftarrow []$
    $x \leftarrow$ Open($address$)
    $thread \leftarrow$ BacklogThread on a new thread

    method BacklogThread()
        while IsOpen($x$) do
            if IsMessageWaiting($x$) then
                $flag, msg \leftarrow$ Recv($x$)
                if $flag$ then
                    Append $backlog$, $msg$
                else
                    Close($x$)
                end if
            else
                Sleep
            end if
        end while
    end method

    method RecvBacklog($msg$)
        if $length(backlog) > 0$ then
            $msg \leftarrow backlog[0]$
            Remove $backlog[0]$
            Return true, $msg$
        else
            Return IsOpen($x$), ”
        end if
    end method

    method Close()
        Close($x$)
        $backlog \leftarrow []$
    end method

end class
```

Algorithm 6 Connection Driver

```
class ConnectionDriver(inputAddress, outputAddress, yml)
    input $\leftarrow$ AsyncReceiver(inputAddress)
    output $\leftarrow$ AsyncSender(outputAddress)
    thread $\leftarrow$ ConnectionThread on a new thread

    method ConnectionThread()
        while true do
            flag, msg $\leftarrow$ input.RecvBacklog
            if not flag then
                break
            end if
            if msg then
                Transform msg based on units and expected data format
                flag $\leftarrow$ output.SendBacklog(msg)
                if not flag then
                    break
                end if
            else
                Sleep
            end if
        end while
        input.CloseAsync
        output.CloseAsync
    end method

    method Terminate()
        input.CloseAsync
        output.CloseAsync
    end method

    method Stop()
        while IsOpen(input.x) and IsOpen(output.x) do
            Sleep
        end while
        Terminate
    end method

end class
```

discuss algorithm

3.2.5. *Sending/Receiving Large Messages*

All of the communication tools leveraged by cis_interface have intrinsic limits on the allowed size for a single message. Some of these limits can be quite large (2^{20} bytes for ZMQ and RMQ), while others are very limiting (2048 bytes on Mac OS X for IPC queues). Although messages consisting of a few scalars are unlikely to exceed these limits, biological inputs and outputs are often much more complex. For example, structural data represented as a 3D mesh can easily exceed these limits. To handle messages that are larger than the limit of the communication mechanism being used, cis_interface splits the message up into multiple smaller messages. In addition, for large messages, cis_interface creates new, temporary communication channels that are used exclusively for a single message and then destroyed. The address associated with the temporary channel is sent in header information as a message on the main channel along with metadata about the message that will be sent through the temporary channel like size and data type. Temporary channels are used for large messages to prevent mistakenly combining the pieces from two different large messages that were received at the same time such as in the case that a model is receiving input from two different models working in parallel.

algorithm

3.2.6. *Interface*

cis_interface provides interface functions/classes for communication that are written in each of the supported languages. Language specific interfaces allow users to program in the language(s) with which they are already familiar. The Python interface provides communication classes for sending and receiving messages. The Matlab interface provides a simple wrapper class for the Python class, that exposes the appropriate methods and handles conversion between Python and Matlab data types. The C interface provides structures and functions for accessing communication channels and sending/receiving messages. The C++ interface provides classes that wrap the C structures with functions called as methods.

In addition to basic input and output, each interface also provides access to more complex data types and communication patterns.

3.3. *Transformation*

Messages are passed as raw bytes. In order to understand the messages begin passed, parallel processes that communicate must agree upon the format used to do so. Without community standards, different models will often use very different data formats for their input and output. Differences between data formats can include, but are not limited to, type, precision, fields, or units. While some data formats are self-descriptive and include these types of information as meta data, this is not true of all data formats. To combat this, `cis_interface` requires models to explicitly specify the format of input and output expected by a model in the model YAML. `cis_interface` can then handle a number of conversions between models without prompting as well as serialization/deserialization to the correct type in each of the supported languages. Data formats currently supported by `cis_interface` include:

- Scalars (e.g. integers, decimals)
- Arrays
- Text-encoded tables (e.g. CSV or tab-delimited)
- Pandas data frames (pan 2008)
- 3D geometry structures (PLY (Turk 1994) & OBJ (obj 1994))

In addition, `cis_interface` offers the option to specify units for scalars, arrays, tabular data, and pandas data frames. Units are tracked using the `unyt` package (Goldbaum et al. 2018), which allows physical units to be associated with scalars and arrays. If two models use different units (and both are specified), `cis_interface` will automatically perform the necessary conversions before passing data from one model to the next.

algorithm?

4. WORKED EXAMPLE

In order to illustrate how `cis_interface` is intended to be used by model writers, the following walks through the integration of the two examples models from §1. All of the source code and YAML files discussed in this section are available in the `cis_interface` GitHub repository and will be included with future releases on PyPI and conda-forge. This example assumes that the user starts with the following existing model source code:

```c
double calc_root_mass(double r_r, double dt, double R_t) {
  return R_t + (R_t * r_r * dt);
};
```

Listing 1: root.h: Source code for root model in C.

```python
def calc_shoot_mass(r_s, dt, S_t, R_t, R_tp1):
    r"""Calculate the shoot mass.

    Args:
        r_s (float): Relative shoot growth rate.
        dt (float): The time step.
        S_t (float): Previous shoot mass.
        R_t (float): Previous root mass.
        R_tp1 (float): Root mass at the next timestep.

    Returns:
        float: Shoot mass at the next timestep.

    """
    return (S_t * r_s * dt) + S_t - (R_tp1 - R_t)
```

Listing 2: shoot.py: Source code for shoot model in Python.

The root source code in Listing 1 is a standalone C header library containing a single function `calc_root_mass` that calculates and returns the root mass at the next time step (a double precision floating point number) from input of the relative root growth rate (`r_r`, double), the time step (`dt`, double), and the root mass as the current time step (`R_t`, double). The shoot source code in Listing 2 is a standalone Python module containing a single function `calc_shoot_mass` that calculates and returns the shoot mass at the next time step from input of the relative shoot growth rate (`r_s`), the time step (`dt`, double), the shoot mass at the current time step (`S_t`), the root mass at the current time step (`R_t`), and the root mass at the next time step (`R_tp1`).

Both of the example models used here start out in the form of the function. If possible, model writers should try to pose or wrap their model as a function prior to starting the integration process. This format makes the integration process easier and allows for the model to be used in a larger number of integration patterns.

4.1. *Model YAMLs*

The first step in the integration process is to create model YAML files describing the models including the location of the source code and the input/output variables. This step only has to be completed once per model and the model YAML can then be reused to include the model in any integration. The model YAMLs for the example root and shoot models are shown in Listings 3 and 4 respectively.

```yaml
model:
  name: RootModel
  language: c
  args: ./src/root_wrapper.c
  inputs:
    - name: root_growth_rate
      units: hr**-1
    - name: init_root_mass
      units: g
    - name: root_time_step
      units: hr
  outputs:
    - name: next_root_mass
      units: g
```

Listing 3: root.yml: Model YAML specification file for root model.

```yaml
model:
  name: ShootModel
  language: python
  args: ./src/shoot_wrapper.py
  inputs:
    - name: shoot_growth_rate
      units: d**-1
    - name: init_shoot_mass
      units: kg
    - name: init_root_mass_shoot
      units: kg
    - name: shoot_time_step
      units: d
    - name: next_root_mass
      units: kg
  outputs:
    - name: next_shoot_mass
      units: kg
```

Listing 4: shoot.yml: Model YAML specification file for shoot model.

Each model yaml must include, at minimum, a name that is unique within the set of models being integrated, the language that the model is written in, and the location of the model source code (this can be a list/sequence if the model is split between multiple files). If the path to the source files is not absolute, as in the case of Listings 3 and 4, the path is interpreted as being relative to the directory containing the model YAML. In this case, the source code for each model is a wrapper (described in §4.2) that calls the model code such that no modification needs to be done to the original model code in order to integrate it.

In additional to the basic required fields, each model should have an entry for each of its input/output variables in the appropriate `"inputs"` or `"outputs"` section. At a minimum, input/output entries should include a name that will be used to identify a communication channel connected to the model. It is also highly recommended that the units of each variable also be specified (if applicable) so that `cis_interface` can handle any necessary conversions.

There are other optional model and input/output keywords for more advanced cases (e.g. compilation flags or data types), but they are beyond the scope of this limited introduction. Additional examples and descriptions of these options can be found in the documentation [5].

It should also be noted that it is possible to pass multiple variables as a single input/output. However, unless the variables will *always* be coupled in the model, it is advised that every variable be specified separately for clarity and to allow flexibility in the way other models can integrate with it. In addition, if brevity is a concern because a model has a large number of input variables, there are features currently under active development (see §**??**) that will allow fields to be aggregated within connection YAMLs and simplify the send and receive calls in such cases.

4.2. *Wrapper*

Once the model YAMLs are complete, the next step is to write a wrapper for each model that will make the necessary calls to the `cis_interface` API to set up channels and send/receive messages. Writing the wrapper is the most involved step in any integration as it requires the most thought about how one model should interact with others, but this wrapper is written in the same language as the model and so should be a comfortable process for the model author since it is a language they are already familiar with. There is work in progress to add features which will allow `cis_interface` to perform this step automatically based on YAML options for simple cases such as single loops or if statements (see §6.3.3), but these will not cover all potential cases and it is instructive to walk through the process and understand how the wrapper acts as an intermediary between `cis_interface` and the actual model function.

4.2.1. *Integration Pattern*

To write the wrapper, a integration pattern must first be selected. For the root/shoot model integration, the obvious pattern is to loop over time steps, outputting the evolution of the root and shoot masses over time. Although this is the pattern adopted for this example, you could imagine another pattern in which the loop is performed over the growth rates in a parameter sweep that would require a slightly different wrapper.

Once an integration pattern is selected, the wrappers can be flushed out in pseudocode independently by assuming that all input is received from a file and all output is sent to a file (i.e. the models are independent). The pseudocode for the root and shoot model wrappers is shown in Algorithms 7 and 8 respectively.

Algorithm 7 Root wrapper

```
Receive $r_r, R_t$
Send $R_t$                                        ▷ So that there is a complete record.
while true do
    if $dt$ available then
        Receive $dt$
    else
        Break
    end if
    $R_{t+1} \leftarrow \texttt{calc\_root\_mass}(r_r, dt, R_t)$
    $R_t \leftarrow R_{t+1}$
    Send $R_t$
end while
```

Algorithm 8 Shoot wrapper

```
Receive $r_s, S_t, R_t$
Send $S_t$                                        ▷ So that there is a complete record.
while true do
    if $dt$ available then
        Receive $dt$
    else
        Break
    end if
    Receive $R_{t+1}$
    $S_{t+1} \leftarrow \texttt{calc\_shoot\_mass}(r_s, dt, S_t, R_t, R_{t+1})$
    $S_t \leftarrow S_{t+1}$
    $R_t \leftarrow R_{t+1}$
    Send $S_t$
end while
```

[5] `https://cropsinsilico.github.io/cis_interface/yaml.html`

Both models follow a similar pattern. They first receive static variables that will not change over the course of the run and then send the initial mass to output so that the output record is a complete history of the mass. Then both entire a while loop that is only broken when there is not a new time step available. When a new time step is available, it is received along with the next root mass in the case of the shoot model. Next, both model wrappers make the call to the actual model function. Finally, the models advance the time step by setting the masses to those calculated for the next time step and output the calculated mass.

4.2.2. *Wrapper Code*

With some translation into the appropriate language and the addition of calls to the appropriate cis_interface model interface to establish input and output channels, the code for the root and shoot model wrappers can then be written as Listings 5 and 6 respectively. In the syntax of the target language, the wrapper should be executable. For Python/Matlab this might be a script, while for C/C++ this should be compilable as an executable with a `main` function.

```c
#include <stdio.h>
#include <stdlib.h>
// Include C interface
#include "CisInterface.h"
// Include C header containing model calculation
#include "root.h"


int main(int argc, char *argv[]) {
  int i, ret;
  double r_r, dt, R_t, R_tp1;

  // Create input/output channels
  cisInput_t RootGrowthRate = cisInput("root_growth_rate");
  cisInput_t InitRootMass = cisInput("init_root_mass");
  cisInput_t TimeStep = cisInput("root_time_step");
  cisOutput_t NextRootMass = cisOutputFmt("next_root_mass", "%lf\n");

  // Receive root growth rate
  ret = cisRecv(RootGrowthRate, &r_r);
  if (ret < 0) {
    printf("root: Error receiving root growth rate.\n");
    return -1;
  }
  printf("root: Received root growth rate: %lf\n", r_r);

  // Receive initial root mass
  ret = cisRecv(InitRootMass, &R_t);
  if (ret < 0) {
    printf("root: Error receiving initial root mass.\n");
    return -1;
  }
  printf("root: Received initial root mass: %lf\n", R_t);

  // Send initial root mass
  ret = cisSend(NextRootMass, R_t);
  if (ret < 0) {
    printf("root: Error sending initial root mass.\n");
    return -1;
  }

  // Keep advancing until there arn't any new input times
  i = 0;
  while (1) {

    // Receive the time step
    ret = cisRecv(TimeStep, &dt);
    if (ret < 0) {
      printf("root: No more time steps.\n");
      break;
    }
    printf("root: Received next time step: %lf\n", dt);

    // Calculate root mass
    R_tp1 = calc_root_mass(r_r, dt, R_t);
    printf("root: Calculated next root mass: %lf\n", R_tp1);

    // Output root mass
    ret = cisSend(NextRootMass, R_tp1);
    if (ret < 0) {
      printf("root: Error sending root mass for timestep %d.\n", i+1);
      return -1;
    }

    // Advance root mass to next timestep
    R_t = R_tp1;
    i++;

  }

  return 0;

}
```

Listing 5: root_wrapper.c: Wrapper source code for root model in C.

```
# Import Python interface
from cis_interface.interface import CisInput, CisOutput
# Import Python module containing model calculation
from shoot import calc_shoot_mass


# Create input/output channels
ShootGrowthRate = CisInput('shoot_growth_rate')
InitShootMass = CisInput('init_shoot_mass')
InitRootMass = CisInput('init_root_mass_shoot')
TimeStep = CisInput('shoot_time_step')
NextRootMass = CisInput('next_root_mass')
NextShootMass = CisOutput('next_shoot_mass', '%lf\n')

# Receive shoot growth rate
flag, input = ShootGrowthRate.recv()
if not flag:
    raise RuntimeError('shoot: Error receiving shoot growth rate.')
r_s = input[0]
print('shoot: Received shoot growth rate: %f' % r_s)

# Receive initial shoot mass
flag, input = InitShootMass.recv()
if not flag:
    raise RuntimeError('shoot: Error receiving initial shootmass.')
S_t = input[0]
print('shoot: Received initial shoot mass: %f' % S_t)

# Receive inital root mass
flag, input = InitRootMass.recv()
if not flag:
    raise RuntimeError('shoot: Error receiving initial root mass.')
R_t = input[0]
print('shoot: Received initial root mass: %f' % R_t)

# Send initial shoot mass
flag = NextShootMass.send(S_t)
if not flag:
    raise RuntimeError('shoot: Error sending initial shoot mass.')

# Keep advancing until there arn't any new input times
i = 0
while True:

    # Receive the time step
    flag, input = TimeStep.recv()
    if not flag:
        print('shoot: No more time steps.')
        break
    dt = input[0]
    print('shoot: Received next time step: %f' % dt)

    # Receive the next root mass
    flag, input = NextRootMass.recv()
    if not flag:
```

```
56            # This raises an error because there must be a root mass for each time step
57            raise RuntimeError('shoot: Error receiving root mass for timestep %d.' % (i + 1))
58        R_tp1 = input[0]
59        print('shoot: Received next root mass: %f' % R_tp1)
60
61        # Calculate shoot mass
62        S_tp1 = calc_shoot_mass(r_s, dt, S_t, R_t, R_tp1)
63        print('shoot: Calculated next shoot mass: %f' % S_tp1)
64
65        # Output shoot mass
66        flag = NextShootMass.send(S_tp1)
67        if not flag:
68            raise RuntimeError('shoot: Error sending shoot amss for timestep %d.' % (i + 1))
69
70        # Advance masses to next timestep
71        S_t = S_tp1
72        R_t = R_tp1
73        i += 1
```

Listing 6: shoot_wrapper.py: Wrapper source code for shoot model in Python

Prior to any calls, both model wrappers must first locate the necessary cis_interface and model code that they will call on via directives. In the C root model wrapper, this takes the form of #include statements on lines 3-6. In the Python shoot model wrapper, this takes the form of import statements. cis_interface takes care of the paths at compile/runtime so that the appropriate API library can be located while it is expected that the model source code is in the same directory as the wrapper and automatically discovered.

The first step in each model wrapper is to connect to the appropriate channels via the cis_interface interface. This step occurs on lines 14-17 in the root model wrapper (Listing 5) and lines 8-13 in the shoot model wrapper (Listing 6). Regardless of the language, each has a similar call signature. Inputs require a single input that is a string specifying the name of the model input from the model YAML that the returned object should access. Outputs are similar in that their first argument is a string specifying the name of the model output from the mode YAML that the returned object should access. In addition, outputs can also be provided with a format string that tells cis_interface interface what data type to expect how the output should be formatted if it is sent to a file. Additional information about how these strings are processed can be found in the "C-Style Format Strings" section of the documentation[6].

The next step in each model wrapper is to receive the static input variables (those that are not being looped over). This step occurs on lines 19-33 in the root model wrapper (Listing 5) and lines 15-34 in the shoot model wrapper (Listing 6). Next, each model wrapper sends the initial mass to the output so that the output will contain the entire mass history. This occurs on lines 58-63 in the root model wrapper and lines 36-39 in the shoot model wrapper. The majority of these lines are devoted to error handling and print statements. The syntax for sending/receiving messages differs slightly from language to language, but each will return a flag indicating if the send/receive was successful or not. If an input channel is still open, a receive call will block until the channel is closed or a message is received. If the input channel is closed (either because of an error or because it was closed by the source), the flag will indicate this and the received message should not be used. If an output channel is open, a send call will return immediately while a worker thread handles asynchronous completion of the send request. If an output channel is closed, a send call will return immediately with a flag indicating failure. Additional information about the interface calls for sending and receiving can be found in the "Model Interface" section of the documentation[7].

Once the static variables have been received and the initial masses have been sent to output, both model wrappers enter their while loop. The flag returned by the receive call for the new time step is then used to decide if the loop should be broken (lines 46-52 in the root model wrapper and lines 45-51 in the shoot model wrapper). In the case of the shoot model wrapper, a failure to also receive the root mass for the next time step results in an error as it is required that there be a new root mass for each new time step (lines 53-59). Finally, both model wrappers make calls to the appropriate model functions, send the result to the output, and advance the masses to the next time step.

While it is valid to include these calls in the model code itself, this is not advised for several reasons. First, reproducibility is important. If there is a model that has been used to produce past scientific results, it is important that the model is preserved (in the form it was in during production of the results) so that other scientists can reproduce that result if need be. The use of a wrapper allows the original model code to be preserved while still adding the necessary API calls for integration via cis_interface. Second, although the original model code could be duplicated and then altered to preserve the original code, this duplication results in redundancy and complicates the process

[6] https://cropsinsilico.github.io/cis_interface/c_format_strings.html
[7] link

of incorporating changes that may occur in the original model code due to bug fixes or added features. Third, it is possible that a model may be used in several different integration patterns. If the model code is altered directly, it will be difficult to add new integration patterns and there will again be redundant copies of the model.

4.3. *Connection YAMLS*

Once the wrappers are complete, it is time to write the connection YAML. Connection YAMLs contain information about how inputs and outputs should be connected between models and files.

4.3.1. *Isolated Models*

Before connecting the models to each other, it is useful to first write a connection YAML that connects all of a model's inputs and outputs to files so that it can be run (and tested) in isolation. In addition, much of the connection YAML for an isolated model can be reused in connection YAMLs for the model in an integration. The connection YAMLs for running the root and shoot models in isolation with file input/output are show in Listings 7 and 8 respectively.

```
connections:
  # Input connections
  - input: ./Input/root_growth_rate.txt
    output: root_growth_rate
    filetype: table
  - input: ./Input/init_root_mass.txt
    output: init_root_mass
    filetype: table
  - input: ./Input/timesteps.txt
    output: root_time_step
    filetype: table

  # Output connections
  - input: next_root_mass
    output: ./Output/root_output.txt
    filetype: table
    field_names: root_mass
```

Listing 7: root_files.yml: Connection YAML specification file for running the root model in isolation.

```
connections:
  # Input connections
  - input: ./Input/shoot_growth_rate.txt
    output: shoot_growth_rate
    filetype: table
  - input: ./Input/init_shoot_mass.txt
    output: init_shoot_mass
    filetype: table
  - input: ./Input/init_root_mass.txt
    output: init_root_mass_shoot
    filetype: table
  - input: ./Input/timesteps.txt
    output: shoot_time_step
    filetype: table
  - input: ./Input/root_output.txt
    output: next_root_mass
    filetype: table

  # Output connections
  - input: next_shoot_mass
    output: ./Output/shoot_output.txt
    filetype: table
    field_names: shoot_mass
```

Listing 8: shoot_files.yml: Model YAML specification file for running the shoot model in isolation.

Each model's connection YAML contains one item under `connections` for each input/output listed in the model's model YAML. Here they have been grouped by input and output, but they can be in any order. Every model input/output must have a connection for an integration to be valid. If there were a model input/output without a connection, whether to a file or a model, an error would be raised at run time.

Every connection has, at a minimum, an input and and output. Connection inputs can be model outputs, files, or a set of multiple model outputs or files. Connection outputs can be model inputs, files, or a set of multiple model inputs or files. An error will be raised if a connection just connects two files. In the connection YAMLs shown in Listings 7 and 8, all of the connections include a file as the input or output. Therefore, these connection YAMLs are closed systems. For connections including files, there is also the option of specifying a `filetype` that will determine how cis_interface reads the file. All of the files used in this example have `filetype` of `table`, indicating that the files are ASCII tables with some number of columns and rows, assumed to be tab-delimited by default. By default, cis_interface will read a table row by row, splitting each row up into its constituent column elements. The output connections in Listings 7 and 8 also include a `field_names` entry. This instructs cis_interface to add the designated field names as a header line in the output table that is produced.

There are many file formats that cis_interface supports which have additional YAML options. Information about these file formats and their YAML options can be found in the "Connection Options" subsection of the "YAML Files" section of the documentation[8].

4.3.2. *Integrated Models*

The connection YAML for integrating the two models is shown in Listing 9 and should look very similar to the connection YAMLs for running the models in isolation from Listings 7 and 8.

```
 1 connections:
 2   # Root input connections
 3   - input: ./Input/root_growth_rate.txt
 4     output: root_growth_rate
 5     filetype: table
 6   - input: ./Input/init_root_mass.txt
 7     output: init_root_mass
 8     filetype: table
 9   - input: ./Input/timesteps.txt
10     output: root_time_step
11     filetype: table
12 
13   # Root-to-shoot connection
14   - input: next_root_mass
15     output: next_root_mass
16 
17   # Shoot input connections
18   - input: ./Input/shoot_growth_rate.txt
19     output: shoot_growth_rate
20     filetype: table
21   - input: ./Input/init_shoot_mass.txt
22     output: init_shoot_mass
23     filetype: table
24   - input: ./Input/init_root_mass.txt
25     output: init_root_mass_shoot
26     filetype: table
27   - input: ./Input/timesteps.txt
28     output: shoot_time_step
29     filetype: table
30 
31   # Shoot output connection
32   - input: next_shoot_mass
33     output: ./Output/shoot_output.txt
34     filetype: table
35     field_names: shoot_mass
```

Listing 9: root_to_shoot.yml: Connection YAML specification file for the root/shoot integration.

The connections on lines 3-11 in Listing 9 are identical to lines 3-11 in Listing 7. Similarly, lines 18-29 and 32-35 in Listing 9 are identical to lines 3-14 and 20-23 respectively in Listing 8. This duplication results because there is only one connection between the two models. All of the remaining model inputs and outputs are still connected to files. As a result, the only new connection in Listing 9 is on lines 14-15. This connection between the two models occupies the `next_root_mass` output from the root model and the `next_root_mass` input to the shoot model, thereby eliminating the need for the output connection on lines 14-17 of Listing 7 and the input connection on lines 15-17 of Listing 8. Although the model output and input being connected in this example have the same name (`next_root_mass`), this is not a requirement.

A careful observer would note that there is a discrepancy between the units of these two models. For one, both models are receiving their time steps from the same file, but, as designated in their model YAMLs, the root model expects its time steps to have units of hours and the shoot model expects its time steps to have units of days. In addition, the root model outputs masses in units of grams, while the shoot model expects the root masses to have units of kilograms. However, there is no need to do any conversions within the models themselves. Because the units are

[8] https://cropsinsilico.github.io/cis_interface/yaml.html#connection-options

specified in the model YAMLs and the headers of the input tables, cis_interface is able to perform the appropriate conversions during the asynchronous transfer of data from one model to the next using the unyt package (Goldbaum et al. 2018).

In addition to units transformations, cis_interface is also capable of handling basic transformations between compatible data types (e.g. int to float or obj to ply) and inferring data types from sent/received messages in dynamically typed programming languages (i.e. Python & Matlab). For more advanced transformations, users can instruct cis_interface to use any arbitrary Python function to transform data by passing the module import path as a connection field.

4.4. *Running the Integration*

Integrations are run by calling the command line utility cisrun. cisrun takes as input one or more paths to YAML files describing the integration. These files can be passed in any order.

4.4.1. *Isolated Models*

To run the models in isolation, the user would pass cisrun the model YAML and the connection YAML specifying connections to files. For the root model this would be

```
cisrun root.yml root_files.yml
```

and for the shoot model this would be

```
cisrun shoot.yml shoot_files.yml.
```

4.4.2. *Integrated Models*

To run the integrated models, the suer would pass cisrun both model YAMLs and the connection YAML specifying the complete integration. For the example, this would be

```
cisrun root.yml shoot.yml root_to_shoot.yml
```

and the output to stdout would include interleaved messages from both models as well as log messages from cis_interface.

4.5. *Advanced Integration Topics*

The example integration used here was simplified in several respects which we would like to address for those who would use the package for more complex integrations.

Model Input/Output Complexity: Both of the example models has a limited and homogenous set of scalar inputs. More realistic models are unlikely to have data limited to such cases. cis_interface also has support for sending more complex data types like those discussed in §3.3. In addition, work is underway to add generic support for any data type that can be expressed as a JSON object (see §6.3.6). Examples using these different data types can be found in the "Formatted I/O" section of the documentation[9].

Model Networks: It is possible to build up complex integration networks using cis_interface one connection at a time. While not show here, integrations with more than two models are constructed in much the same way, one connection at a time. Tools are currently under development for composing integration networks visually from a palette of models (see §6.3.7) that will assist in this step. With or without such tools, it is recommended that users start with integrating models in isolation with input/output from/to files, then being testing each individual connection in isolation, and then slowly add the connections together to form a complete network.

Time Step Synchronization: The example models used here both took the same time step as input. This is an unlikely case in actual models, particularly for those that are capturing processes at different physical scales (e.g. cellular vs. field). Without strict constraints on the problem formulation and model relationships, it is not possible for cis_interface to determine how two time steps should be reconciled. Therefore, the current version of cis_interface requires the user to handle this part of the integration. There are plans for new features which allow automated creation and integration of models that can be described symbolically as Ordinary Differential Equations (ODEs, see §6.3.4). For two coupled ODE models, cis_interface may be able to determine the

[9] https://cropsinsilico.github.io/cis_interface/formatted_io.html

correct time step for synchronization, but, as an incorrect time step could drastically impact the results of an integration, it will still be recommend that users play an active roll in determining or evaluating the correct time step for a given integration.

Intermediate Output: In the integrated example presented, the output from the root model is passed to the shoot model without being output to a file. However, in real integrations, it is likely that users would want to know the value of this output as well. In order to output to both a model and a file, users must be able to direction connections to multiple channels (i.e. a model input and a file for output). While the current version of cis_interface has limited support for "forking" connections, full support for this feature is under way as part of the data aggregation feature (see §6.3.5) and will be a part of the version 1.0 release.

5. RESULTS & DISCUSSION

In order to evaluate cis_interface, performance tests were run on machines with Linux, Mac OS X, and Windows operating systems for different communication mechanisms, Python versions, and language combinations. During each run, N_{msg} messages of size S_{msg} were sent from a source model (in one language) to a destination model (in another language) which then sends the messages to an output file for verification. These are test models that do not perform any operations outside of passing the messages in order to accurately gauge the performance of the cis_interface machinery on its own. Performance tests were run using the perf package (Stinner 2018). Each run was repeated 10 times with warm-up runs in between to mitigate fluctuations due to external loads on the test machines.

5.1. *Communication Mechanism*

Figure 2 compares the execution times for the different communication mechanisms discussed in §3.2 for sending different numbers/sizes of messages from one Python model to another Python model. The tests were run on a Dell tower with Dual Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz processors running Ubuntu 14.04. The left panel of Figure 2 shows the total time required to run both models and send a varying number of 1000 bytes messages from one model to the other. The right panel of Figure 2 shows the total time required to run the models and send 5 messages of varying lengths from one to the other.

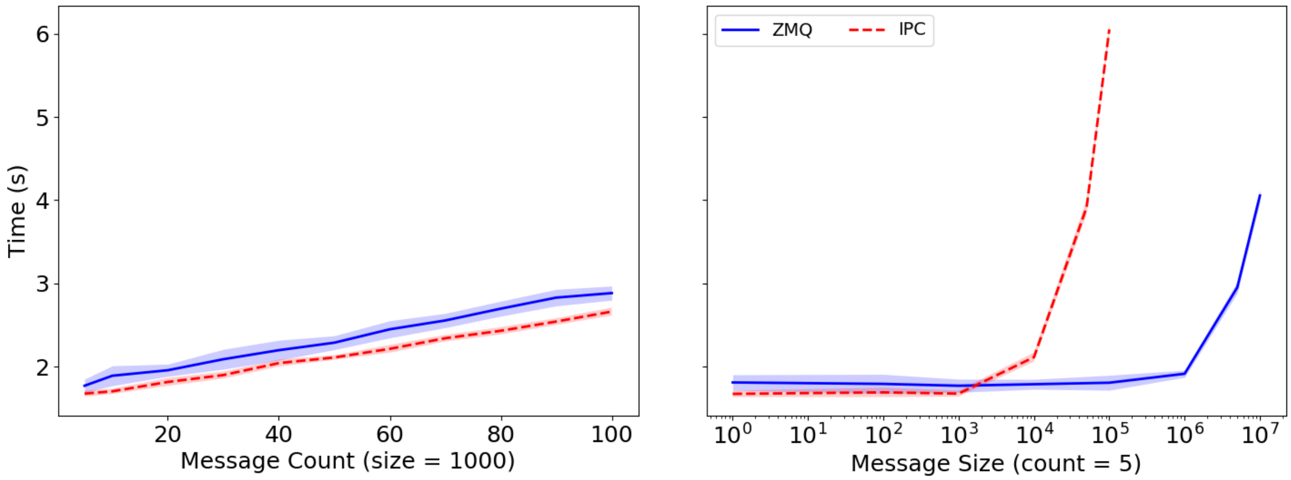

Figure 2: Comparison of communication time scaling across communication methods on Linux. Lines show the average run time for each test with the standard deviation shown in the shaded region. Left: Scaling of execution time with number of 1000 byte messages sent. Right: Scaling of execution time with the size of the 5 messages sent.

Table 1 shows how the two communication mechanisms compare as determined by performing a linear fit to the scaling of execution time with message count from Figure 2. Time per message (slope of Figure 2) is the average amount of time required to send a single message containing 1000 bytes and constrains how quickly messages can be passed between the models. Overhead (intercept of Figure 2) is the amount of execution time that would be required if no messages were passed between the models and includes the time required to set up communication mechanisms, start the models, and clean up the integration network. Although IPC queues are slightly faster than ZMQ TCP sockets in both time per message and overhead for messages smaller than the limit, IPC queues have a much smaller maximum size limit for messages. As discussed in §3.2.5, messages larger than this limit (2048 bytes) must be broken up into multiple messages, compounding the time required to send these messages and making IPC queues a poor choice for sending data over ∼ 10^5 bytes. This limit is much larger for ZMQ sockets (∼ 10^6 bytes). However, for messages less than the limit, both ZMQ and IPC communication has no measurable dependence on message size.

Mechanism	Time per Message (s)	Overhead (s)
IPC	0.010	1.60
ZMQ	0.012	1.73

Table 1: Effect of communication mechanism on performance.

5.2. *Language*

Figure 3 compares the execution times for integrations with communications between models in different combinations of languages and Table 2 reports the time per message and overhead. The runs for Figure 3 and Table 2 were run on the Linux machine from above while those for Figure 4 and Table 3 were run on a 2015 MacBook Pro with a 2.9 GHz Intel Core i5 processor running macOS High Sierra 10.13.4 that had Matlab R2017a installed. In both cases, ZMQ communication was used with Python 2.7.

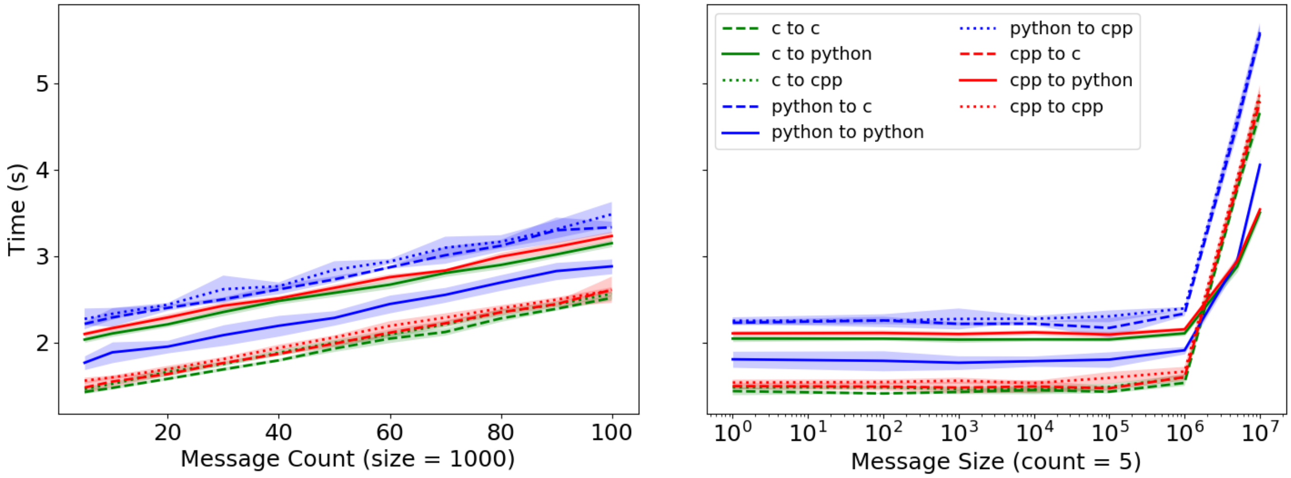

Figure 3: Comparison of communication time scaling between languages on Linux without Matlab. The same fiducial message size (1000 bytes) and count (5) was used as in Figure 2.

Except for Matlab models, the time required per message is not affected by the language of either model for smaller messages. There is a very small increase in the time per message when there is a Python model involved, but this is much lower than the standard deviation resulting from background activity on the test machine and so is in conclusive. Matlab models have a much higher time per message than any other language. This results from the way the Matlab interface was implemented, by calling the Python interface. The time required per message is largest when the receiving model is written in Matlab because the receiving models both receives messages from the other model and send output to a file, resulting in twice the number of calls to the wrapped interface. The performance of the Matlab interface may be improved in the future by either implementing it in Matlab directly or by wrapping the C routines.

Language also plays a role for messages larger than the maximum (2^{20} bytes), when messages must be broken into smaller pieces. In particular, integrations that include a C or C++ receiving model scale more strongly than for those that have a Python receiving model because, in addition to the input/output connection drivers, the Python

Source	Destination	Time per Message (s)	Overhead (s)
C	C	0.011	1.36
C	C++	0.012	1.42
C	Python	0.012	1.99
C++	C	0.012	1.42
C++	C++	0.011	1.49
C++	Python	0.012	2.05
Python	C	0.012	2.15
Python	C++	0.012	2.21
Python	Python	0.012	1.73

Table 2: Effect of model language on performance (Linux).

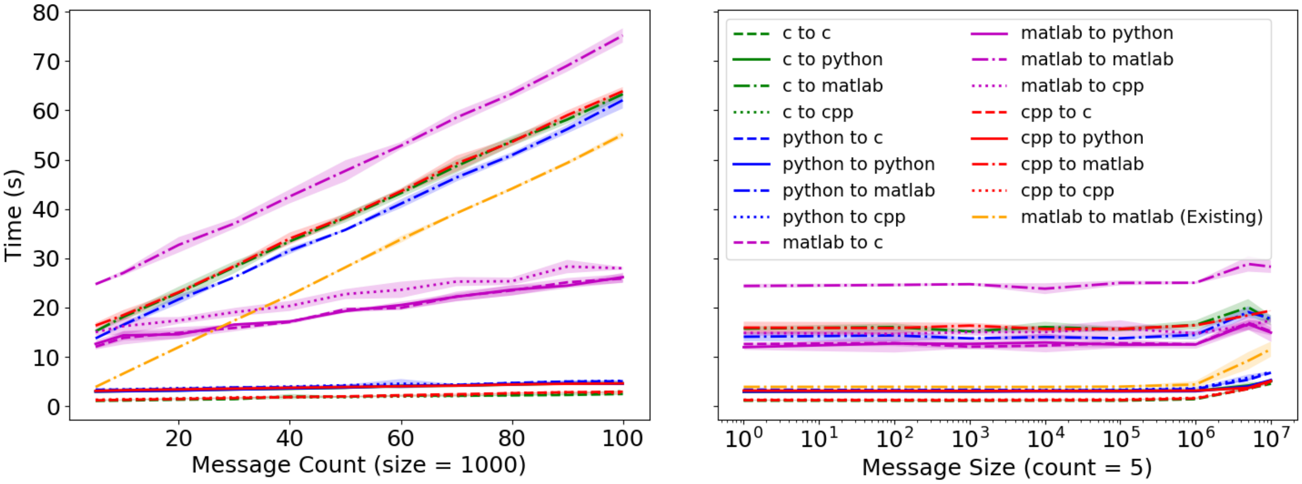

Figure 4: Comparison of communication time scaling between languages on Mac OS X including Matlab.

Source	Destination	Time per Message (s)	Overhead (s)
C	C	0.015	1.11
C	C++	0.016	1.15
C	Python	0.019	2.86
C	Matlab	0.505	12.98
C++	C	0.019	1.10
C++	C++	0.016	1.25
C++	Python	0.016	3.10
C++	Matlab	0.505	13.45
Python	C	0.019	3.17
Python	C++	0.019	3.26
Python	Python	0.019	2.86
Python	Matlab	0.500	11.31
Matlab	C	0.144	11.82
Matlab	C++	0.142	14.79
Matlab	Python	0.139	12.23
Matlab	Matlab	0.526	21.71
Matlab (started)	Matlab (started)	0.537	1.25

Table 3: Effect of model language on performance (Mac OS X) with Matlab.

interface itself is asynchronous while the C/C++ interface is not. This means that while Python receiving models have concurrent operations to receive, backlog, and confirm messages, C/C++ models will block on each receive until the input connection driver completes the confirmation handshake. In addition, because C/C++ models are not continuously receiving and backlogging messages, the channel will become saturated. Once it reaches its maximum, the input connection driver can no longer pass along messages and must wait for the model to receive the next message. One remedy that is being explored for future releases, is to make the C/C++ interface asynchronous as well. However, this is a low priority as only models with very rapid and high volume input (such as the test models) are affected.

The largest difference between the different test cases is in in the overhead. Every model language entails some overhead. For interpreted languages (e.g. Python and Matlab), the majority of the overhead comes from starting the interpreter. For Matlab, this is particularly time consuming (> 10 s). Although this is only a one-time cost required at the start of an integration, `cis_interface` does offer ways to alleviate this if multiple runs are necessary by starting a Matlab engine prior to the first integration that can be used by subsequent runs. When Matlab is started in advance (the dashed-dotted orange line in Figure 4), the overhead for Matlab models drops to 1.25 s, less than Python (2.86 s) but slightly more than C (1.11 s).

For compiled languages (C and C++), the overhead comes from compiling the source code which is done in serial. The time required for compilation will depend on the compiler used and the complexity of the model code. For command line compiled models, `cis_interface` forces the model to be recompiled every time to ensure any changes to the source code are propagated. However, for models that use Make and CMake, overhead due to compile time can be reduced by using existing builds.

Interestingly, the recorded overheads were not symmetric or additive. For example, the Python-to-C integrations required more overhead than the C-to-Python integrations. This is because the models are started in parallel, but do not require the same amount of time to start and being execution. C models, while requiring compilation, start much faster than Python models. In a Python-to-C integration, the C model must wait for the Python model to finish its much slower startup and begin sending message, while in a C-to-Python integration, the C model can immediately begin sending messages that will be received by the Python model once it finishes starting up. In addition, the Python-to-Python integrations required less time than any of the other integrations that included Python because, although Python models take longer to start up, the start up is done in parallel while the compilation for the C and C++ models is done in serial. This source of overhead could be ameliorated by compiling models in parallel or in advanced, but is usually not significant enough to effect overall performance.

5.3. *Python Version*

Figure 5 and Table 4 compare the execution times when using different versions of Python. The performance tests for both Python 2.7 and Python 3.5 were performed on the Linux machine from above using ZMQ communication and two different integration (Python-to-Python and C-to-C). There is ~ 1 s additional overhead with Python 3.5 compared to Python 2.7 due to the increased startup time for the Python 3.5 interpreter. This overhead is only present for integrations that include Python models. For comparison, there is no difference in the time per message and overhead between the different Python versions for the C-to-C integration (dashed lines in Figure 5). There is a slight difference in time per message between the two versions for all models that shows up in the scaling of execution time with message size for large messages ($> 10^6$ bytes). Differences at large message sizes between the two versions arrises from the use of Python classes to transport messages between models and is more pronounced in integrations with Python models due to the increased amount of Python code.

Python Version	Language	Time per Message (s)	Overhead (s)
2.7	Python	0.012	1.73
2.7	C	0.011	1.36
3.5	Python	0.013	2.62
3.5	C	0.011	1.36

Table 4: Effect of Python version on performance.

5.4. *Operating System*

Figure 6 and Table 5 compare the execution times on different operating systems. The performance tests reported for Linux were run on the machine from above. The tests for Mac OS X were run on a 2015 MacBook Pro with a

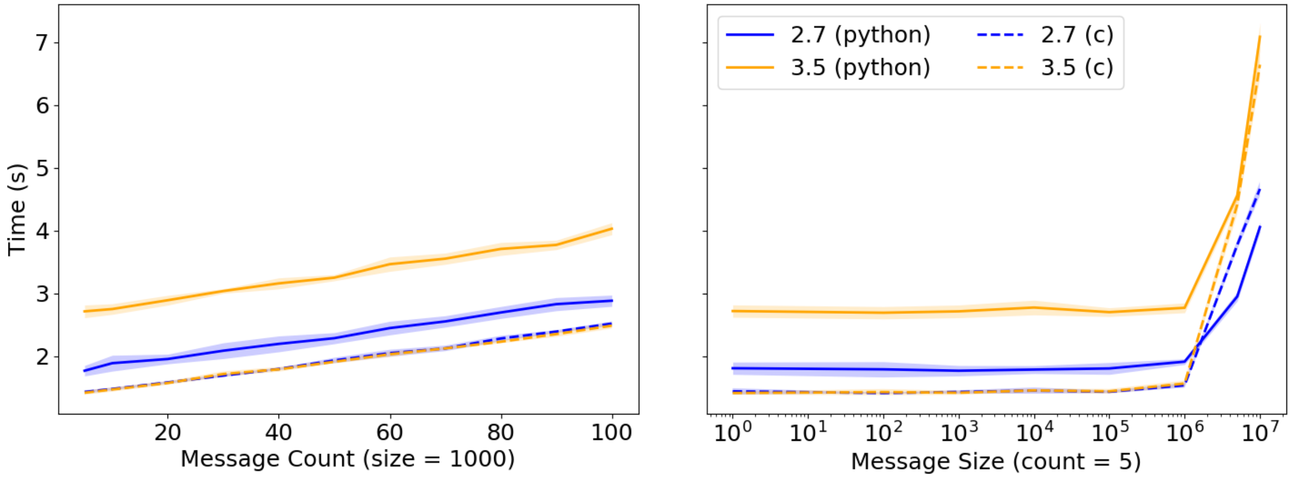

Figure 5: Comparison of communication time scaling between Python versions on Linux. The solid lines are for the Python-to-Python integrations and the dotted lines are for the C-to-C integration.

2.9 GHz Intel Core i5 processor running macOS High Sierra 10.13.4. The tests for Windows were run on the same machine from a dual boot of Windows 10. All platform dependent performance tests use ZMQ communication and the Python-to-Python integration. Both the time per message and overhead changed between operating systems with

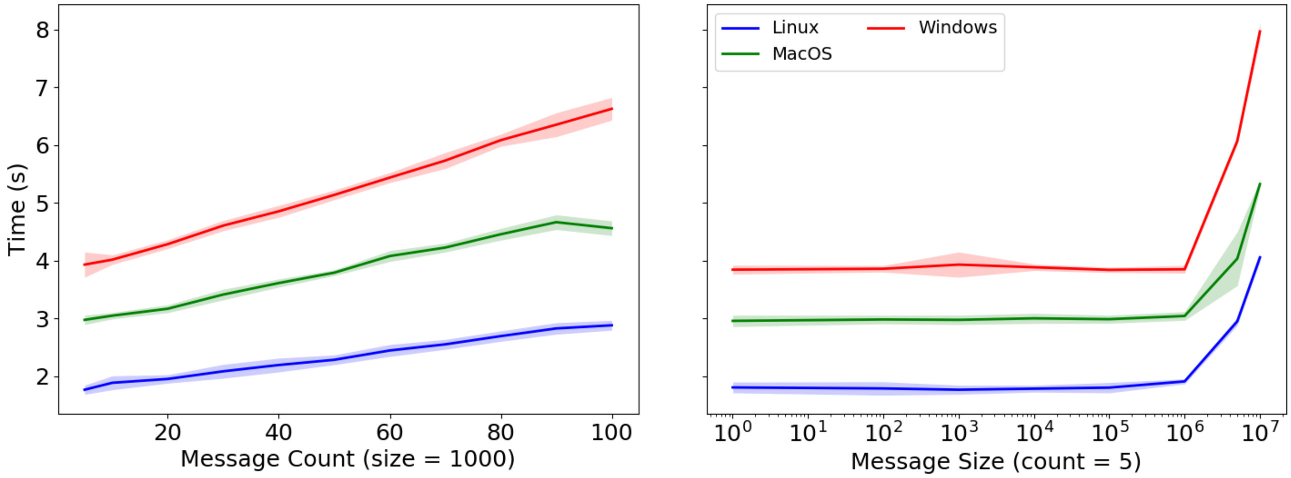

Figure 6: Comparison of communication time scaling between operating systems.

Operating System	Time per Message (s)	Overhead (s)
Linux	0.012	1.73
Mac OS X	0.019	2.86
Windows	0.029	3.73

Table 5: Effect of operating system on performance.

the Linux machine performing the most consistently and fastest and Windows offering the slowest speeds. Because the

Linux test machine was very different from the test machine for both Mac OS X and Windows, it cannot be determined that operating system alone contributed to the observed differences. The same can be said of comparing the Mac OS X and Windows tests given that operating systems are often optimized for their intended hardware. However, the tests do show that cis_interface performed consistently on each of the supported operating systems.

5.5. *Speedup from Parallelism*

While efforts have been made to limit the overhead introduced by cis_interface in the startup and communication steps of the integration, cis_interface cannot outperform a direct integration of two models if they are tightly coupled and must run sequentially. For example, consider an alternate version of the shoot model from §4 where the shoot model itself calculates the current root mass, sends the root mass to the root model, and then waits for the response. While the root model is performing the calculation, the shoot model will be idle, waiting for the response and, while the shoot model is calculating the root mass or handling the response, the root model will be idle, waiting for the next input. As a result, such a system will essentially execute in serial, incurring the overhead from using cis_interface to run the models and coordinate communication without the advantage of parallel execution. Because native language variable passing will always be faster, a direct integration of the two models by translating one into the language of the other will always be faster than the cis_interface integration in such cases. However, if the two models are not as tightly coupled and have calculations that can occur simultaneously, cis_interface can offer a significant speedup via parallelism.

Speedup, in the case of parallelism, is defined as the ratio of the serial execution time to the parallel execution time.

$$S = \frac{T_{serial}}{T_{parallel}} \tag{3}$$

For the example presented above, the speedup offered by cis_interface through parallelism can be parameterized in terms of the durations of calculations for each model (designated model A & B for brevity) performed in the different phases of the process (See Figure 7). During the initial phase ($t_{init,A}$ & $t_{init,B}$), both models perform setup tasks before immediately entering the loop. On entering the loop at Phase 1 ($t_{1,A}$ & $t_{1,B}$), both models perform calculations in preparation for the first communication where Model A sends a message to Model B. In Phase 2 ($t_{2,A}$ & $t_{2,B}$), Model B processes the message it received and Model A performs actions in preparation for receiving a response during the second communication. In Phase 3 ($t_{3,A}$ & $t_{3,B}$), Model A analyzes the response and Model B performs any additional calculations needed before the end of the current loop. Following completion of Phase 3 tasks, the models will immediately either enter another loop or enter the final phase ($t_{final,A}$ & $t_{final,B}$) to complete any necessary cleanup and teardown actions. If the models in the toy example were run in serial (without cis_interface), the total execution time would be

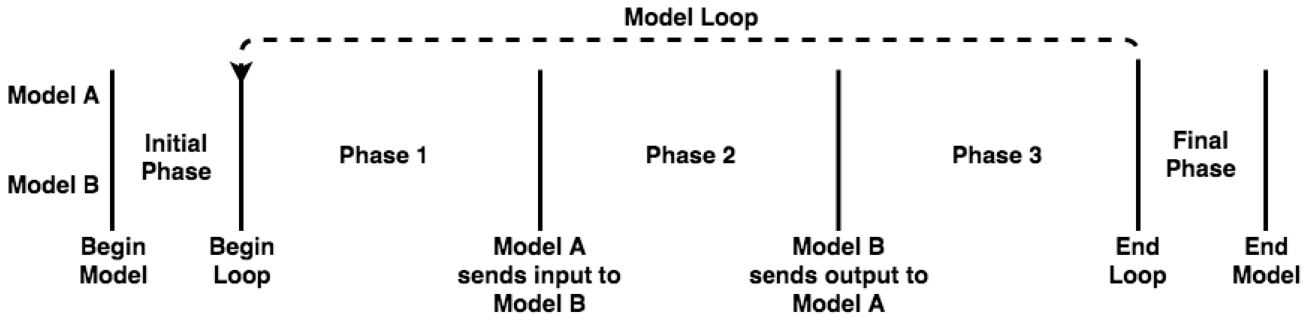

Figure 7: Diagram of phases in parallel execution of models A (top) and B(bottom). Each model performs some setup actions in the initial phase, enters a loop that contains two communication (first from model A to model B, then from model B to model A), and then performs some teardown actions in the final phase.

$$T_{serial} = t_{init,A} + t_{init,B} + N_{loop}\left[\sum_{i=1}^{3}(t_{i,A} + t_{i,B})\right] + t_{final,A} + t_{final,B}, \tag{4}$$

where N_{loop} is the number of loops executed and $t_{x,A}$ and $t_{x,B}$ are the execution times of each phase of the process for models A and B respectively. In order to calculate the parallel execution time, we must consider the effect of asynchronous communication. For every asynchronous call, the receiving model is limited by the sending model, but the sending model is not limited by the receiving model. For a single asynchronous communication between two parallel processes, the execution time will be

$$T_{async} = \max\left\{\left[t_{before,send} + t_{after,send}\right], \left[\max(t_{before,send}, t_{before,recv}) + t_{after,recv}\right]\right\}, \tag{5}$$

where $t_{before,send}$ and $t_{after,send}$ are the time the sending model spends in the phases before and after sending the message respectively and $t_{before,recv}$ and $t_{after,recv}$ are the same times, but for the receiving model. The parallel execution time for the example which includes a loop around two asynchronous communications is then a recursive relationship where the execution time for one loop depends upon the relative execution times of the two models for the previous loop. This relationship can be written as a recursive formula

$$T_{parallel} = T_{overhead} + \max(T_{A,N_{loop}}, T_{B,N_{loop}}), \tag{6}$$

$$\begin{cases} T_{A,0} = t_{init,A} + t_{final,A} \\ T_{A,n} = \max\left[(T_{A,n-1} + t_{comm} + t_{1,A} + t_{2,A} - t_{final,A}), (T_{B,n} - t_{3,B} - t_{final,B})\right] + t_{3,A} + t_{final,A} \\ T_{B,0} = t_{init,B} + t_{final,B} \\ T_{B,n} = \max\left[(T_{B,n-1} + t_{comm} + t_{1,B} - t_{final,B}), (T_{A,n-1} + t_{1,A} - t_{final,A})\right] + t_{2,B} + t_{3,B} + t_{final,B} \end{cases} \tag{7}$$

where $T_{overhead}$ is the overhead incurred by using cis_interface to run the models and setup the communication network, t_{comm} is the time required to complete the exchanges via each cis_interface asynchronous communication, n is the number of loops, and $T_{A,n}$ and $T_{B,n}$ are the times required by each model to complete their individual task in parallel for the given number of loops. While $T_{overhead}$ and t_{comm} can be determined from the previous results sections for a given set of languages, the other variables are entirely dependent on the speed of the code occupying each phase. This relationship can also be written more intuitively as an algorithm (See Algorithm 9)

Algorithm 9 Parallel Execution Time

```
$T_A \leftarrow t_{init,A}$
$T_B \leftarrow t_{init,B}$
$iloop \leftarrow 0$
while $iloop <= N_{loop}$ do
    $T_A += t_{1,A}$
    $T_B += t_{1,B}$
    $T_B = \max(T_A, T_B + t_{comm})$          ▷ Model B blocking on receive.
    $T_A += t_{2,A}$
    $T_B += t_{2,B}$
    $T_A = \max(T_A + t_{comm}, T_B)$          ▷ Model A blocking on receive.
    $T_A += t_{3,A}$
    $T_B += t_{3,B}$
    $iloop += 1$
end while
$T_A += t_{final,A}$
$T_B += t_{final,B}$
$T_{parallel} = \max(T_A, T_B)$
```

If the models are completely dependent on one another such that neither has any calculations to perform while the other is working, the situation would be that shown in Figure 8. Assuming that $t_{init,B} < (t_{init,A} + t_{1,A})$, this integration would have a parallel execution time and speedup of

$$T_{parallel} = T_{overhead} + N_{loop}(t_{comm} + t_{1,A} + t_{2,B} + t_{3,A}) + \max\left[(t_{init,A} + t_{final,A}), (t_{init,B} + t_{final,B})\right] \tag{8}$$

$$S = \frac{t_{init,A} + t_{init,B} + N_{loop}(t_{1,A} + t_{2,B} + t_{3,A}) + t_{final,A} + t_{final,B}}{T_{overhead} + N_{loop}(t_{comm} + t_{1,A} + t_{2,B} + t_{3,A}) + \max\left[(t_{init,A} + t_{final,A}), (t_{init,B} + t_{final,B})\right]} \tag{9}$$

If the initial and final phases require an insignificant amount of time in comparison with the time spent in the loop, this will approach a speedup of 1, indicating that there was no change in the execution time. However, due to the required overhead ($T_{overhead}$) and communication times (t_{comm}) incurred by a cis_interface integration, it is more likely that the speedup will be below 1 in such a case unless the calculations have a long enough duration to render these costs inconsequential. For example, if we consider two Python models ($t_{comm} = 0.02\,\mathrm{s/msg}$ and $T_{overhead} = 2.86\,\mathrm{s}$) where

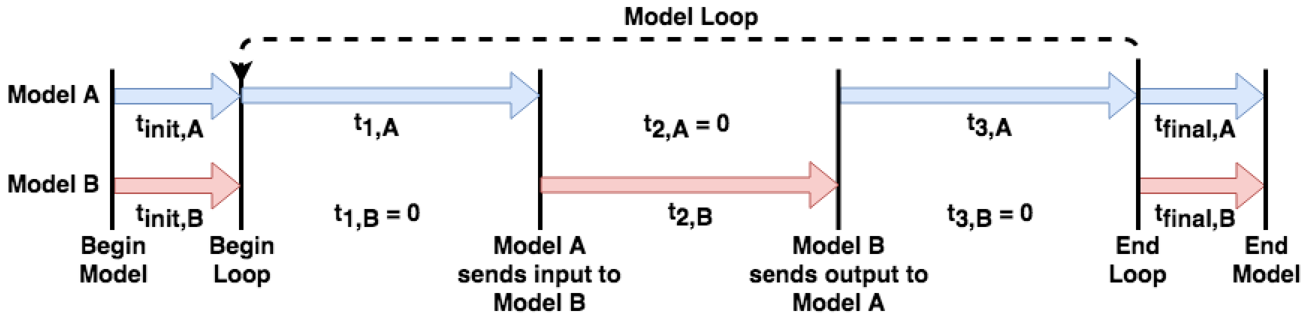

Figure 8: Diagram of parallelism for a fully coupled set of models. The shaded arrows represent active calculation by models A (top) and B(bottom). Absence of an arrow indicates idle time where the model process is not performing any calculations.

$t_{1,A}$, $t_{2,B}$, and $t_{3,A}$ are all 1 s with initial and final phases that are 0.1 s for both models, the speedup will be 0.92 for 10 loops and 0.99 for 100 loops.

If the models are perfectly matched in that each spends exactly the same amount of time in each phase ($t_{init,A} = t_{init,B}$, $t_{1,A} = t_{1,B}$, etc.) as in Figure 9, the parallel execution time and speedup of the integration would be

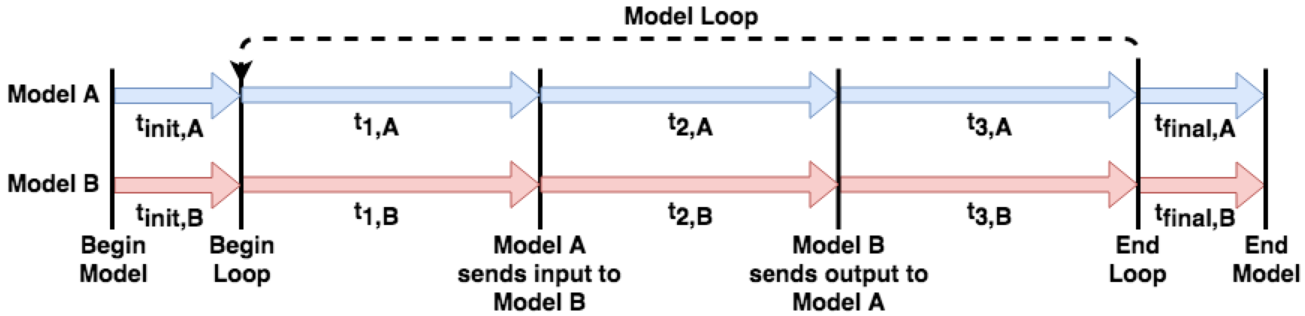

Figure 9: Diagram of parallelism for a set of perfectly matched models.

$$T_{parallel} = T_{overhead} + t_{init,A} + N_{loop}\left(t_{comm} + \sum_{i=1}^{3} t_{i,A}\right) + t_{final,A} \tag{10}$$

$$S = \frac{2\left[t_{init,A} + N_{loop}\left(\sum_{i=1}^{3} t_{i,A}\right) + t_{final,A}\right]}{T_{overhead} + t_{init,A} + N_{loop}\left(t_{comm} + \sum_{i=1}^{3} t_{i,A}\right) + t_{final,A}}. \tag{11}$$

If the runtime is sufficiently long such that contributions from $T_{overhead}$ and t_{comm} become insignificant, the speedup will approach 2, the maximum for two processes. For example, if we consider two Python models ($t_{comm} = 0.02$ s/msg and $T_{overhead} = 2.86$ s) where both models have Phases 1-3 of 1 s in duration and initial and final phases of 0.1 s in duration, the speedup will be 1.82 for 10 loops and 1.97 for 100 loops.

However, the situations shows in Figures 8 and 9 are rarely the case for actual models. Figure 10 shows a more realistic case in which the models have some overlapping operations, but are not perfectly aligned. Speedups in such cases would be between 1 and 2. For example given the parameters shown in Table 6 for two Python models ($t_{comm} = 0.02$ s/msg and $T_{overhead} = 2.86$ s), the speedup will be 1.38 for 10 loops and 1.49 for 100 loops.

For more complex integration involving more than 2 models or more than 2 communications per loop, speedup calculations will be more complex. In general, the maximum possible speedup will be greater for integrations of models that are more independent, but wrappers can be adjusted to deliberately perform tasks during periods where a model would be waiting for a message in order to get greater performance boosts.

6. SUMMARY

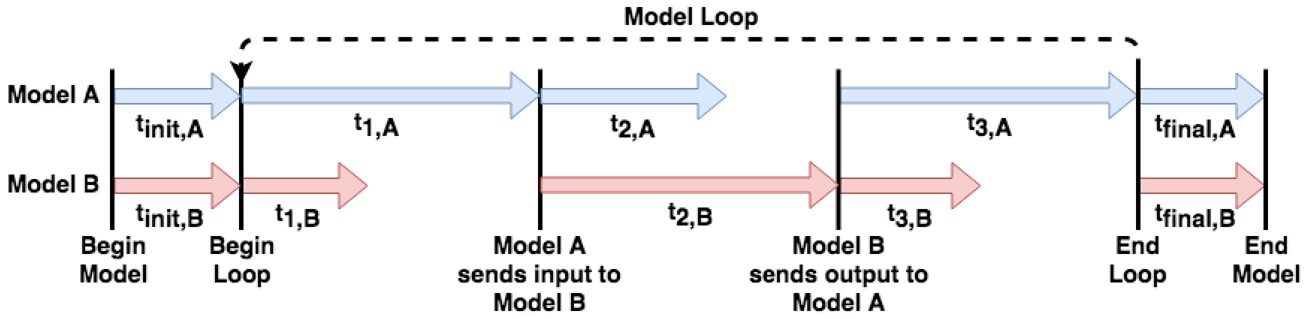

Figure 10: Diagram of partial parallelism for a set of imperfectly matched models.

Table 6: Speedup calculation example parameters for a set of imperfectly matched models.

Phase	Duration in Model A (s)	Duration in Model B (s)
initial	0.1	0.2
1	1.0	0.3
2	1.2	2.0
3	0.1	0.05
final	0.1	0.1

6.1. *Current Status*

cis_interface is an open-source Python package for connecting computational models across programming languages and scales to form integration networks. Models are run in parallel with asynchronous message passing handled under-the-hood via threading and one of three communication mechanisms. cis_interface works on Linux, Mac OS X, and Windows operating systems with Python 2.7, 3.4, 3.5, 3.6 and 3.7. cis_interface currently supports running models written in Python, C, C++, and Matlab with additional support for compiling models C/C++ using Make (Stallman et al. 2004) or CMake (Martin & Hoffman 2006) and running models written in the LPy (Boudon et al. 2012) DSL.

Integration via cis_interface requires minimal modification of the existing model source code yet greatly increases the model's potential by allowing it to be reused in combination with other models. cis_interface has already been used to integrate several plant models within the Crops in Silico organization (Marshall-Colon et al. 2017) with ongoing work to add additional models to the integration network. Figure 11 shows the progress so far. Each square is a model, color indicates the programming language that the model is written in, and lines between the models represent connections where information is exchanged. Solid lines are connections that have already been implemented, dashed lines are connections currently being implemented, and dotted lines are connections planned for the future. When complete, the integration network will include 9 models of molecular, genetic, cellular, organ, and plant level processes in 4 different programming languages, produced by > 10 researchers at 4 different institutions.

Although not yet complete, partial integrations from this network are already yielding new insights. For example, by integrating a metabolic flux model with gene expression data from soybean (*Glycine max*) plants grown under ambient (380 μmol/mol) and elevated (550 μmol/mol) carbon dioxide (CO_2), Kannan et al. (2018) simulated the increased carbon assimilation rate observed in field studies (Bernacchi et al. 2005). The integrated modeling also identified gene candidates predicted to regulate enzymes involved in the light and dark reactions of photosynthesis under elevated CO_2.

6.2. *Use Cases & Limitations*

In addition to allowing scientists to reuse existing models in new integrations, cis_interface has the potential to significantly increase a model's accessibility. cis_interface could be used a method of running models without knowledge of the model's implementation. At publication, model code could be released along with a wrapper and YAML specification file for use with cis_interface. These additional materials would be particularly powerful when considered alongside the interactive features being developed for use with cis_interface (see §6.3.7 and §6.3.8). Add discussion of limitations

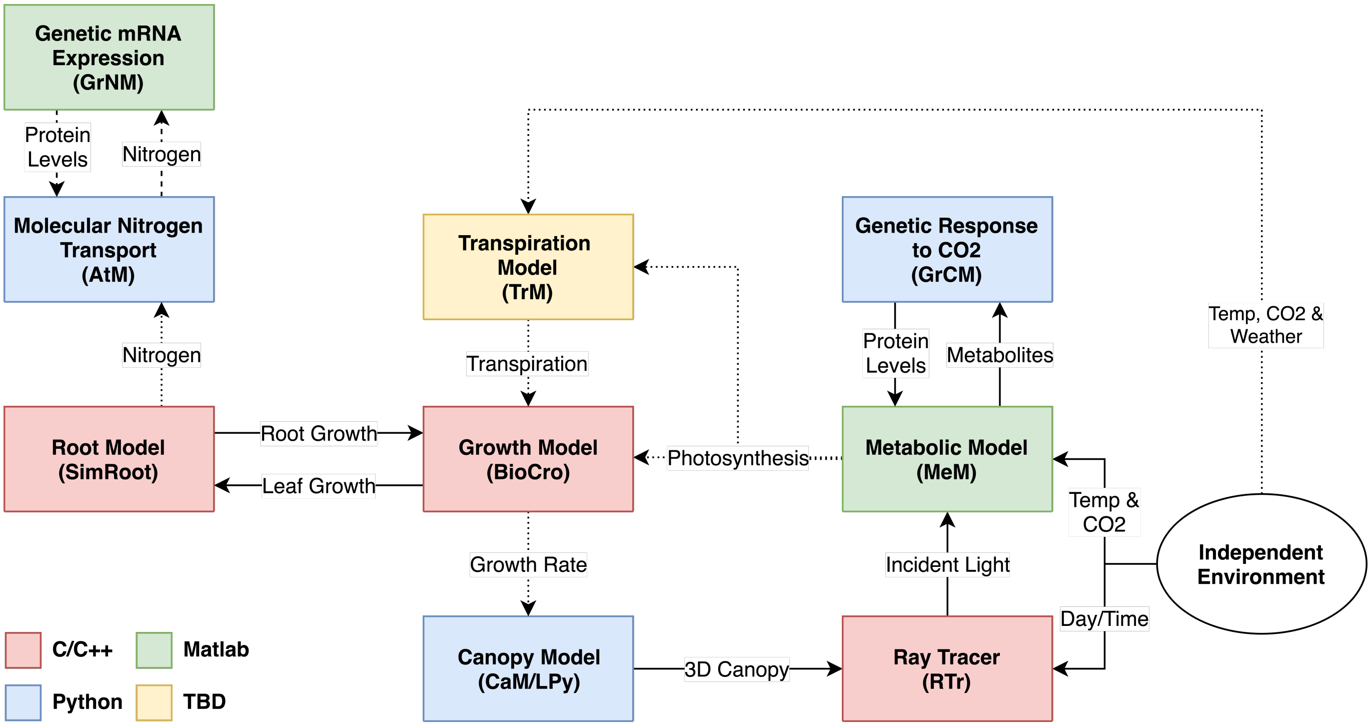

Figure 11: Progress towards full Crops in Silico integration network. Each square represents a model and the color of the square indicates the language in which the model is written. Arrows between models represent connections being made between the models using cis_interface. Solid lines are connections that have already been made, dashed lines are in-progress connections, and dotted lines are planned connections.

6.3. *Current/Future Improvements*

cis_interface is being actively developed to expand the number of models that can be used in integration networks and the complexity of integration networks that can be executed. Several improvements are already in progress/planned for cis_interface.

6.3.1. *Language Support*

The biggest barrier to running new models is language support. Plant models are written in many different languages. While the current language support covers many of the most popular languages among plant biologists, additional languages must be added to unlock the full set of potential integrations. Surveys of the plant modeling community have helped identify the core languages that models have been written in. Based on these results, we plan to expand cis_interface to support models written in R, Fortran, Java, and the SBML (Hucka et al. 2003) DSL.

We also plan to add support for running Matlab models using Octave (Eaton 2002). Matlab models require a Matlab license to run. Given that plant modelers do not all use Matlab, it cannot be assumed that everyone will have access to a Matlab license. Octave is open source and provides much of the same functionality as Matlab and can run many Matlab codes. Support for Octave will improve modelers ability to collaborate without worrying about access to a Matlab license.

6.3.2. *Distributed Systems*

High performance computing (HPC) and cloud compute resources are powerful tools in the current computing ecosystem that could be used for running complex integration networks. To this end, we plan to expand cis_interface support for running integration networks on distributed compute resources. cis_interface already uses communication tools that can be adapted for use in a distributed pattern. We will leverage tools like the libsubmit package from the Parsl project (Babuji et al. 2018) for automating the submission process to HPC and compute resources. In addition, we will add tools for running models as a service and using RMQ to permit access to these models within integration networks.

6.3.3. *Control Flow*

Currently, modelers must explicitly specify how a model should process input within the model code including things like which input variables are static and received once versus which variables change and should be looped over. We plan on adding options to cis_interface for dynamically generating model code based a user provided function call and list of static/variable input channels. This will allow better reusability of model code such as during parameter studies. Control flow will be part of the cis_interface 1.0 release.

6.3.4. *Symbolic ODE Models*

Many biological models can be posed as a series of coupled ordinary differential equations (ODEs, Wang et al. 2015). The equations describing such models can be easily formulated into symbolic representations (e.g. SymPy Meurer et al. 2017) with well defined inputs and outputs that can be solved by existing tools designed specifically for that task (e.g. scipy.integrate.ode Jones et al. ????; Hairer et al. 1993). Using these tools, we will add features to cis_interface that allow ODE models to be constructed directly from symbolic representation provided in YAML specification files.

6.3.5. *Data Aggregation*

Much of the original version of cis_interface centered around the use of tabular input data. While tabular data is used heavily by plant models, it presents several barriers for constructing integration networks. Tabular input data only works when one model outputs the same columns that are expected as input by another model. However, this is unlikely to be the case if two models are developed independently. Future improvements to cis_interface will allow tabular input to models to be composed by aggregating output data from more than one model and/or file. Data aggregation will be part of the cis_interface 1.0 release.

6.3.6. *JSON Data Type Specification*

While cis_interface supports serialization of several data types/formats, we would like to make serialization as flexible as possible. To this end, cis_interface will be adapted to read JSON schema (jso 2018) for user defined types that can be used to automatically create the appropriate data structures alongside methods for parsing and serializing them. JSON serialization will be part of the cis_interface 1.0 release.

6.3.7. *Graphical User Interface*

In an effort to make cis_interface as user friendly as possible, we have also begun work on a graphical user interface (GUI) for entering model information, composing integration networks from an existing palette of models, and displaying basic output from an integration run. The current prototype of the GUI can only handle model ingestion and network composition; model execution must be done locally. However, our ultimate goal with the GUI is to provide a service where users can register their model, compose and run integration networks on dedicated cloud compute resources, and view output in real time all from the web browser. This GUI for model integration and execution will also greatly increase the accessibility of models as standalone scientific products. Once complete, users will be able to use the GUI to graphically run models and adjust parameters without any knowledge of the underlying algorithms, programming language, and/or data formats. Such a workflow could be a powerful tool for both the peer review process and collaboration.

6.3.8. *Models as Functions*

In addition to a dedicated GUI, tools are also being developed for using cis_interface to call integrated models as Python functions, allowing users to interactively explore and visualize a model. This feature is particularly powerful when combined with Jupyter notebooks (Kluyver et al. 2016), which allow code to be presented alongside text and/or images describing what the code does. Model developers could publish a Jupyter notebook exposing their model as a function via cis_interface that can be used by those unfamiliar with modeling or the model's particular programming language to explore the model's behavior. This workflow would be particularly well suited to examining models during the peer review process and allowing on-the-fly parameter adjustment/estimation by experimentalists. Widgets can make these notebooks even more user friendly, allowing code to be replaced with graphical interfaces (sliders, toggles, etc.) for modifying model parameters and allowing even those unfamiliar with Python to explore the model. In addition, tools like Binder, BinderHub, and mybinder.org (Jupyter et al. 2018) make it possible for those interested in a model to launch such notebooks stored in public repositories from their browser in custom environments deployed on cloud resources without the need to install cis_interface or any of the model code on their machine. They can then interact with the model using either widgets or the full set of Python libraries. Model execution as Python functions will be part of the cis_interface 1.0 release.

CODE

The cis_interface package is available publicly on Github and can be installed using pip or conda. To ensure code health, cis_interface boasts 100% code coverage with automated testing on Linux, Mac OS X, and Windows for Python version 2.7, 3.4, 3.5, 3.6, and 3.7 via continuous integration with TravisCI (tra 2018) and Appveyor (app 2018). Full documentation for cis_interface can be found here including step-by-step directions from the tutorial conducted during the 2018 Crops in Silico Hackathon.

ACKNOWLEDGMENTS

This work was supported by funding from the Foundation for Food and Agriculture Research (FFAR) through Grant 515760 and Cost Share C2252 with the Institute for Sustainability, Energy, and Environment (iSEE) and the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), as well as the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4561 to Matthew Turk. The authors would like to thank Mike Lambert and Craig Willis for their ongoing work on a graphical user interface for visually composing and running integration networks for cis_interface as well as David Raila for his work on the original integration code that cis_interface was based on.

REFERENCES

1994, B1. Object Files (.obj), Tech. rep.

2006, PyYAML, https://pyyaml.org/

2007, RabbitMQ, https://www.rabbitmq.com/

2008, pandas: Python Data Analysis Library, https://pandas.pydata.org/

2011, JS-YAML - YAML 1.2 parser / writer for JavaScript, https://github.com/nodeca/js-yaml

2018, AppVeyor: Continuous Integration solution for Windows and Linux, https://www.appveyor.com/

2018, Celery, https://github.com/celery/celery

2018, JSON Schema — The home of JSON Schema, https://json-schema.org/

2018, Luigi, https://github.com/spotify/luigi

2018, Travis CI - Test and Deploy Your Code with Confidence, https://travis-ci.org/

Akgul, F. 2013, ZeroMQ : use ZeroMQ and learn how to apply different message patterns (Packt Publishing)

Babuji, Y., Chard, K., Foster, I., et al. 2018, in 10th Int. Work. Sci. Gateways

Behnel, S., Bradshaw, R., Citro, C., et al. 2011, Comput. Sci. Eng., 13, 31

Ben-Kiki, O., Evans, C., & döt Net, I. 2009, YAML Ain't Markup Language (YAML$^{\text{TM}}$) Version 1.2 YAML Ain't Markup Language (YAML$^{\text{TM}}$) Version 1.2 3 rd Edition, Patched at 2009-10-01, Tech. rep.

Bernacchi, C. J., Morgan, P. B., Ort, D. R., & Long, S. P. 2005, Planta, 220, 434

Boudon, F., Pradal, C., Cokelaer, T., Prusinkiewicz, P., & Godin, C. 2012, Front. Plant Sci., 3, 76

Coakley, S., Gheorghe, M., Holcombe, M., et al. 2012, in 2012 IEEE 14th Int. Conf. High Perform. Comput. Commun. 2012 IEEE 9th Int. Conf. Embed. Softw. Syst. (IEEE), 538–545

Cuellar, A. A., Lloyd, C. M., Nielsen, P. F., et al. 2003, Simulation, 79, 740

Demir, E., Cary, M. P., Paley, S., et al. 2010, Nat. Biotechnol., 28, 935

Eaton, J. W. 2002, GNU Octave Manual (Network Theory Limited)

GEO. 2017, MINiML - GEO - NCBI, https://www.ncbi.nlm.nih.gov/geo/info/MINiML.html

Goldbaum, N. J., Zuhone, J. A., Turk, M. J., Kowalik, K., & Rosen, A. L. 2018, J. Open Source Softw., 3, 809

Goudriaan, J., & Laar, H. V. 1994, Modelling potential crop growth processes (Dordrecht: Kluwer Academic Publishers), 238

Hairer, E. E., Nørsett, S. P. S. P., & Wanner, G. 1993, Solving ordinary differential equations. I, Nonstiff problems, 528

Hall, A. J., & Minchin, P. E. H. 2013, Plant. Cell Environ., 36, 2150

Houdini FX. 2018, L-System geometry node, http://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html

Hucka, M., Finney, A., Sauro, H. M., et al. 2003, Bioinformatics, 19, 524

Humphries, S., & Long, S. 1995, Bioinformatics, 11, 361

Jones, E., Oliphant, T., & Peterson, P. ????, SciPy: Open source scientific tools for Python, http://www.scipy.org/

Jupyter, P., Bussonnier, M., Forde, J., et al. 2018, in Proc. 17th Python Sci. Conf., 113–120

Kannan, K., Wang, Y., Lang, M., et al. 2018, in prep.

Kappas, M., Degener, J., & Bauboeck, R. 2013, Am. Geophys. Union, Fall Meet. 2013, Abstr. id. GC33A-1088

Kluyver, T., Ragan-Kelley, B., Pérez, F., et al. 2016, in Position. Power Acad. Publ. Play. Agents Agendas, 87–90

Marshall-Colon, A., Long, S. P., Allen, D. K., et al. 2017, Front. Plant Sci., 8, 786

Martin, K., & Hoffman, B. 2006, Mastering CMake: A Cross-Platform Build System, 3rd edn. (Kitware)

Merks, R. M. H., Guravage, M., Inzé, D., & Beemster, G. T. S. 2011, Plant Physiol., 155, 656

Meurer, A., Smith, C. P., Paprocki, M., et al. 2017, PeerJ Comput. Sci., 3, e103

Postma, J. A., Kuppe, C., Owen, M. R., et al. 2017, New Phytol., 215, 1274

Pradal, C., Boudon, F., Nouguier, C., Chopard, J., & Godin, C. 2009, Graph. Models, 71, 1

Pradal, C., Fournier, C., Valduriez, P., & Cohen-Boulakia, S. 2015, in Proc. 27th Int. Conf. Sci. Stat. Database Manag. - SSDBM '15 (New York, New York, USA: ACM Press), 1–6

Rusling, D. A. 1999, The Linux Kernel, 0th edn.

Sharkey, T. D. 2016, Plant. Cell Environ., 39, 1161

Simonov, K. 2006, LibYAML - A YAML parser and emitter library., https://pyyaml.org/wiki/LibYAML

Song, Q., Zhang, G., & Zhu, X.-G. 2013, Funct. Plant Biol., 40, 108

Stallman, R., McGrath, R., & Smith, P. D. 2004, GNU make : a program for directed recompilation : GNU make version 3.81 (GNU Press), 184

Stinner, V. 2018, Python perf module, https://perf.readthedocs.io/en/latest/

Turk, G. 1994, The PLY Polygon File Format, Tech. rep., The Board of Trustees of The Leland Stanford Junior University

Wang, D., Jaiswal, D., Lebauer, D. S., et al. 2015, Plant. Cell Environ., 38, 1850

Wang, Y., Long, S. P., & Zhu, X.-G. 2014, Plant Physiol., 164, 2231

Zhu, X.-G., Wang, Y., Ort, D. R., & Long, S. P. 2013, Plant. Cell Environ., 36, 1711

To do:

- Replace links with those referencing new package name
- Forking example
- Transformation example
- Add JSON to documentation
- Add forking to documentation
- Add transformation to documentation
- Add schema to documentation
- Regenerate documentation
- Fix matlab autodocs for interface