`cis_interface`: A Python Package for Integrating Computational Models Across Languages and Scales

Meagan Lang[1]

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 X, 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, with visual tools developed in parallel for composing the integration networks.

1. Introduction

Computational plant biologists have produced a wealth of computational models to describe the biological processes governing plant growth and development, covering scales from atomic 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 can also be combined, or integrated, to probe more complex biological mechanisms. By integrating a metabolic flux model with a genetic regulatory model for carbon uptake, Kannan et al. (2018) was able to identify gene candidates for regulating photosynthesis under elevated CO_2. 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.

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

We present an Open Source Python package, `cis_interface`, for creating and running integration networks by connecting existing computational models written in Python, Matlab, C and C++. 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, §4 presents several tests demonstrating the performance of message passing between integrated models, and §5 summarizes the features of the `cis_interface` package, describes a few use cases, and outlines areas for future development.

2. Background

Computational plant models are usually written with a very specific research question in mind. The resulting programs are highly tuned for one purpose and written specifically for scientists within an isolated group of collaborators. Scientists will generally write programs in the language they are most familiar with and, given that there is no consensus on one programming language as ideal for all scientific applications, the language of computational models will often vary between research groups, if not between members of the groups themselves. The same can be said of data formats which can be highly tailored to the target question and may or may not include metadata like the data types, field names, or units. To integrate two biological models, the following questions must be addressed:

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?

These three questions can be addressed through manual integration on the command line or using scripting (e.g. running model A, converting output from A into input expected by model B, running model B). 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 > 3 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.

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), 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 do nothing for existing models that are not in the DSL or cannot be expressed within the constraints of the DSL. In addition, learning a new DSL, in addition to a programming language, can be daunting for new researchers.

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. 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 them in mind.

`cis_interface` interface is an example of a framework that overcomes these issues by exposing light-weight 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 dosn't block when a message is sent.
- **Flexible.** Provide the same interface to the user, regardless of the communication mechanism being used or the platform the model is being executed on.

`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 §5.2.1).

3.1. Orchestration

`cis_interface` is executed via a command line interface (CLI), `cisrun` with YAML specification files (§3.1.1) as input. Based on the information contained in the YAML 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. YAML Specification

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. 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. 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. Connections between models are specified by references to the input/output variables associated with each model. This format allows users the flexibility to create complex integration networks.

3.1.2. Model Drivers

Model drivers handle model execution, monitoring, and compilation if necessary. 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 there is significant overhead associated with starting the Matlab interpreter (See §4.2), a Matlab shared engine is used to execute the model. To speed up the execution, the shared engine 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 `cis_interface` handle the compilation, including linking against the appropriate `cis_interface` header 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 drivers, lines are added to the recipe in order to allow linking against `cis_interface`.

Once model are running, each model driver uses a thread to monitor the progress of the model and report back model output (e.g. log messages printed to stdout) and any status changes to the master process. If a model issues any errors, the master process will shut down any model processes that are still running. If a model completes without any errors, the master process will cleanup any input/output channels that are no longer required. The master process will only complete once an error is encountered or all model processes have completed.

3.2. Communication

Communication within `cis_interface` was designed to be flexible in terms of the languages, platforms, and data types. To accomplish this, `cis_interface` leverages three tools different tools for communication, System V IPC Queues (§3.2.2), ZeroMQ (§3.2.3), and RabbitMQ (§3.2.4), which are applied as needed. The particular mechanism used by `cis_interface` for communication is hidden from the user, who will always use the same interface in the language of their model (See §3.2.6).

3.2.1. Asynchronous Message Passing

Regardless of the specific communication mechanism, the same asynchronous communication strategy is used. Models do not block on sending messages to output channels; the model is free to continue working on its task while a separate thread waits for the message to be routed and received. Similarly, a thread continuously pings input channels, moving received messages into a buffer queue so that they are ready and waiting for the receiving model when it asks for input. As a result, models in complex integration networks can work in parallel, improving the overall efficiency with which computational resources are used. Figure 1 describes the general flow of messages.

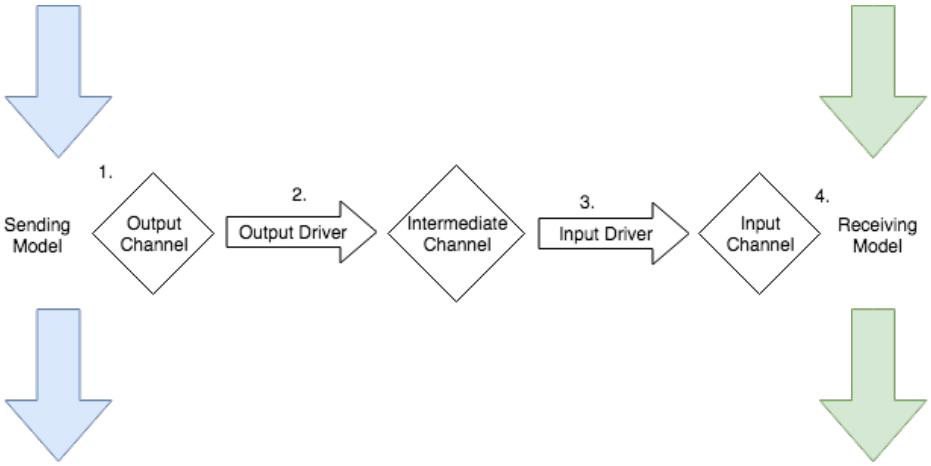

Fig. 1.— Diagram of how messages are passed asynchronously using input/output drivers and an intermediate channel.

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.

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 RabbitMQ 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.

In addition to communication between two models, users can also specify that a model should receive/send input/output from/to a file. This is specified in the integration network YAML and does not impact the way the model will receive/send messages. In this way, users can test their model integration in isolation with input/output form/to files rather than another model.

3.2.2. 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 light weight, fast (See §4.1), and are part of Posix operating systems, they do not work in all situations. Sys V 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 §4.1). These limits make sending large messages relatively inefficient when compared with other communications mechanisms. As a result, Sys V IPC queues are used by `cis_interface` as a fallback on Posix systems if the necessary ZeroMQ libraries have not been installed.

3.2.3. ZeroMQ

The preferred communication mechanism used by `cis_interface` are ZeroMQ sockets (Akgul 2013). ZeroMQ 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. ZeroMQ was adopted by `cis_interface` in order to allow support on Windows and for future target languages (See §5.2) that could not be accomplished using System V IPC queues. In addition, while ZeroMQ allows interprocess communication via IPC queues like System V IPC queues, ZeroMQ 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.

3.2.4. RabbitMQ

While broker-less communication like ZeroMQ 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 2007) that will be used during future development to allow integrations to run on distributed resources or include remote models run as services (§5.2.2). Due to the slower message speed, `cis_interface` does not currently use RabbitMQ for communication unless explicitly specified by the user in their integration network YAML.

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 ZeroMQ and RabbitMQ), while others are very limiting (2048 bytes on Mac OS X for Sys V 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.

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, floats)
- Arrays
- Text-encoded tables (e.g. CSV or tab-delimited)
- Pandas data frames
- PLY
- OBJ

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). 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.

4. Results

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 the source model (in one language) to the destination model (in another language) which then output the messages to file for verification. Performance tests were run using the perf package (Stinner 2018). Each run was repeated 10 times to mitigate fluctuations due to external loads on the test machines.

4.1. 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.

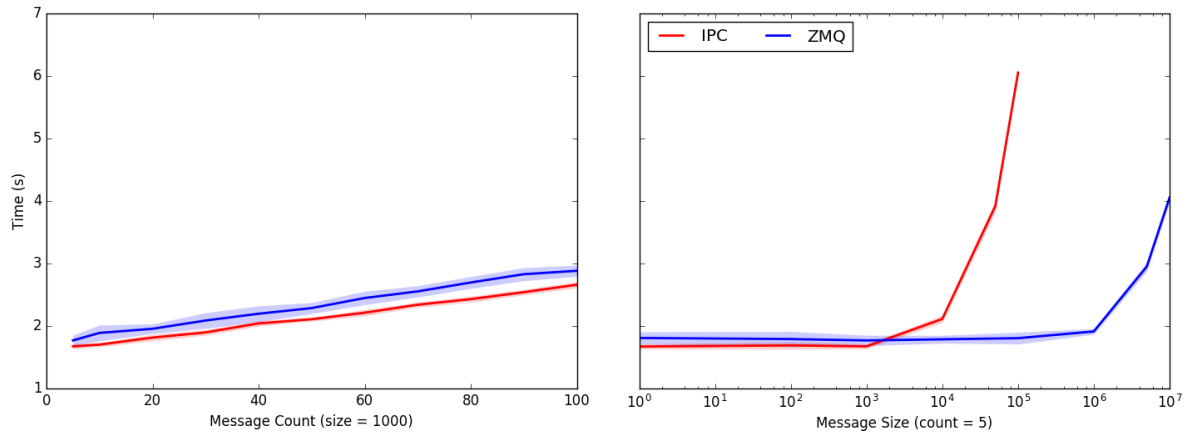

Fig. 2.— Comparison of communication time scaling across communication methods on Linux. Lines show the average run time for each test with the standard deviation shown in the shaded region. Left: Scaling of execution time with number of 1000 byte messages sent. Right: Scaling of execution time with 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 model drivers, and clean up the integration network. Although Sys V IPC queues are slightly faster than ZeroMQ 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 $\sim 10^5$ bytes. This limit is much larger for ZeroMQ sockets ($\sim 10^6$ bytes).

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.

However, for messages less than the limit, both ZMQ and IPC communication has no measurable dependence on message size.

4.2. 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, ZeroMQ communication was used with Python 2.7.

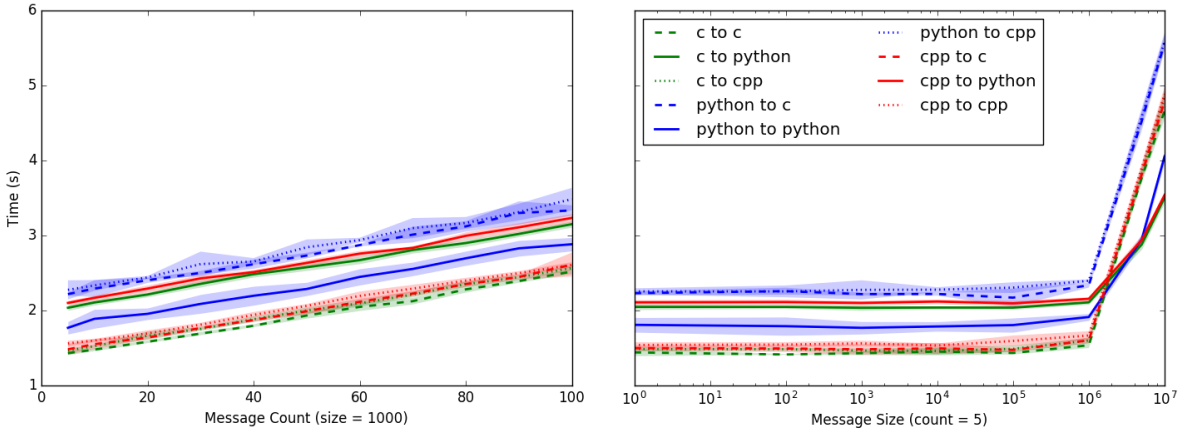

Fig. 3.— Comparison of communication time scaling between languages on Linux without Matlab. The same fiducial message size (1000 bytes) and count (5) was used as in Figure 2.

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

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).

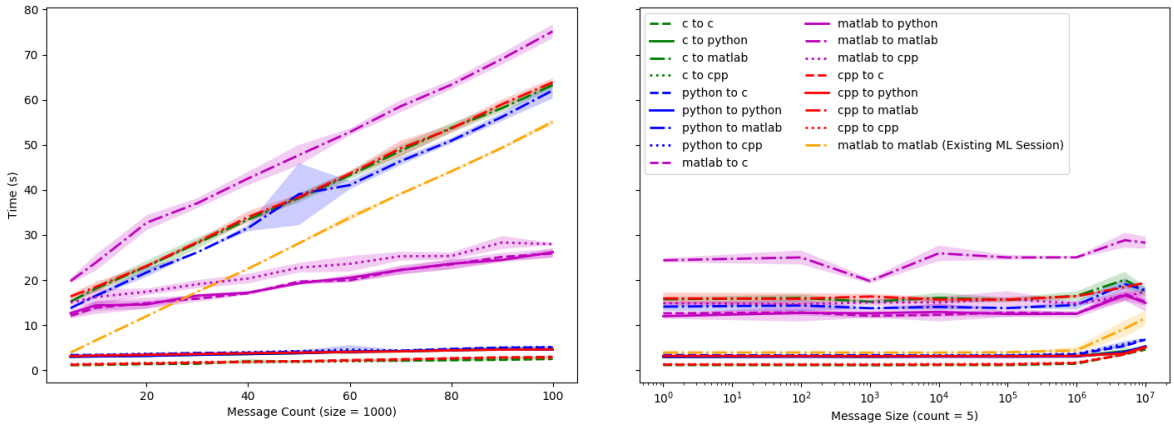

Fig. 4.— Comparison of communication time scaling between languages on Mac OS X including Matlab.

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	11.61
Matlab	C	0.144	11.82
Matlab	C++	0.142	14.79
Matlab	Python	0.139	12.23
Matlab	Matlab	0.559	19.28
Matlab (started)	Matlab (started)	0.537	1.25

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

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 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 driver 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.

4.3. 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 ZeroMQ communication and two different integration (Python-to-Python and C-to-C). There is $\sim 1\,\mathrm{s}$ additional overhead with Python 3.5 compared to Python 2.7 due to the

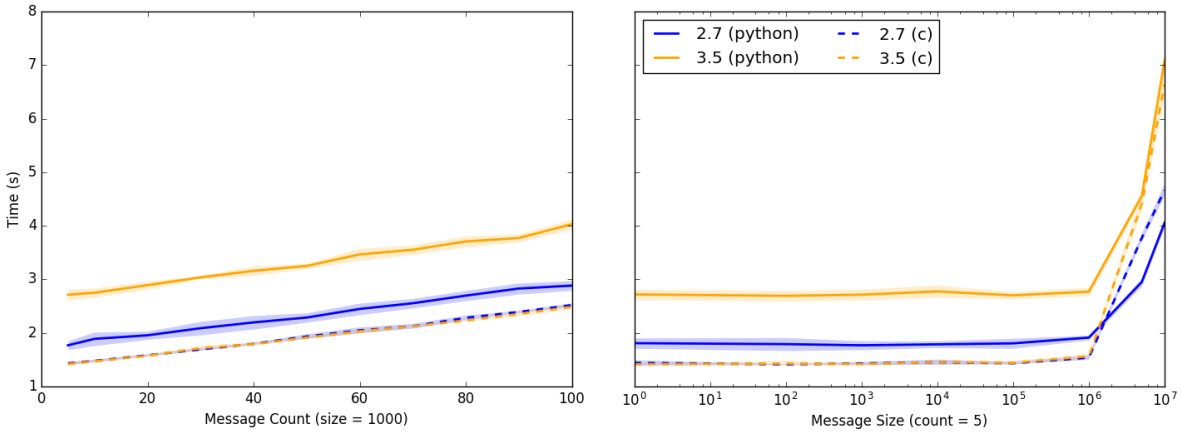

Fig. 5.— Comparison of communication time scaling between Python versions on Linux. The solid lines are for the Python-to-Python integrations and the dotted lines are for the C-to-C integration.

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.

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 drivers to transport messages between models and is more pronounced in integrations with Python models due to the increased amount of Python code.

4.4. 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 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 ZeroMQ communication and the Python-to-Python integration. Both the time per message and overhead changed between operating systems with the

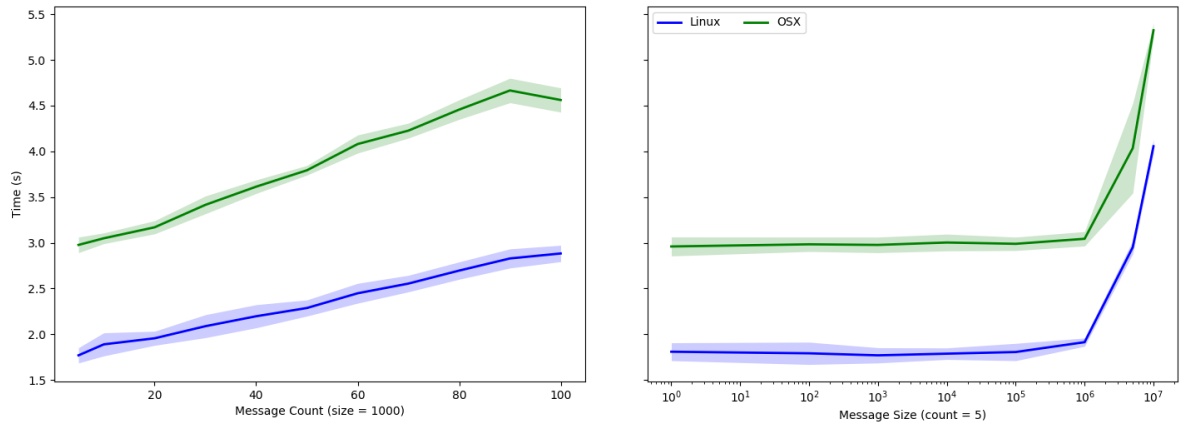

Fig. 6.— Comparison of communication time scaling between operating systems.

Linux machine performing the most consistently and fastest. 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 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. Windows results & discussion

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

Table 5: Effect of operating system on performance.

5. Summary & Discussion

5.1. 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.

`cis_interface` has already been used to integrate several plant models within the Crops in Silico organization with ongoing work to add additional models to the integration network. Figure 7 shows the progress so far. Each square is a model and lines between the models represent connections where information is exchanged. Solid lines are connections that have already implemented, dashed lines are connections currently being implemented, and dotted lines are connections planned for the future. Results from one integration, a metabolic model and genetic response to CO_2 model, can be found in Kannan et al. (2018).

5.2. 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`.

5.2.1. 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

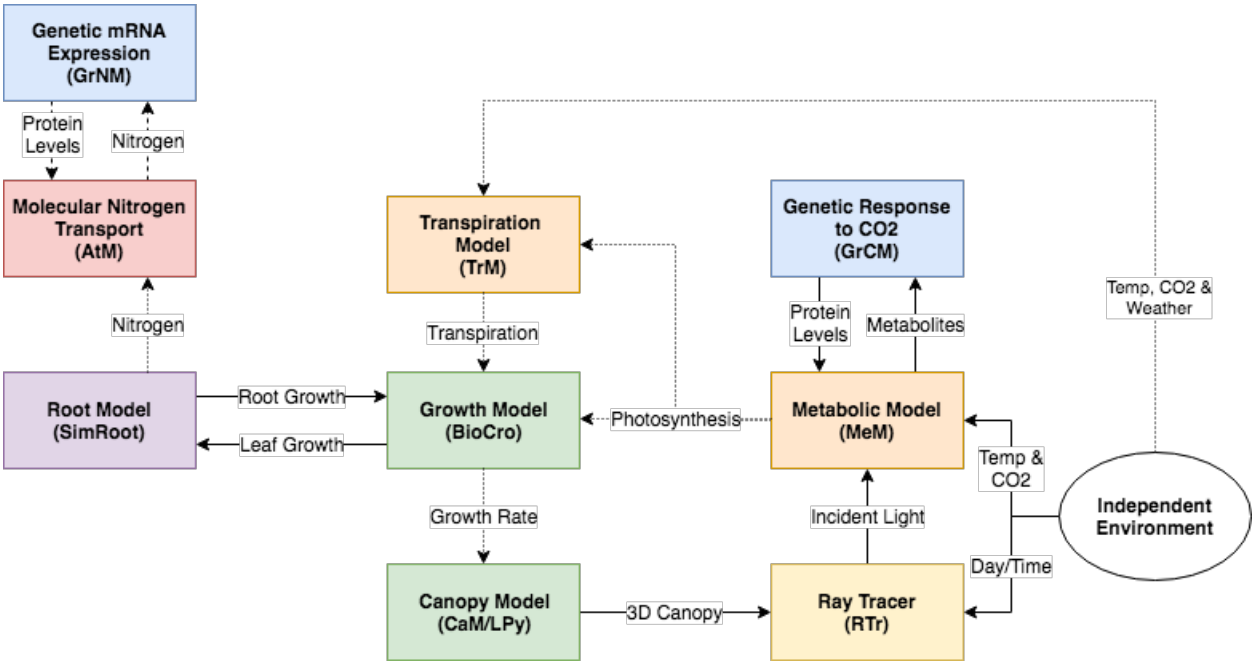

Fig. 7.— Progress towards full Crops in Silico integration network.

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.

5.2.2. 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 RabbitMQ to permit access to these models within integration networks.

5.2.3. 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.

5.2.4. 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.

5.2.5. 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.

5.2.6. 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.

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), Institute for Sustainability, Energy, and Environment (iSEE), the National Center for Supercomputing Applications (NCSA), and 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

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

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

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

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

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)

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

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

Kannan, K., Wang, Y., & Marshall-Colon, A. 2018, in prep.

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

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.

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/