INSTRUCTIONS

1. **Author groups:** Please check that all names have been spelled correctly and appear in the correct order. Please also check that all initials are present. Please check that the author surnames (family name) have been correctly identified by a pink background. If this is incorrect, please identify the full surname of the relevant authors. Occasionally, the distinction between surnames and forenames can be ambiguous, and this is to ensure that the authors' full surnames and forenames are tagged correctly, for accurate indexing online. Please also check all author affiliations.

2. **Figures:** If applicable figures have been placed as close as possible to their first citation. Please check that they are complete and that the correct figure legend is present.

3. **Missing elements:** Please check that the text is complete and that all figures, tables and their legends are included.

4. **Funding:** Please provide a Funding statement, detailing any funding received. Remember that any funding used while completing this work should be highlighted in a separate Funding section. Please ensure that you use the full official name of the funding body, and if your paper has received funding from any institution, such as NIH, please inform us of the grant number to go into the funding section. We use the institution names to tag NIH-funded articles so they are deposited at PMC. If we already have this information, we will have tagged it and it will appear as coloured text in the funding paragraph. Please check the information is correct. [red text to be used for suppliers who are tagging the funding]

5. **Conflict of interest:** All authors must make a formal statement indicating any potential conflict of interest that might constitute an embarrassment to any of the authors if it were not to be declared and were to emerge after publication. Such conflicts might include, but are not limited to, shareholding in or receipt of a grant or consultancy fee from a company whose product features in the submitted manuscript or which manufactures a competing product. The following statement has been added to your proof: 'Conflict of Interest: none declared'. If this is incorrect please supply the necessary text to identify the conflict of interest.

6. **Abbreviations:** some commonly used abbreviations have been automatically expanded for clarity to readers. By using the tracked changes proof for reference, please check the expanded abbreviations have been made correctly and mark up any corrections (if required) here or by email, listing the line and page numbers your correction refers to.

MAKING CORRECTIONS TO YOUR PROOF

These instructions show you how to mark changes or add notes to your proofs using Adobe Acrobat Professional versions 7 and onwards, or Adobe Reader DC. To check what version you are using go to **Help** then **About**. The latest version of Adobe Reader is available for free from get.adobe.com/reader.

DO NOT OVERWRITE TEXT, USE COMMENTING TOOLS ONLY.

DISPLAYING THE TOOLBARS

Adobe Reader DC

In Adobe Reader DC, the Comment toolbar can be found by clicking 'Comment' in the menu on the right-hand side of the page (shown below).
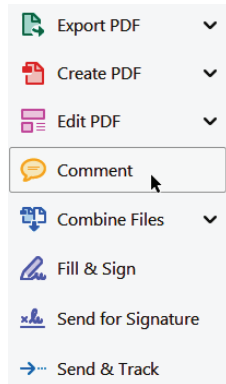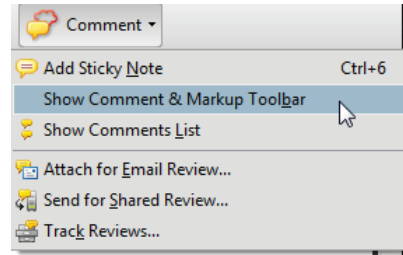
The toolbar shown below will then display along the top.

Acrobat Professional 7, 8, and 9

In Adobe Professional, the Comment toolbar can be found by clicking 'Comment(s)' in the top toolbar, and then clicking 'Show Comment & Markup Toolbar' (shown below).
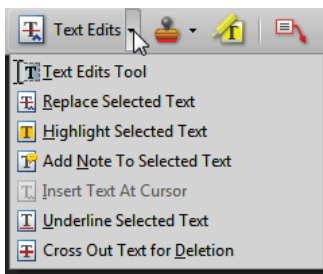
The toolbar shown below will then be displayed along the top.

USING TEXT EDITS AND COMMENTS IN ACROBAT

This is the quickest, simplest and easiest method both to make corrections, and for your corrections to be transferred and checked.

1. Click **Text Edits**
2. Select the text to be annotated or place your cursor at the insertion point and start typing.
3. Click the **Text Edits** drop down arrow and select the required action.

You can also right click on selected text for a range of commenting options, or add sticky notes.

SAVING COMMENTS

In order to save your comments and notes, you need to save the file (**File, Save**) when you close the document.

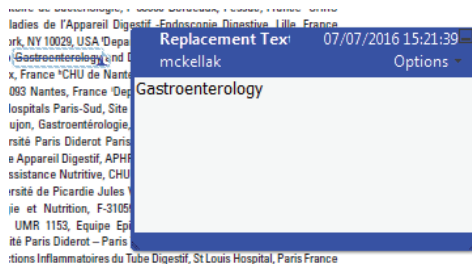## USING COMMENTING TOOLS IN ADOBE READER

All commenting tools are displayed in the toolbar.
To edit your document, use the highlighter, sticky notes, and the variety of insert/replace text options.
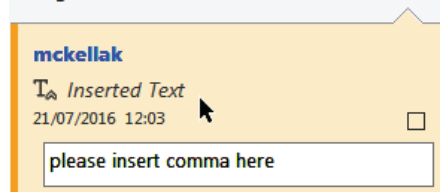
POP-UP NOTES

In both Reader and Acrobat, when you insert or edit text a pop-up box will appear. In **Acrobat** it looks like this:

In **Reader** it looks like this, and will appear in the right-hand pane:

AUTHOR QUERIES - TO BE ANSWERED BY THE CORRESPONDING AUTHOR

Please ensure that all queries are answered, as otherwise publication will be delayed and we will be unable to complete the next stage of the production process.

Please note that proofs will not be sent back for further editing.

The following queries have arisen during the typesetting of your manuscript. Please click on each query number and respond by indicating the change required within the text of the article. If no change is needed please add a note saying "No change."

AQ1	Figures have been placed as close as possible to their first citation. Please check that they have no missing sections and that the correct figure legend is present.
AQ2	Please provide online accessed date for all references having website address.
AQ3	Please check that all names have been spelled correctly and appear in the correct order. Please also check that all initials are present. Please check that the author surnames (family name) have been correctly identified by a pink background. If this is incorrect, please identify the full surname of the relevant authors. Occasionally, the distinction between surnames and forenames can be ambiguous, and this is to ensure that the authors' full surnames and forenames are tagged correctly, for accurate indexing online. Please also check all author affiliations.
AQ4	Please provide publisher location for Akgul (2013).
AQ5	Please list author names in all references having et al.
AQ6	Please provide article title for all journal-type references.
AQ7	Please provide the article title presented in the conference, expanded version of conference name and publisher detail for Coakley et al. (2012).
AQ8	Please provide publisher location for Eaton (2002).
AQ9	Please provide the article title presented in the conference, expanded version of conference name for Gabriel et al. (2004), Jupyter et al. (2018), Kluyver et al. (2016).
AQ10	Please provide journal title and page range for Hairer et al. (1993).
AQ11	Unpublished references are not allowed in the reference list. So, please update Kannan et al. (2018) with full publication detail. If this is still in preparation, delete it from the reference list and replace the citation with K. Kannan et al., in prepartaion.
AQ12	Please provide publisher location for Martin and Hoffman (2006).
AQ13	Please provide article title, journal title, volume number and page range for Ooms (2014).
AQ14	Please provide the article title presented and expanded version of the conference name.
AQ15	Please confirm if edition number is correct and provide publisher detail for the reference Rusling (1999).
AQ16	Please provide publisher location for Stallman et al. (2004), Turk (1994).
AQ17	Please review the suggested running head. If it is not okay, provide a short title less than 75 characters, including spaces.
AQ18	Resolution for all the supplied figures are found to be low than usually printed. Please provide high resolution images for better processing.

Original Research

yggdrasil: a Python package for integrating computational models across languages and scales

AQ3
Meagan Lang*

National Center for Supercomputing Application, University of Illinois, Urbana-Champaign, IL

Abstract. Thousands of computational models have been created within both the plant biology community and broader scientific communities 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 have slowed this progress. We present yggdrasil (previously cis_interface), a Python package for running integration networks with connections between models across languages and scales. yggdrasil 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 modelling. yggdrasil is designed to be user-friendly and can be accessed at https://github.com/cropsinsilico/yggdrasil. Although originally developed for plant models, yggdrasil can be used to connect computational models from any domain.

Keywords: Communication; computational framework; model integration; modelling; parallel processing; plant modelling

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 simple root growth model with a shoot growth model. Our example root growth model is written in C (see Listing 1) and can be expressed as

$$R_{t+1} = R_t \times r_r \times dt + R_t. \qquad (1)$$

*Corresponding author's e-mail address: langmm@illinois.edu

The inputs to the root growth model are: R_t the root mass at time t (g), r_r the relative root growth rate (h^{-1}) and dt the time step (h), 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

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

The inputs to the shoot model are: S_t the shoot mass at time t (kg), r_s the relative shoot growth rate (d^{-1}), dt the time step (days), R_t the root mass at time t (kg), and R_{t+1} the root mass at time $t + 1$ (kg), and the output of the shoot growth model is S_{t+1}, the shoot mass at the next time step. Although 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 duplication of model algorithms.

We present an Open Source Python package, yggdrasil, for creating and running integration networks by connecting existing computational models written in different programming languages. yggdrasil 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. yggdrasil is available on Github (https://github.com/cropsinsilico/yggdrasil) with full documentation (https://cropsinsilico.github.io/yggdrasil/) and can be installed from PyPI (https://pypi.org/project/yggdrasil/) via pip install yggdrasil-framework or conda-forge (https://github.com/conda-forge/yggdrasil-feedstock) via conda -c conda-forge install yggdrasil. Although developed for the purpose of integrating plant biology models, yggdrasil can be applied to any situation requiring coordination between models in the supported languages. The following section provides background information on existing efforts for cross-language model integration, the Methods section describes the methods used by the yggdrasil package to integrate models, the Worked example section provides an example of integrating the two models presented above, the Results and discussion section presents several tests demonstrating the performance of message passing between integrated models and the final section describes a few use cases for yggdrasil, summarizes the features and limitations of the yggdrasil package and also outlines areas of ongoing and future development.

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), Visual Basic (Humphries and Long 1995; Hall and Minchin 2013), R (Wang *et al.* 2015), Java (Kappas *et al.* 2013; Song *et al.* 2013), Fortran (Goudriaan and Laar 1994) and several domain-specific languages (DSLs, e.g. Systems Biology Markup Language (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 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 centimetres, the photosynthetic photon flux density output by a ray tracer like fastTracer (Song *et al.* 2013) (which expects a geometry in units of metres) would be incorrect.

As a result, integrating the two biological models such as the root and shoot models discussed in the Introduction section 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?

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 (i) computationally inefficient when many iterations are necessary, such as during parameter searches or steady-state convergence tests, (ii) unique to every integration, requiring the production of new scripts, (iii) time consuming, (iv) prone to error and (v) 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: DSLs and frameworks.

Domain-specific languages

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 DSLs for model representation. For example, SBML is an XML-based format for representing models of biological processes (Hucka *et al.* 2003). Scientists can compose their models using SBML mark-up 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.

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 co-ordinating the execution of tasks in parallel or serial (e.g. Babuji *et al.* 2018; Celery 2018; Luigi 2018). Although 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: (i) the model must be written in (or exposable to) the language of the framework and (ii) 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 often provided limited support for models written without the framework in mind.

yggdrasil is an example of a framework that overcomes these issues by exposing simple and easily accessible interfaces in the languages of the models (see Interface section) that permit messages to be passed between the model processes as they run in parallel (see Communication section). As a result, models' writers only need knowledge of the language in which their model is written.

Methods

yggdrasil was designed to transform models into building blocks that can be easily combined with other blocks to build complex structures, thus promoting model reuse and collaboration between scientists with varying degrees of overlapping expertise. yggdrasil does so by addressing the questions from Background section 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 does not 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.

yggdrasil is built upon several well-established open-source software packages (see below) along with new tools developed explicitly for yggdrasil including

utilities for running and monitoring models written in other languages from Python, dynamically creating and managing communication networks, data-type conversions between languages and asynchronous message passing with variable underlying communication mechanisms. yggdrasil currently support models written in Python, Matlab, C and C++ with additional DSL support for LPy models (Boudon *et al.* 2012). Support for additional languages is planned for future development (see Language support section).

Orchestration

yggdrasil is executed via a command line interface (CLI), yggrun with specification files (see Specification files section) as input. On the basis of the information contained in the specification files, yggdrasil dynamically establishes a network of asynchronous communication channels (see Communication section) and launches the models on new processes (see Model execution section). Although written in Python, yggdrasil was written such that users need not have any knowledge of the Python language and can interact with yggdrasil solely through the CLI. In addition to the CLI, more advanced capabilities are also exposed via the Python API including access to more detailed information about the running integration.

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. PyYAML 2006; Simonov 2006; JS-YAML 2011). The declarative format allows user to specify exactly what they want to do, without describing how it should be done. Although 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 (e.g. array, scalar, mesh) and physical 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 yggdrasil 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 as will be seen in Worked example section.

Model execution. yggdrasil launches each model in an integration in 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 the Introduction section (described further in the Worked example section) require 10.28 and 10.40 s, respectively, to run for 100 time steps in isolation with direct input/output from/to files in their native language. If these two models were manually integrated in serial via the command line, the integration would require a total of 20.68 s. However, because yggdrasil offers parallel execution of the models, the same integration requires only 16.03 s when using yggdrasil, a speed-up of 1.29. The exact speedup provided by parallel integration using yggdrasil 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. If the models are dependent on one another, the models may have to wait to receive messages and the speedup will not be as great. The Speedup from parallelism section provides additional information about the speedups achievable through yggdrasil parallel integration and how model structure influences the size of the performance boost.

Although every model is executed in a new process, how the model is handled depends on the language it is written in. yggdrasil has a dedicated driver for each of the supported languages with utilities that allow yggdrasil to launch and monitor executables written in that language from Python. Models written in interpreted languages (Python and Matlab) are executed on the command line via the interpreter. In the case of Matlab, where a significant amount of time is required to start the Matlab interpreter (see Language), 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 include the source code for the appropriate dependencies at compilation and link against the appropriate yggdrasil header library. yggdrasil provides several command line tools for determining the locations of the necessary libraries and any required compilation/linking flags. Alternatively, users can provide the location of the model source code and let one of the several yggdrasil drivers handle the compilation, including linking against the appropriate yggdrasil interface library. yggdrasil also has support for compiling models using Make (Stallman *et al*. 2004) and CMake (Martin and 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 yggdrasil.

Once the models are running, yggdrasil 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 are provided as Algorithms 4 and 5, respectively, in Appendix.

Communication

Integrating models requires that they are able to send and receive information to and from other models written in different languages via some communication mechanism. Communication within yggdrasil integration networks was designed to be flexible in terms of the languages, platforms and data types available. To accomplish this, yggdrasil leverages three different tools for communication, System V IPC Queues (see System V IPC Queues section), ZeroMQ (see ZeroMQ section) and RabbitMQ (see RabbitMQ section), which each have their own strengths and weaknesses. The particular communication mechanism used by yggdrasil 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, simplifying the number of routines that users need to use for integration (see Interface section).

yggdrasil 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 communication mechanisms, using a tool developed specifically for yggdrasil 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 yggdrasil processes (e.g. enforcing dependencies on other model and connection threads).

System V interprocess communication queues. The first communication mechanism used by yggdrasil 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 Communication mechanism section) 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 or 256 64-bit numbers). Once the queue is full or if the message is larger than the limit, any process attempting to send an additional message will stop 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 stop indefinitely. This can be handled by splitting large messages into multiple smaller messages (see Sending/receiving large messages section); however, the time required to send a message increases with the number of message it must be broken into (see Communication mechanism section). These limits make sending large messages relatively inefficient when compared with other communications mechanisms. As a result, IPC queues are used by yggdrasil only as a fallback on Posix systems if none of the other supported communication libraries have not been installed.

ZeroMQ. The preferred communication mechanism used by yggdrasil 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 yggdrasil in

order to allow support on Windows and for future target languages (see Ongoing/future improvements section) that could not be accomplished using IPC queues. In addition, while ZMQ allows ~~IPC~~ via IPC, ZMQ also supports protocols for distributed communication via an Internet Protocol (IP) network. While yggdrasil does not currently support using these protocols for distributed integration networks, this is an avenue of future development that has been prepared for.

In addition to using the default ZMQ libraries, yggdrasil also includes supporting routines for allowing received messages to be confirmed. Because ZMQ is broker-less, a socket has no way of knowing if the message it sent was successfully received. This lack of confirmation makes it harder to determine whether there is was error somewhere along the network. yggdrasil overcomes this by generating two ZMQ sockets for every connection: one for passing the messages and one for confirming them. Each time a message is received, the receiving socket confirmation thread will send a confirmation including a unique ID for the received message and then wait for a reply to that confirmation. The sending socket confirmation thread will continuously check the confirmation sockets for messages indicating that sent messages were received. Once a confirmation message is received, it will send a reply to the receiving confirmation socket and record the ID of the message that was confirmed. This handshake operation ensures that all messages are accounted for so yggdrasil knows if a message is lost and where it was lost.

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, yggdrasil includes support for brokered communication via RabbitMQ (RMQ; RabbitMQ 2007) that will be used during future development to allow integrations to run on distributed resources or include remote models run as services (see Distributed systems section). Due to the slower message speed, yggdrasil does not currently use RMQ for communication unless explicitly specified by the user in their integration network YAML.

Asynchronous message passing. The same asynchronous communication strategy is used with each of the communication mechanisms supported by yggdrasil. 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 checks input channels on another thread, moving received messages into a intermediate 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. Fig. 1 describes the general flow of messages, Algorithms 6 and 7 in Appendix describe the asynchronous output and input channel procedures and Algorithm 8 describes the procedure followed by input and output connection drivers.
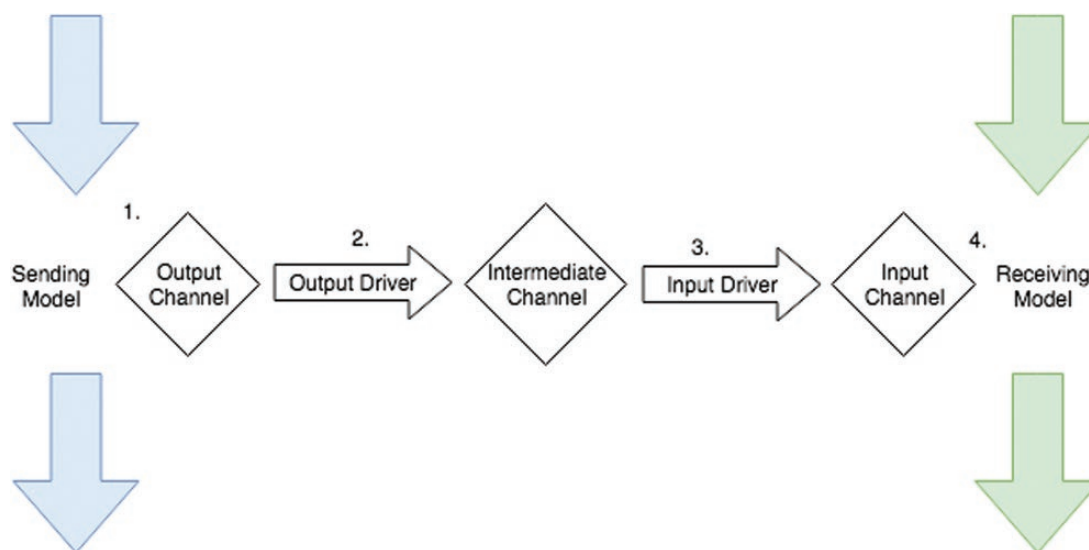

Fig. 1. Diagram of how messages are passed asynchronously using input/output drivers and an intermediate channel.

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 an 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.

Sending/receiving large messages. All of the communication tools leveraged by yggdrasil 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, yggdrasil splits the message up into multiple smaller messages. In addition, for large messages, yggdrasil 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. The procedures for creating and using temporary channels for sending large messages can be seen in the Send and Recv methods from Algorithms 6 and 7, respectively, in Appendix.

Interface. yggdrasil 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. Each language interface is an implementation of Algorithms 6 and 7 from Appendix that provides users with send and receive calls for passing messages. The Python interface provides communication classes for sending and receiving messages. The Matlab interface provides a simple wrapper class for the Python class, which 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 that can be found in the yggdrasil documentation.

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 metadata, this is not true of all data formats. To combat this, yggdrasil requires models to explicitly specify the format of input and output expected by a model in the model YAML. yggdrasil 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 yggdrasil include:

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

In addition, yggdrasil 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), yggdrasil will automatically perform the necessary conversions before passing data from one model to the next.

Worked example

In order to illustrate how yggdrasil is intended to be used by model writers, the following walks through the integration of the two examples models discussed in the Introduction section. All of the source code and YAML files discussed in this section are available in the yggdrasil 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:~~

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). In addition to the actual calculation, both models include sleep statements (lines 11 and 18 in Listings 1 and 2, respectively). These statements are meant to simulate a longer, more involved calculation representative of a more realistic model.

```c
1 #include <unistd.h>
2
3 /*!
4   @brief  Calculation  of  root  mass
      following time step.
5   @param[in]  r_r  double  Relative  root
      growth rate.
6   @param[in] dt double Duration of time
      step.
7   @param[in]  R_t  double  Current  root
      mass.
8   @returns double Root mass after time
      step.
9 */
10 double calc_root_mass (double r_r, double dt,
     double R_t) {
11  usleep (1e5) ; // To simulate a long
     r calculation
12  return R_t + (R_t * r_r * dt) ;
13 };
```

AQ1

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

```python
1 import time
2
3
4 def calc_shoot_mass (r_s, dt, S_t, R_t, R_tp1) :
5   r """ Calculate the shoot mass.
6
7   Args:
8       r_s (float) : Relative shoot growth
          rate.
9       dt (float) : The time step.
10      S_t (float) : Previous shoot mass.
11      R_t (float) : Previous root mass.
12      R_tp1 (float) : Root mass at the
          next timestep.
13
14  Returns :
15      float : Shoot mass at the next
          timestep.
16
17  """
18  time . sleep (0.1 )  # To simulate a
      longer calculation
19   return (S_t * r_s * dt) + S_t – (R_tp1 – R_t)
```

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

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.

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.

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 the Wrapper section) that calls the model code such that no modification needs to be done to the original model code in order to integrate it.

```
1  model:
2     name : RootModel
3     language : c
4     args : . / src / root_wrapper . c
5     inputs:
6        - name : root_growth_rate
7          units : hr**-1
8        - name : init_root_mass
9          units : g
10       - name : root_time_step
11         units : hr
12    outputs:
13       - name : next_root_mass
14         units : g
```

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

```
1  model:
2     name : ShootModel
3     language : python
4     args : . / src / shoot_wrapper . py
5     inputs:
6        - name : shoot_growth_rate
7          units : d**-1
8        - name : init_shoot_mass
9          units : kg
10       - name : shoot_time_step
11         units : d
12       - name : next_root_mass
13         units : kg
14    outputs:
15       - name : next_shoot_mass
16         units : kg
```

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

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 yggdrasil 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 at https://cropsinsilico.github.io/yggdrasil/yaml.html.

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 the Data aggregation and forking section) that will allow fields to be aggregated within connection YAMLs and simplify the send and receive calls in such cases.

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 yggdrasil 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 yggdrasil to perform this step automatically based on YAML options for simple cases such as single loops or if statements (see the Wrapper automation and control flow section), 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 yggdrasil and the actual model function.

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 1 and 2, respectively. 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.

Algorithm 1. Root wrapper

```
1: Receive r_r, R_t
2: Send R_t              ▷ So that there is a
                            complete record.
3: while true do
4:   if dt available then
5:      Receive dt
6:   else
7:      Break
8:   end if
9:   R_{t+1} ← calc_root_mass(r_r, dt, R_t)
10:  R_t ← R_{t+1}
11:  Send R_t
12: end while
```

Algorithm 2. Shoot wrapper

```
1: Receive r_s, S_t, R_t
2: Send S_t         ▷ So that there is a
                       complete record.
3: while true do
4:   if dt available then
5:      Receive dt
6:   else
7:      Break
8:   end if
9:   Receive R_{t+1}
10:  S_{t+1} ← calc_shoot_mass(r_s, dt, S_t, R_t, R_{t+1})
11:  S_t ← S_{t+1}
12:  R_t ← R_{t+1}
13:  Send S_t
14: end while
```

Wrapper code. With some translation into the appropriate language and the addition of calls to the appropriate yggdrasil 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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 // Include C interface
4 #include "YggInterface.h"
5 // Include C header containing model
  calculation
6 #include "root.h"
7
8
```

```
9 int main (int argc, char *argv[ ]) {
10   int i, ret;
11   double r_r, dt, R_t, R_tp1;
12
13   // Create input / output channels
14   yggInput_t RootGrowthRate = yggInput
     ("root_growth_rate");
15   yggInput_t InitRootMass = yggInput
     ("init_root_mass");
16   yggInput_t TimeStep = yggInput
     ("root_time_step");
17   yggOutput_t NextRootMass = yggOutputFmt
     ("next_root mass", "%lf \n");
18
19   // Receive root growth rate
20   ret = yggRecv (RootGrowthRate, &r_r);
21   if (ret < 0) {
22    printf ("root: Error receiving root
     growth rate .\n");
23     return −1;
24   }
25   printf ("root: Received root growth
     rate: %lf \n", r_r);
26
27   // Receive initial root mass
28   ret = yggRecv (InitRootMass, &R_t);
29   if(ret < 0) {
30     printf ("root: Error receivin
       initial root mass. \n");
31     return −1;
32   }
33   printf ("root: Received initial root
     mass: %lf \n", R_t);
34
35   // Send initial root mass
36   ret = yggSend (NextRootMass, R_t);
37   if (ret < 0)
38     printf ("root: Error sending initial
       root. \n");
39     return −1;
40   }
41
42   // Keep advancing until there arn't
     any new input times
43   i = 0;
44   while (1) {
45
46     // Receive the time step
48     if (ret < 0) {
49       printf ("root: No more time steps.
         \n", dt);
50       break;
51     }
```

10.5
10.10
10.15
10.20
10.25
10.30
10.35
10.40
10.45
10.50
10.52

10.55
10.60
10.65
10.70
10.75
10.80
10.85
10.90
10.95
10.100
10.104

```
52     printf ("root: Received next time
       step: %lf \n", dt);
53
54     // Calculate root mass
55     R_tp1 = calc_root_mass (r_r, dt, R_t);
56     printf ("root: Calculated next root
       mass: %lf \n", R_tp1);
57
58     // Output root mass
59     ret = yggSend (NextRootMass, R_tp1);
60     if (ret < 0) {
61       printf ("root: Error sending root
         mass for time step %d. \n", i + 1);
62       return −1;
63     }
64
65     // Advance root mass to next time step
66     R_t = R_tp1;
67     i++;
68
69 }
70
71 return 0;
72
73 }
```

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

```
1 # Import Python interface
2 from yggdrasil . interface import
  YggInput, YggOutput
3 # Import Python module containing model
  calculation
4 from shoot import calc_shoot_mass
5
6
7 # Create input / output channels
8 ShootGrowthRate = YggInput
  ('shoot_growth_rate')
9 InitShootMass = YggInput
  ('init_shoot_mass')
10 TimeStep = YggInput ('shoot_time_step')
11 NextRootMass = YggInput ('next_root_mass')
12 NextShootMass = YggOutput ('next_shoot_
   mass', '%lf \n')
13
14 # Receive shoot growth rate
15 flag, r_s = ShootGrowthRate . recv ( )
16 if not flag:
17     raise RuntimeError ('shoot : Error
       receiving shoot growth rate.')
18 print ('shoot: Received shoot growth
   rate : %f' % r_s)
19
```

```
20 # Receive initial shoot mass
21 flag, S_t = InitShootMass . recv ( )
22 if not flag:
23     raise RuntimeError ('shoot : Error
       receiving initial shootmass.')
24 print ('shoot: Received initial shoot
   mass : %f' % S_t)
25
26 # Receive inital root mass
27 flag, R_t = NextRootMass . recv ( )
28 if not flag:
29     raise RuntimeError ('shoot : Error
   receiving initial root mass.')
30 print ('shoot : Received initial root
   mass : %f' % R_t)
31
32 # Send initial shoot mass
33 flag = NextShootMass . send (S_t)
34 if not flag:
35     raise RuntimeError ('shoot : Error
       sending initial shoot mass.')
36
37 # Keep advancing until there arn't any
   new input times
38 i = 0
39 while True:
40
41   # Receive the time step
42   flag, dt = TimeStep . recv ( )
43   if not flag:
44     print ('shoot: No more time steps.')
45     break
46   print ('shoot: Received next time step
   : %f' % dt)
47
48   # Receive the next root mass
49   flag, R_tp1 = NextRootMass . recv ( )
50   if not flag:
51     # This raises an error because there
     must be a root mass for each time step
52     raise RuntimeError ('shoot: Error
     receiving root mass for time step
     %d.' % (i + 1))
53   print ('shoot : Received next root
   mass : %f' % R_tp1)
54
55   # Calculate shoot mass
56   S_tp1 = calc_shoot_mass (r_s, dt, S_t,
   R_t, R_tp1)
57   print ('shoot: Calculated next shoot
   mass: %f' % S_tp1)
58
59   # Output shoot mass
```

```
60   flag = NextShootMass . send (S_tp1)
61   if not flag:
62     raise RuntimeError ('shoot: Error
       sending shoot amss for time step
       %d.' % (i+1))
63
64   # Advance masses to next time step
65   S_t = S_tp1
66   R_t = R_tp1
67   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 yggdrasil 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. yggdrasil 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 yggdrasil 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 yggdrasil 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 (https://cropsinsilico.github.io/yggdrasil/c_format_strings.html).

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 whether 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 (https://cropsinsilico.github.io/yggdrasil/model_interface.html).

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 whether 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 yggdrasil. Secondly, 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 of incorporating changes that may occur in the original model code due to bug fixes or added features. Thirdly, 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.

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.

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.

```
1  connections :
2    # 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   # Output connections
14   - input : next_root_mass
15     output : ./Output/root_output.txt
16     filetype : table
17     field_names : root_mass
```

Listing 7. root files.yml: connection YAML specification file for running the root model in isolation.

```
1  connections :
2    # Input connections
3    - input : ./Input/shoot_growth_rate.txt
4      output : shoot_growth_rate
5      filetype : table
6    - input :../Input/init_shoot_mass.txt
7      output : init_shoot_mass
8      filetype : table
9    - input :../Input/timesteps.txt
10     output : shoot_time_step
11     filetype : table
12   - input :../Input/root_output.txt
13     output : next_root_mass
10     filetype : table
15
16   # Output connections
17   - input : next_shoot_mass
18     output :../Output/shoot_output.txt
19     filetype : table
20     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 yggdrasil 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, yggdrasil 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 yggdrasil to add the designated field names as a header line in the output table that is produced.

There are many file formats that yggdrasil 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 (https://cropsinsilico.github.io/yggdrasil/yaml.html#connection-options).

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.

The connections on lines 3–11 in Listing 9 are identical to lines 3–11 in Listings 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.

```
 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/timesteps.txt
25      output : shoot_time_step
26      filetype : table
27
28    # Shoot output connection
29    - input : next_shoot_mass
30      output : ./Output/shoot_output.
         txt
31      filetype : table
32      field_names : shoot_mass
```

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

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 specified in the model YAMLs and the headers of the input tables, yggdrasil 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, yggdrasil 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 and Matlab). For more advanced transformations, users can instruct yggdrasil to use any arbitrary Python function to transform data by passing the module import path as a connection field.

Running the integration

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

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

    yggrun root.yml root_files.yml

and for the shoot model this would be

    yggrun shoot.yml shoot_files.yml.

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

    yggrun 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 yggdrasil.

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. yggdrasil also has support for sending more complex data types like those discussed in the Transformation section. In addition, work is underway to add generic support for any data type that can be expressed as a JSON object (see the JSON data type specification section). Examples using these different data types can be found in the 'Formatted I/O' section of the documentation available at https://cropsinsilico.github.io/yggdrasil/formatted_io.html.

Model networks: It is possible to build up complex integration networks using yggdrasil 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 the Graphical user interface section) 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 yggdrasil to determine how two time steps should be reconciled. Therefore, the current version of yggdrasil 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 the Symbolic ODE models section). For two coupled ODE models, yggdrasil may be able to determine the correct time step for synchronization, but, as an incorrect time step, could drastically affect 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 yggdrasil has limited support for 'forking' connections, full support for this feature is under way as part of the data aggregation feature (see the Data aggregation and forking section) and will be part of the version 1.0 release.

Results and discussion

To evaluate yggdrasil, 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 yggdrasil 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.

Communication mechanism

Fig. 2 compares the execution times for the different communication mechanisms discussed in the Communication section for sending different numbers/sizes of messages from one Python model to another Python model. The tests were run on a Dell tower with
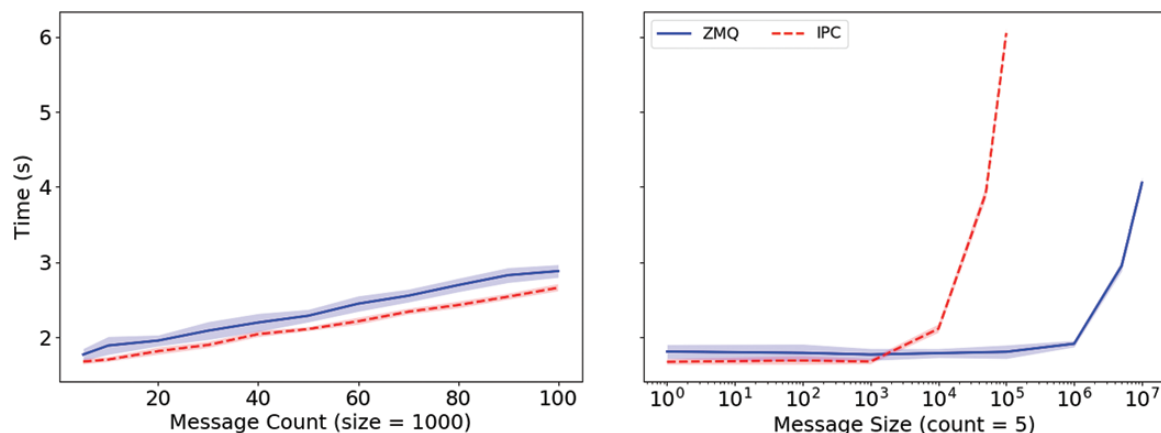

Fig. 2. Comparison of communication time scaling across communication methods on Linux. Lines show the average run time for each test with the standard deviation shown in the shaded region. Left: Scaling of execution time with number of 1000 byte messages sent. Right: Scaling of execution time with the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

Dual Intel(R) Xeon(R) CPU E5-2650 v3 at 2.30-GHz processors running Ubuntu 14.04. The left panel of Fig. 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 Fig. 2 shows the total time required to run the models and send five messages of varying lengths from one to the other.

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 Fig. 2. Time per message (slope of Fig. 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 Fig. 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 the sending/receiving large messages section, 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,
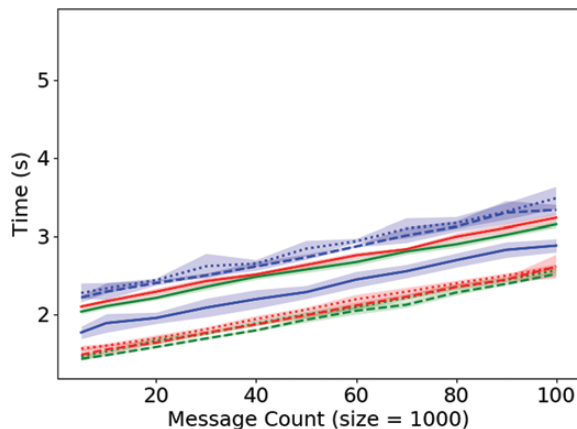
both ZMQ and IPC communication has no measurable dependence on message size.

Language

Fig. 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 Fig. 3 and Table 2 were run on the Linux machine from above while those for Fig. 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.

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 inconclusive. 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

Table 1. Effect of communication mechanism on performance.

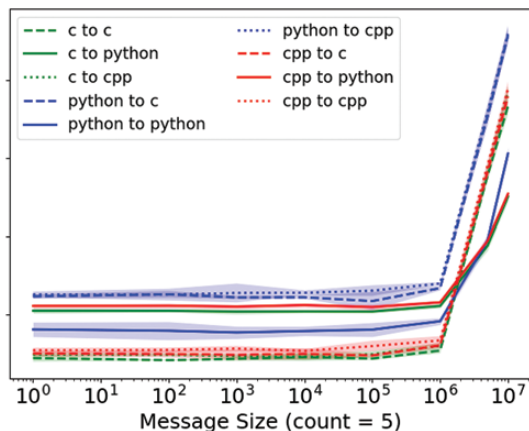| Mechanism	Time per message (s)	Overhead (s)
IPC	0.010	1.60
ZMQ	0.012	1.73

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 Fig. 2.

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 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, yggdrasil 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 Fig. 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, yggdrasil 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++

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

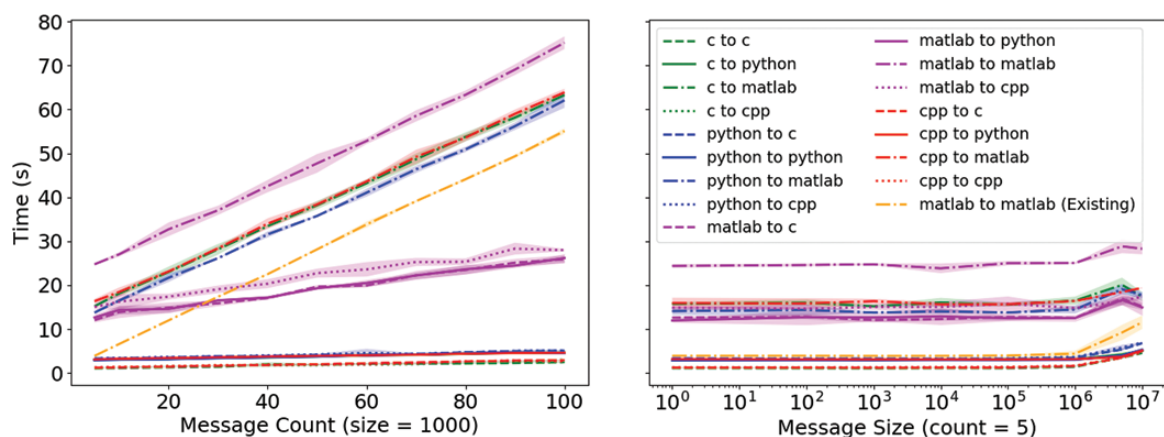| Source	Destination	Time per message (s)	Overhead (s)
C	C	0.011	1.36
C	C++	0.012	1.42
C	Python	0.012	1.99
C++	C	0.012	1.42
C++	C++	0.011	1.49
C++	Python	0.012	2.05
Python	C	0.012	2.15
Python	C++	0.012	2.21
Python	Python	0.012	1.73

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

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.

Python version

Fig. 5 and Table 4 compare the execution times when using different versions of Python. The performance tests

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

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 with 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 Fig. 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.

Operating system

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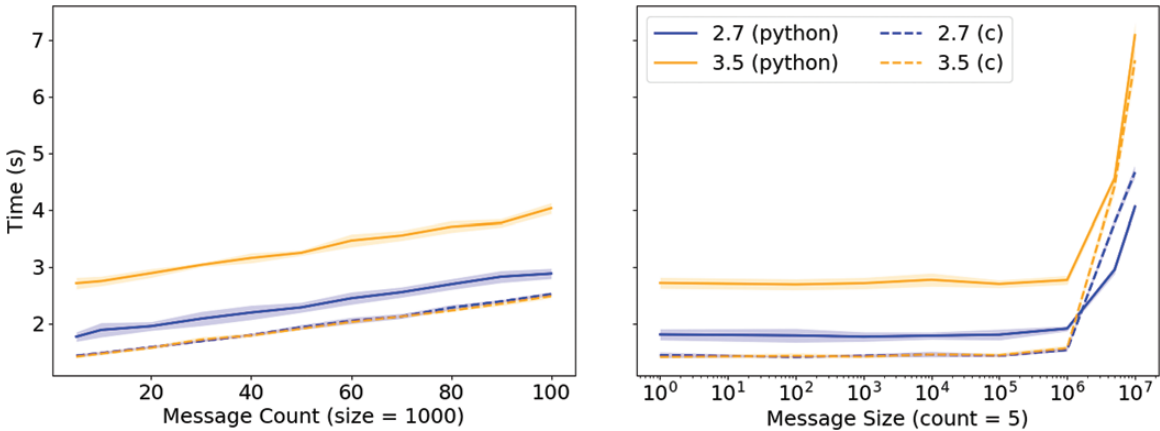

Fig. 5. Comparison of communication time scaling between Python versions on Linux. The solid lines are for the Python-to-Python integrations and the dotted lines are for the C-to-C integration.

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 yggdrasil performed consistently on each of the supported operating systems.

Speedup from parallelism

While efforts have been made to limit the overhead introduced by yggdrasil in the startup and communication steps of the integration, yggdrasil 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 discussed in the Worked example section 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 yggdrasil 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 yggdrasil integration in such cases. However, if the two models are not as tightly coupled and have calculations that can occur simultaneously, yggdrasil 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.

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \qquad (3)$$

For the example presented above of one model (model A) sending output to another (model B) and then waiting for a response, the speedup offered by yggdrasil through parallelism can be parameterized in terms of the durations of calculations for each model performed in the different phases of the process (see Fig. 7). During the initial phase ($t_{\text{init},A}$ and $t_{\text{init},B}$), both models perform setup tasks before immediately entering the loop. On entering the loop at Phase 1 ($t_{1,A}$ and $t_{1,B}$), both models perform calculations in preparation for the first communication where Model A sends a message to Model B. In Phase 2 ($t_{2,A}$ and $t_{2,B}$), 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 ($t_{3,A}$ and $t_{3,B}$), 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 ($t_{\text{final},A}$ and $t_{\text{final},B}$) to complete any necessary cleanup and teardown actions. If the models in the toy example were run in serial (without yggdrasil), the total execution time would be

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

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

Table 4. Effect of Python version on performance.

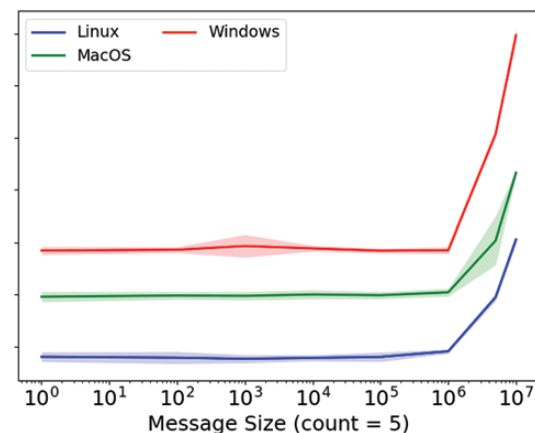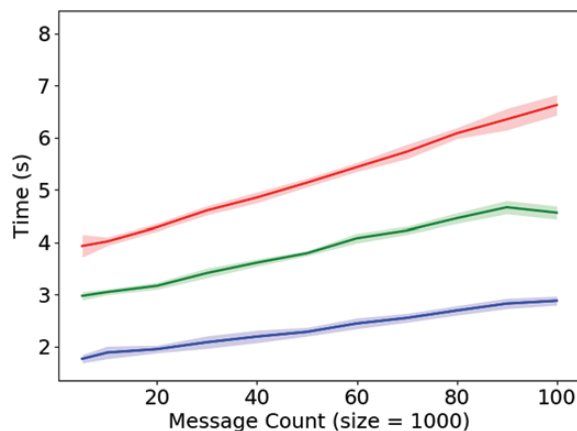| Python Version	Language	Time per message (s)	Overhead (s)
2.7	Python	0.012	1.73
2.7	C	0.011	1.36
3.5	Python	0.013	2.62
3.5	C	0.011	1.36

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

for models A and B, respectively. 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

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

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

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

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

$$(7)$$

where $T_{overhead}$ is the overhead incurred by using yggdrasil to run the models and setup the communication network, t_{comm} is the time required to complete the exchanges via each yggdrasil asynchronous communication, n is the number of

Table 5. Effect of operating system on performance.

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

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 3)

Algorithm 3. Parallel execution time

```
1:  T_A ← t_init;A
2:  T_B ← t_init;B
3:  iloop ← 0
4:  while iloop <= N_loop do
5:      T_A+ = t_1,A
6:      T_B+ = t_1;B
7:      T_B = max(T_A; T_B +      ▷ Model B blocking
            t_comm)                    on receive.
8:      T_A+ = t_2;A
9:      T_B+ = t_2;B
10:     T_A = max(T_A +       ▷ Model B blocking
            t_comm; T_B)            on receive.
11:     T_A+ = t_3;A
12:     T_B+ = t_3;B
13:     iloop+ = 1
14:  end while
15:  T_A+ = t_final;A
16:  T_B+ = t_final;B
17:  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 Fig. 8. Assuming that $t_{init,B} < (t_{init,A} + t_{1,A})$, this integration would have a parallel execution time and speedup of

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

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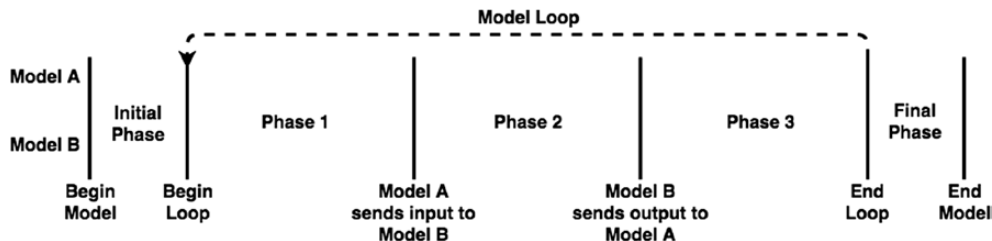$$

Fig. 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.
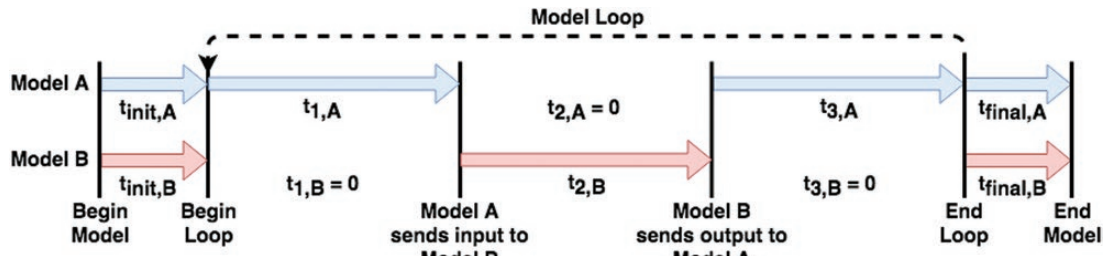
Fig. 8. Diagram of parallelism for a fully coupled set of models. The shaded arrows represent active calculation by models A (top) and B (bottom). Absence of an arrow indicates idle time where the model process is not performing any calculations.
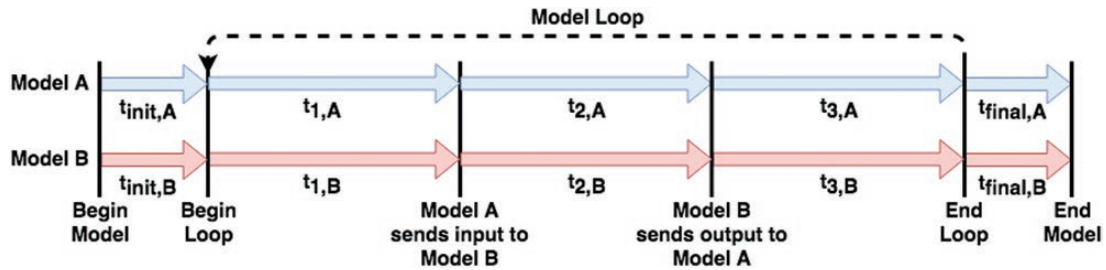
Fig. 9. Diagram of parallelism for a set of perfectly matched models.
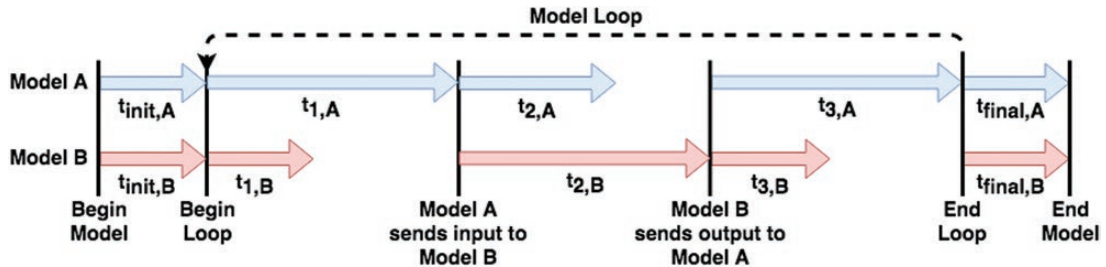
Fig. 10. Diagram of partial parallelism for a set of imperfectly matched models.

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 yggdrasil 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 ($t_{\text{comm}} = 0.02$ s/msg and $T_{\text{overhead}} = 2.86$ s) where $t_{1,A}$, $t_{2,B}$ 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 ($t_{\text{init,A}} = t_{\text{init,B}}$, $t_{1,A} = t_{1,B}$ etc.) as shown in Fig. 9, the parallel execution time and speedup of the integration would be

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

(10)

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

(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 ($t_{\text{comm}} = 0.02$ s/msg and $T_{\text{overhead}} =$ 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 shown in Figs 8 and 9 are rarely the case for actual models. Fig. 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 ($t_{\text{comm}} = 0.02$ s/msg and $T_{\text{overhead}} = 2.86$ 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.

Summary

Current status

yggdrasil is an open-source Python package for connecting computational models across programming languages and scales to form integration networks. 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

are run in parallel with asynchronous message passing handled under-the-hood via threading and one of the three communication mechanisms. yggdrasil works on Linux, Mac OS X and Windows operating systems with Python 2.7, 3.4, 3.5, 3.6 and 3.7. yggdrasil 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 and Hoffman 2006) and running models written in the LPy (Boudon *et al.* 2012) DSL.

yggdrasil is designed to make models as reusable as possible. Once integrated via yggdrasil, models can be combined with any other models that also have a yggdrasil wrapper and YAML specification file with minimal, if any, modification. This 'plug-and-play' approach to models encourages model writers to make their codes public in a reusable form, generating citations and reuse of their work, while also growing a community of models and model writers capable of probing areas otherwise unreachable. yggdrasil 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. Fig. 11 shows the progress so far. Each square is a model, colour 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

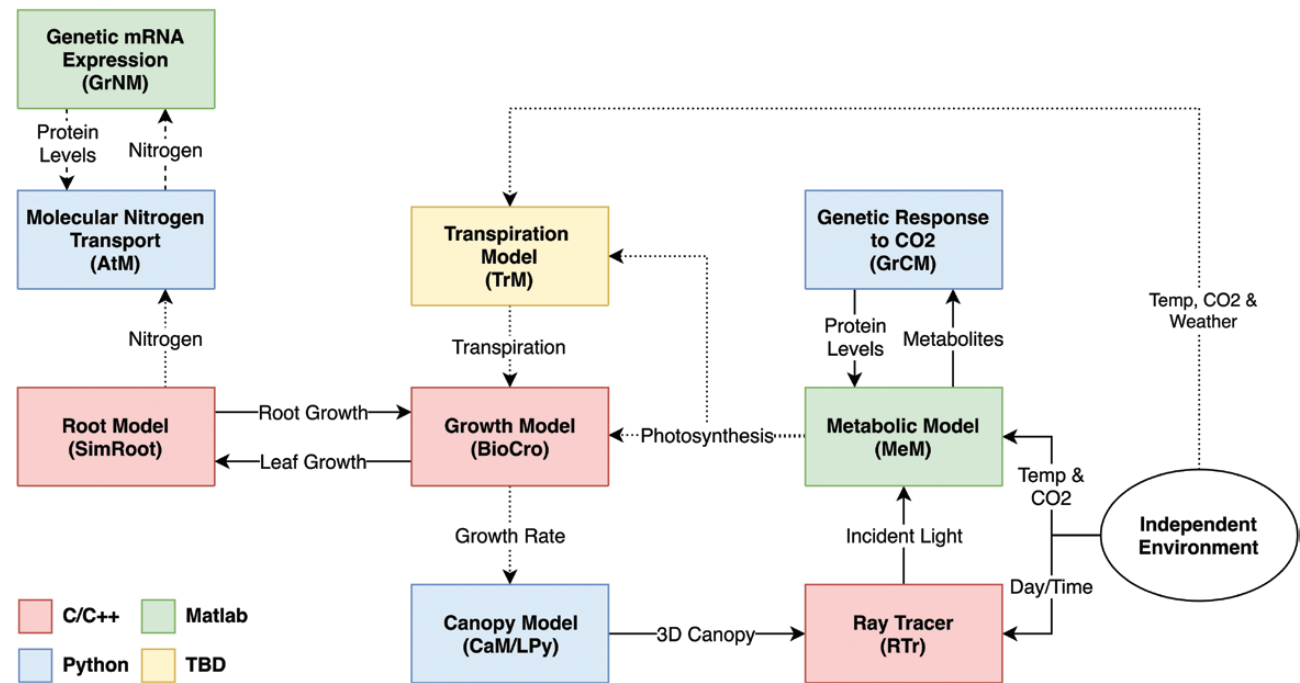**Fig. 11.** Progress towards full Crops in Silico integration network. Each square represents a model and the colour of the square indicates the language in which the model is written. Arrows between models represent connections being made between the models using yggdrasil. Solid lines are connections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

implemented, dashed lines are connections currently being implemented and dotted lines are connections planned for the future. When complete, the integration network will include nine models of molecular, genetic, cellular, organ and plant level processes in four different programming languages, produced by >10 researchers at four 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 modelling also identified gene candidates predicted to regulate enzymes involved in the light and dark reactions of photosynthesis under elevated CO_2.

In addition to allowing scientists to reuse existing models in new integrations and make new connections, yggdrasil has the potential to significantly increase a model's accessibility and promote reuse. yggdrasil could be used as 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 yggdrasil. These additional materials would be particularly powerful when considered alongside the interactive features being developed for use with yggdrasil (see Graphical user interface and Models as functions sections). Potential collaborators could use yggdrasil to run and explore models without the need to be familiar with the inner workings of the model implementation or even implementation language.

Advantages and limitations

Advantages. yggdrasil offers several advantages over manual integrations or other existing tools.

Dynamic communication network: While there are many existing tools for communication that have implementations in multiple languages (some of which yggdrasil uses, e.g. Gabriel *et al.* 2004; RabbitMQ 2007; ZeroMQ 2007~~, ZMQ, RabbitMQ and OpenMPI~~), these tools require that the communication pattern is well established such that it can be hard coded into the model (or model wrapper) itself (e.g. using specific ports or queues). The use of hard coded values greatly restricts the reusability of a model. For every integration a model is used in, the hard coded values for every connection must be updated between the models on either end so that they match. yggdrasil is unique in that it dynamically establishes and manages the networks of communication resources (e.g. queues, channels and sockets) that are required for integrations to run so that models do not need to have these hard coded value and can be reused in multiple integrations, unmodified. This process includes creating new resources, connecting to existing resources, passing the appropriate resource addresses to the model environment, error handling/propagation and resource cleanup.

Hands-off parallelism and asynchronous communication: In addition to inter-language communication, yggdrasil users benefit from speedups due to parallelism and concurrency without any additional work. From the specified model connections, yggdrasil manages the parallel execution of models and asynchronous message passing under-the-hood so that the user does not have to think about it. Existing tools for asynchronous communication between unmanaged parallel tasks (e.g. Jack2, Magoulès and Gbikpi-Benissan 2018) requires explicit definition of the communication pattern in advance, prohibiting dynamic communication networks as discussed above. Work/data flow tools like Parsl (Babuji *et al.* 2018) allow dynamic execution of processes along a dependency chain in parallel, but usually require additional code, are limited to passing data as files or data object native to the tool's implementation language and are ill-suited to models with persistent internal states or intermediate communication. yggdrasil is unique in that it offers users asynchronous communication and parallelism that is flexible enough to support any communication pattern, implicitly handles conversion between native data types in the supported languages, and does not require any additional time investment on the part of the user to achieve a speedup.

No new language: Unlike DSLs or language specific frameworks, yggdrasil does not require a model to be written in one specific language. In addition to expanding the possible number of model combinations, this eases the burden on model writers for whom learning an additional programming language or DSL might be a significant obstacle. yggdrasil is designed such that model writers only need to be familiar with the language that the model is written in to ~~being~~ with. Although yggdrasil is a Python package, users do not even need to be familiar with the Python language and can interact with yggdrasil solely through the command line. While users do need to be able to compose YAML specification files in addition to their model code, these files have a very limited syntax and will be GUI composable in future releases.

Limitations. While yggdrasil is a powerful tool for integrating scientific models and opening up new avenues for collaboration, it is not applicable to every scenario. Before diving into model integration, potential users should determine whether their use case falls under one of the following scenarios.

Single language integrations: For models written in the same language, it is usually much more computationally efficient to either integrate the models directly (i.e. by calling one from the other) or use a language-specific tool for parallel execution and passing data (e.g. Parsl in Python, Babuji *et al.* 2018) than to use yggdrasil. This performance advantage will be particularly pronounced for models that finish quickly or tightly coupled integrations where models make a large number of exchanges in rapid succession (e.g. a set of coupled ODEs). Although the constituent models will still benefit from integration via yggdrasil in regards to reusability, yggdrasil may not be the best choice in such cases if execution time is a concern. Future work on automated creation of wrapper files for function-like (see Wrapper automation and control flow section) and ODE (see the Symbolic ODE models section) models may allow yggdrasil to handle this automatically, replacing managed communication with in-language variable passing for models that are written in the same language and co-located on the same machine.

Fast and tightly coupled models: For models involving quick calculations or for tightly coupled sets of models such as might occur for two models representing coupled ODEs, the computational cost of rapid communication via yggdrasil can quickly surpass the cost of the calculations themselves, causing the yggdrasil to be computationally inefficient in comparison with a manual integration. For example, if the sleep statements were removed from the example root and shoot models discussed in the Worked example section, the calculations become trivial and take significantly less time than the communication between the models. Without the sleep statements, the root and shoot models run for 100 time steps in 0.007 and 0.095 s, respectively, while the integration takes 7.437 s. The speedup offered by yggdrasil is then 0.0137, which is actually a *significant* slow down. For such models, large parameter searches or integrations with many time steps can be computationally prohibitive with yggdrasil.

Unsupported languages: While yggdrasil currently supports models written in Python, Matlab, C and C++, these languages are not an exhaustive list of those languages in which models are written. There are plans to expand support for yggdrasil to other languages (see the Language support section), but in the meantime yggdrasil will not be particularly useful for models that are not written in one of the languages already supported. However, if a language can be called from one of the supported languages or vice versa (e.g. f2py or SWIG, Beazley 2003; Peterson 2009), it may be possible to either write a wrapper or make calls to an interface in a supported language.

Ongoing/future improvements

yggdrasil 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 yggdrasil.

Language support. The biggest barrier to running new models is language support as 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. Informal surveys of the plant modelling community during recent conferences and workshops have helped identify the core languages being used by model developers that would have the largest impact on the number of models accessible via yggdrasil. On the basis of these results, we plan to expand yggdrasil to support models written in R, Fortran, Java and the SBML (Hucka *et al.* 2003) DSL during the next major development push with support for additional models in subsequent releases as requested.

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 modellers 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 modellers ability to collaborate without worrying about access to a Matlab license.

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 yggdrasil support for running integration networks on distributed compute resources. yggdrasil already uses communication tools that can

be adapted for use in a distributed pattern (ZMQ, RMQ; ZeroMQ 2007; RabbitMQ 2007) and 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. This work will also enable yggdrasil to adaptively ~~spawning~~ additional model processes on elastic compute resources when a model becomes a bottle neck to the integration process. This feature will be particularly useful for integrations simulating large ensembles of biological units (e.g. leaves on a plant or plants in a field) and will enable scientists to explore emergent properties of these systems.

In addition, we will also add tools to yggdrasil for running models as a service and using RMQ to permit access to these models within integration networks. This feature will be useful for scientists who are unable or unwilling to share the source code for their model. Instead, these models can be hosted on dedicated compute resources managed by the model's home institution and collaborators interested in using the model in an integration can connect to it remotely, submitting requests and receiving results as needed.

Wrapper automation and control flow. Currently, modellers must explicitly decide how a model will be used when writing the model wrapper. These decisions can include things like which input and output variables are static versus ~~variable~~, if there are sets of variables that should be updated at the same time, and how often output is generated, resulting in a hard coded use pattern. If someone wishing to use the model in another integration requires a different use patter, they will need to write their own wrapper. However, for models that have a functional form, the process of writing the model wrapper can be automated and done dynamically for each integration. We plan on adding options to yggdrasil for dynamically generating model wrappers based a user-provided function call and list of static/variable input channels. This feature will improve the reusability of model code such as during parameter studies and decrease the amount of work required for integrating models that are written or can be exposed in a functional form.

Dynamic generation of model wrappers also opens up possibilities for managing the flow of data through an integration network from the YAML. As part of the wrapper automation feature, we will add options for users to specify conditional execution of models via the YAML specification file. For example, a user may have two models for the same process that are valid under different conditions. This feature will allow users to declare this conditional in the YAML such that yggdrasil automatically switches between the two models as appropriate without the need to write an additional wrapper encoding the conditional.

Symbolic ODE models. Many biological models can be posed as a series of coupled ODEs (Wang *et al.* 2015). The equations describing such models can easily be 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, Hairer *et al.* 1993; Jones *et al.* 2001). Using these tools, we will add features to yggdrasil that allow ODE models to be constructed directly from symbolic representations provided in YAML specification files. These features will allow users to quickly prototype new models and integrate old ones without the need to write a wrapper.

Symbolic integration of ODE-based models also has advantages for performance. Integrating models containing coupled ODEs is currently one of the hardest problems for yggdrasil to tackle due to the performance cost associated with rapid communications between models (see Limitations section). However, symbolic ODE representations can be used to represent two coupled ODE models as a single model which is much more computationally efficient as it does not require communication via yggdrasil. As such, ODE support for yggdrasil will also include automated coupling at runtime such that, while two ODE models can be declared separately and used independently in integrations, they will be combined into a single code when connected. This feature will greatly improve the performance of yggdrasil for integration of coupled ODEs.

Data aggregation and forking. Much of the original version of yggdrasil centred around the use of tabular or groups sets of input data where multiple inputs to a model are updated simultaneously such as during a loop. While this type of input is used heavily by plant models, it presents several barriers for constructing integration networks. Simultaneously, updating multiple variables requires that either all of the variables are received from the same source (e.g. a row in a table with inputs taken from each column) or that the variables are updated simultaneously from multiple sources. The first case, while useful for input from a correctly formatted table, does not work for integration unless it is between two models where one model outputs exactly the same variables that are expected by the other. However, this is unlikely to be the case if two models are developed independently. The second case is also not ideal as it makes receiving and updating input variables tedious for more than a few variables as each receive call has to be checked independently for errors.

Future improvements to yggdrasil will allow input and output variables to be grouped based on when they will be received or sent within a model while remaining independent connections internally to yggdrasil. In this way, grouped inputs/outputs can be composed of variables received/sent from /to different sources, but will be received/sent simultaneous by the model. This feature will reduce the number of send/receive calls used within a model wrapper, streamline the error checking process, and greatly simplify the model wrappers that user need to write.

In addition to aggregation, this line of improvements will also add features to yggdrasil to allow connections to receive/send from/to multiple sources. Connection forking will enable users to duplicate data among several destinations, including files, allowing intermediate outputs to be recorded along the way. It will also allow single models to process input from multiple source models such as in the case of using multiple instances of a model to speed up performance of rate limiting integration step. Data aggregation and forking will be part of the yggdrasil 1.0 release.

JSON data type specification. yggdrasil currently supports serialization of several data types/formats, but support for new data types must be added manually and can prevent or slow down the integration of new models. Therefore, expanding the flexibility of yggdrasil serialization will allow a greater variety of models to be connected and is a priority for future development. yggdrasil will be adapted to read JSON schema (JSON Schema 2018) for user-defined types that can be used for automatically creating the appropriate data structures alongside methods for parsing and serializing them.

In addition to using JSON schema for adding new data types, JSON schema will be developed for validating YAML files and existing data types will be transitioned to a JSON-friendly representation. As there are many existing tools in different languages for encoding/decoding JSON (e.g. FSON, jsonlite, Ooms 2014; Levin and Croucher 2019), this will greatly decrease the amount of new development required to add support to yggdrasil for new languages. JSON serialization will be part of the yggdrasil 1.0 release.

Graphical user interface. In an effort to make yggdrasil as user-friendly as possible, the Crops in Silico group at the University of Illinois at Urbana-Champaign has 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, the ultimate goal is to provide the GUI as 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.

Models as functions. In addition to a dedicated GUI, tools are also being developed for using yggdrasil 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 yggdrasil that can be used by those unfamiliar with modelling or the model's particular programming language to explore the model's behaviour.

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 such as 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 yggdrasil 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 yggdrasil 1.0 release.

Code

The yggdrasil package is available publicly on Github and can be installed using pip or conda. To ensure code health, yggdrasil 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 (tr___ 2018) and Appveyor (___)

2018). Full documentation for yggdrasil can be found here including step-by-step directions from the tutorial conducted during the 2018 Crops in Silico Hackathon.

Sources of Funding

Acknowledgements

Conflict of Interest

None declared.

Literature Cited

1994. *B1. Object Files (.obj)*, Technical report.

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

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

2007. ZeroMQ. http://zeromq.org/

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, Linux and 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, deploy your code with confidence and 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 *International Workshop on Science Gateways*

Beazley D. 2003. *Future Generation Computer Systems* **19**, 599.

Behnel S, Bradshaw R, Citro C. 2011. *Computing in Science and Engineering* **13**, 31.

Ben-Kiki O, Evans C, dot Net I. 2009. *YAML Ain't Markup Language (YAMLTM) version 1.2 YAML Ain't Markup Language (YAMLTM) Version 1.2, 3rd Edition, Patched at 2009-10-01, Technical report.*

Bernacchi CJ, Morgan PB, Ort DR, Long SP. 2005. *Planta* **220**, 434.

Boudon F, Pradal C, Cokelaer T, Prusinkiewicz P, Godin C. 2012. *Frontiers in Plant Science* **3**, 76.

Coakley S, Gheorghe M, Holcombe M, et al. 2012. in 2012 IEEE 9th Int. Conf. Embed. Softw. Syst. (IEEE), 538–45.

Cuellar AA, Lloyd CM, Nielsen PF, et al. 2003. *Simulation* **79**, 740.

Demir E, Cary MP, Paley S, et al. 2010. *Nature Biotechnology* **28**, 935.

Eaton JW. 2002. *GNU octave manual.* Network Theory Limited.

Gabriel E, Fagg GE, Bosilca G, et al. 2004. Proceedings, 11th Eur. PVM/MPI Users' Gr. Meet. Budapest, Hungary. Berlin, Heidelberg: Springer, 97–104.

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

Goldbaum NJ, Zuhone JA, Turk MJ, Kowalik K, Rosen AL. 2018. *Journal of Open Source Software* **3**, 809

Goudriaan J, Laar HV. 1994. *Modelling potential crop growth processes*. Dordrecht: Kluwer Academic Publishers, 238.

Hairer EE, Nørsett PSPS, Wanner G. 1993. *Solving ordinary differential equations. I. Nonstiff problems.* 528.

Hall AJ, Minchin PEH. 2013. *Plant, Cell and Environment* **36**, 2150.

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

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

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

Jones E, Oliphant T, Peterson, P. 2001. 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., 113–20

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

Kappas M, Degener J, Bauboeck R. 2013. BioSTAR, a new biomass and yield modeling software. American Geophysical Union, Fall Meeting 2013. *AGU Fall Meeting Abstracts.* Abstract id. GC33A-1088.

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

Levin JA, Croucher, A. 2019. Fortran 95 JSON Parser. https://github.com/josephalevin/fson

Magoulès F, Gbikpi-Benissan G. 2018. *Advances in Engineering Software* **119**, 116.

Marshall-Colon A, Long SP, Allen DK, et al. 2017. *Frontiers in Plant Science* **8**, 786.

Martin K, Hoffman B. 2006. *Mastering CMake: a cross-platform build system*, 3rd edn. Kitware.

Merks RMH, Guravage M, Inzé D, Beemster GTS. 2011. *Plant Physiology* **155**, 656.

Meurer A, Smith CP, Paprocki M. et al. 2017. *PeerJ - Computer Science Journal* **3**, e103.

Ooms J. 2014.

Peterson P. 2009. *International Journal on Computer Science and Engineering* **4**, 296.

Postma JA, Kuppe C, Owen MR, et al. 2017. *New Phytologist* **215**, 1274.

Pradal C, Boudon F, Nouguier C, Chopard J., Godin C. 2009. *Graphical Model* **71**, 1.

AQ14 Pradal C, Fournier C, Valduriez P, Cohen-Boulakia S. 2015. ~~Proc. 27th Int. Conf. Sci. Stat. Database Manag. SSDBM '15,~~ New York, NY: ACM Press.

AQ15 Rusling DA. 1999. *The Linux Kernel*, ~~0th edn.~~

Sharkey TD. 2016. *Plant, Cell and Environment* **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. *Functional Plant Biology* **40**, 108.

AQ16 Stallman R, McGrath R, Smith PD. 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, Technical report.* ~~The Board of Trustees of The Leland Stanford Junior University.~~

Wang D, Jaiswal D, Lebauer DS, ~~et al.~~ 2015. *Plant, Cell and Environment* **38**, 1850.

Wang Y, Long SP, Zhu X-G. 2014. *Plant Physiology* **164**, 2231.

Zhu X-G, Wang Y, Ort DR, Long, SP. 2013. *Plant, Cell and Environment* **36**, 1711.

Appendix. Algorithms

Algorithm 4. Generic model driver class. This algorithm does not represent the actual implementation of any one of the languages supported by yggdrasil. 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 behaviour pattern upon which every model driver is based.

```
 1: class ModelDriver (name, args, yml)
 2:    errors ← []
 3:    executable ← Compile(args)
 4:    process ← ModelProcess on a new process
 5:    thread ← ModelThread on a new thread

 6:    method ModelProcess()
 7:       Executable
 8:    end method

 9:    method ModelThread()
10:       while IsAlive(ModelProcess) do
11:          if HasOutput(ModelProcess) then
12:             Print GetOutput(ModelProcess)
13:          else
14:             Sleep
15:          end if
16:       end while
17:       if HasErrors(ModelProcess) then
18:          errors ← GetErrors(ModelProcess)
19:       end if
20:    end method

21:    method Terminate()
22:       Kill(ModelProcess)
23:    end method
```

```
24:    method Stop()
25:       while not errors do
26:          Sleep
27:       end while
28:       Terminate
29:    end method
30: end class
```

Algorithm 5. Basic steps executed on the master thread to create, start and monitor model and connection drivers. This pseudocode is not an exhaustive reproduction of the entire implementation, but represents the broad strokes needed to understand the purpose and behaviour of the master thread. The algorithm for the ModelDriver and ConnectionDriver can be found in Algorithms 4 and 8, respectively.

```
 1: models ← Parse YAML files
 2: inputChannels ← Map()
 3: outputChannels ←          ▷ Create Drivers
    Map()
 4: for all minmodels do
 5:    for all iinm.inputs do
 6:       if not i.isFile then
 7:          inputAddress ← i.address
 8:       else
 9:          inputAddress ← 'generate'
10:       end if
11:       i.driver ← ConnectionDriver(inputAddress,
          'generate', i)
12:       inputChannels[i.name] = i
13:    end for
14: end for
15: for all minmodels do
16:    for all o ∈ m.outputs do
17:       if o.isFile then
18:          outputAddress ← o.address
19:       else
20:          if o.args ∈ inpurChannels then
21:             outputAddress ← inputChannels
                [o.args]
22:          else
23:             Error
24:          end if
25:       end if
26:       o.driver ← ConnectionDriver('generate',
          outputAddress, o)
27:       outputChannels[o.name] = o
28:    end for
29: end for
30: for all minmodels do
31:    m.driver ← ModelDriver(m.name, m.args, m)
32: end for                   ▷ Start Drivers
```

```
33:  for all minmodels do
34:    for all iinm.inputs do
35:      i.driver.Start
36:    end for
37:  end for
38:  for all minmodels do
39:    for all o ∈ m.outputs do
40:      o.driver.Start
41:    end for
42:  end for
43:  for all minmodels do
44:    m.driver.Start
45:  end for
                          ▷ Monitor Drivers
46:  errorFlag ← false
47:  while not errorFlag do
48:    for all minmodels do
49:      if m.driver.errors then
50:        errorFlag ← true
51:        break
52:      end if
53:    end for
54:  end while
                          ▷ Stop Drivers
55:  if errorFlag then
56:    allErrors ← []
57:    for all minmodels do
58:      m.Terminate
59:      Append allErrors, m.errors
60:    end for
61:    Error(allErrors)
62:  else
63:    for all minmodels do
64:      m.Stop
65:    end for
66:  end if
```

Algorithm 6. Asynchronous communication class for sending messages to an output channel. *x* is a communication object implemented by an external communication package (one of the three supported by yggdrasil). The package must have at minimum methods NewAddress, Open, Close, Send and IsOpen for interacting with the object *x*. Encode is dependent on the message type and will use an extended version of JSON encoding in the future (see Data Type Specification section).

```
1:  class AsyncSender(address)
2:    backlog ← []
3:    temp ← []
4:    if address =='generate' then
5:      address ← NewAddress
6:    end if
7:    x ← Open(address)
8:    Run BacklogLoop on a new thread
9:    method BacklogLoop()
10:     while x.IsOpen do
11:       if length(backlog) > 0 then
12:         flag ← x.Send(backlog[0])
13:         if flag then
14:           Remove backlog[0]
15:         else
16:           x.Close
17:         end if
18:       else
19:         Sleep
20:       end if
21:     end while
22:   end method
23:   method SendBacklog(msg)
24:     if x.IsOpen then
25:       Append msg to the end of backlog
26:     end if
27:     Return x.IsOpen
28:   end method
29:   method SendMultipart(msg)
30:     currPos ← 0
31:     while currPos < length(msg) do
32:       ilength ← min(x.maxMsgSize,
                length(msg) – currPos)
33:       Send(msg[currPos : (currPos +
                ilength)])
34:       currPos+ = ilength
35:     end while
36:   end method
37:   method Send(msg)
38:     header ←        ▷ Additional informa-
         Map()            tion will be added to
                          the header based on
                          communication mecha-
                          nism and encoding.
39:     totalMsg ← Encode(msg, header)
40:    if length(totalMsg) > x.maxMsgSize then
41:       newTemp ← AsyncSender('generate')
42:       Append newTemp to the end of temp
43:       header['workerAddress']
           ← newTemp.address
44:       totalMsg ← Encode(msg, header)
45:       if not SendBacklog(totalMsg[:
           x.maxMsgSize]) then
46:         Return false
47:       end if
48:       if not newTemp.
           SendMultipart(totalMsg[x.maxMsgSize
           :]) then
49:         Return false
```

```
50:       end if
51:       Return true
52:     else
53:       Return SENDBACKLOG(totalMsg)
54:     end if
55:   end method
56:   method CLOSE(dontWaitForRecv = false)
57:     if x.IsOPEN then
58:       SEND(EOF)
59:     end if
60:     if not dontWaitForRecv then
61:       while x.IsOPEN and ((length(backlog)
             > 0) or x.IsMESSAGEWAITING) do
62:         Sleep
63:       end while
64:     end if
65:     x:CLOSE
66:     backlog ← []
67:     for all t ∈ temp do
68:       t.CLOSE(dontWaitForRecv)
69:     end for
70:   end method
71: end class
```

Algorithm 7. Asynchronous communication class for receiving messages from an input channel. x is a communication object implemented by an external communication package (one of the three supported by yggdrasil). The package must have at minimum methods NewAddress, OPEN, CLOSE, RECV, IsOPEN, and IsMESSAGEWAITING for interacting with the object x. Decode is dependent on the message type and will use an extended version of JSON decoding in the future (see 🗨 Data Type Specification section).

```
1:  class ASYNCRECEIVER(address)
2:    backlog ← []
3:    temp ← []
4:    if address == 'generate' then
5:      address ← NewAddress
6:    end if
7:    x ← OPEN(address)
8:    thread ← BACKLOGLOOP on a new thread
9:    method BACKLOGLOOP()
10:     while x.IsOPEN do
11:       if x.IsMESSAGEWAITING then
12:         flag, msg ← x.RECV
13:         if flag then
14:           Append msg to the end of
               backlog
15:         else
16:           x.CLOSE
17:         end if
18:       else
19:         Sleep
20:       end if
21:     end while
22:   end method
23:   method RecvBacklog(msg)
24:     if length(backlog) > 0 then
25:       msg ← backlog[0]
26:       Remove backlog[0]
27:       Return true, msg
28:     else
29:       Return x.IsOPEN, ''
30:     end if
31:   end method
32:   method RECVMULTIPART(msgSize)
33:     msg ← ''
34:     while length(msg) < msgSize do
35:       flag, msgPart ← RECV
36:       if not flag then
37:         Return flag, msg
38:       end if
39:       msg ← msg + msgPart
40:     end while
41:     Return true, msg
42:   end method
43:   method RECV()
44:     flag, msgStart ← RECVBACKLOG
45:     if not flag then
46:       Return flag, ''
47:     end if
48:     header ← DECODEHEADER(msgStart)
49:     if 'workerAddress' ∈ header then
50:       newTemp ← ASYNCRECEIVER(header
             ['workerAddress'])
51:       Append newTemp to the end of temp
52:       flag, msgRemain ← newTemp.RECVMULTIPART
53:       if not flag then
54:         Return flag, ''
55:       end if
56:       header, msg ← DECODE(msgStart +
             msgRemain)
57:     else
58:       header, msg ← DECODE(msgStart)
59:     end if
60:     Return true, msg
61:   end method
62:   method CLOSE()
63:     x.CLOSE
64:     backlog ← []
65:     for all t ∈ temp do
66:       t.CLOSE
67:     end for
68:   end method
69: end class
```

30.5 30.10 30.15 30.20 30.25 30.30 30.35 30.40 30.45 30.50 30.52 30.55 30.60 30.65 30.70 30.75 30.80 30.85 30.90 30.95 30.100 30.104

Algorithm 8. Connection driver

```
 1: class ConnectionDriver(inputAddress,
        outputAddress, yml)
 2:    input ← AsyncReceiver(inputAddress)
 3:    output ← AsyncSender(outputAddress)
 4:    thread ← ConnectionLoop on a new thread

 5:    method ConnectionLoop()
 6:      while true do
 7:        flag, msg ← input.RecvBacklog
 8:        if not flag then
 9:          break
10:        end if
11:        if msg then
12:          Transform msg based on units and
              expected data format
13:          flag ← output.SendBacklog(msg)
14:          if not flag then
15:            break
16:          end if
17:        else
18:          Sleep
19:        end if
20:      end while
21:      input.Close
22:      output.Close
23:    end method

24:    method Terminate()
25:      input.Close
26:      output.Close(dontWaitF orRecv = true)
27:    end method

28:    method Stop()
29:      while input.x.IsOpen and
            output.x.IsOpen do
30:        Sleep
31:      end while
32:      Terminate
33:    end method

34: end class
```

31.5

31.10

31.15

31.20

31.25

31.30

31.35

31.40

31.45

31.50

31.52

31.55

31.60

31.65

31.70

31.75

31.80

31.85

31.90

31.95

31.100

31.104