Draft version September 30, 2018
Preprint typeset using LaTeX style emulateapj v. 12/16/11

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

Meagan Lang[1]

Draft version September 30, 2018

ABSTRACT

Thousands of computational models have been created within the plant biology community within the past two decades that have the potential to be combined into complex integration networks capable of capturing more complex biological processes than possible as isolated models. However, the technological barriers introduced by differences in language and data formats as slowed this progress. We present `cis_interface`, a Python package for running integration networks with connections between models across languages and scales. `cis_interface` coordinates parallel execution of models in Python, C, C++, and Matlab and handles communication in a number of data formats common to computational plant modeling. `cis_interface` is designed to be user-friendly, including visual tools for composing the integration networks.

1. INTRODUCTION

Computational plant biologists have produced a wealth of computational models for different biological processes governing plant growth and development, covering scales from atomic to global. Although usually developed independently to address questions specific to an organ, species, or growth process, computational plant models can also have applications beyond their original scope. For example, a photosynthesis model developed for maize can also be adapted to work for other species like soy cite Kavya and Yu. Computational models for different scales, organs, or processes also have the potential for being combined to probe more complex biological mechanisms. With advances in compute power, it should be possible to link biologically related models to create 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.

We present an Open Source Python package, `cis_interface`, for creating and running integrations networks from existing computational models written in Python, Matlab, C and C++. Although developed for the purpose of integrating plant biology models, `cis_interface` can be applied to any situation requiring coordination between models in the supported languages. §2 provides background information on existing efforts for cross-language model integration, §3 describes the methods used by the `cis_interface` package to integrate models, §4 presents several tests demonstrating the performance of message passing between integrated models, and §5 summarizes the features of the `cis_interface` package, describes a few use cases, and outlines areas for future development.

2. BACKGROUND

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

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

These three questions can be addressed through manual integration on the command line or using scripting (e.g. running model A, converting output from A into input expected by model B, running model B). However, manual integration is 1) computational inefficient when many iterations are necessary, such as during parameter searches or steady state convergence tests, 2) unique to every integration, requiring the production of new scripts, and 3) complex when more than > 3 models are being integrated. Within both computational biology and the larger scientific computing community, groups have developed around different solutions to the problem of connecting models.

2.1. *Domain Specific Language (DSL)*

One solution is to agree upon a language. The questions of orchestration, communication, and translation become trivial when models are all written in the same language. Beyond selecting a single programming language, many communities have developed dedicated domain specific languages (DSLs) for model representa-

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

tion. For example, the Systems Biology Markup Language (SBML Hucka et al. 2003) is an XML-based format for representing models of biological processes. Scientists can compose their models using SBML markup and then run their model using software designed to parse SBML files. Other DSLs for biological models include LPy (Boudon et al. 2012), CellML (Cuellar et al. 2003), and BioPAX (Demir et al. 2010). Dedicated model DSLs improve the reusability of models written in the DSL and have the advantage of often implicitly handling data transformation. However, they do nothing for existing models that are not in the DSL or cannot be expressed within the constraints of the DSL. In addition, learning a new DSL, in addition to a programming language, can be daunting for new researchers.

2.2. *Workflow/Data Flow/Framework*

Another approach is a workflow/data flow/framework tools for models that are written in the same programming language or can be wrapped. For example, there are many generic tools in Python for coordinating the execution of tasks in parallel or serial (e.g. Babuji et al. 2018; cel 2018; lui 2018). While these tools are powerful for organizing complex work flows, it is the user's onus to make sure that the components they connect are compatible and handle an sort of 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 visualizations. Domain specific frameworks are more intuitive and ultimately more flexible than DSLs in the types of operations they allow models to perform since they have access to the full power of a programming language. However, they are stricter in several respects: 1) the model must be written in (or exposable to), the language of the framework and 2) the model must be written in a way that is aware of the framework and/or the format of models with which it will interact. Like DSLs, frameworks also provided limited support for models written without them in mind.

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

3. METHODS

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

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

`cis_interface` currently support models written in Python, Matlab, C, and C++ with additional Domain Specific Language (DSL) support for LPy models (Boudon et al. 2012).

3.1. *Orchestration*

3.1.1. *YAML Specification*

Users specify information about models and integration networks via declarative YAML files (Ben-Kiki et al. 2009). The YAML file format was selected because it is human readable and there are many existing tools for parsing YAML formats in many different languages. The declarative format allows user to specify exactly what they what to do, without describing how it should be done. While the information about models and integration networks can be contained in a single YAML file, the information can naturally be split between two files, one containing information about the models 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 type and units. Integration networks are specified via YAML files declaring the connections between models. Connections are declared by pairing an output variable from one model with the input variable of another model. The `cis_interface` command line interface (CLI) sets up the necessary communication mechanisms to then direct data from one model to the next in the specified pattern. Models can have as many input and/or output variables as is desired. Connections between models are specified by references to the input/output variables associated with each model. This format allows users the flexibility to create complex integration networks.

3.1.2. *Model Drivers*

Model drivers handle model execution, monitoring, and compilation if necessary. While every model is executed on a new process, how the model is handled depends on the language it is written in. Models written in interpreted languages (Python and Matlab) are executed on the command line with the interpreter. In the case of Matlab, where there is significant overhead associated with starting the Matlab interpreter, a Matlab shared engine is used to execute the model. To speed up the execution, the shared engine can be started in advance and then reused.

For the compiled languages (C and C++) there are a few options. The user can compile the model themselves, provided they link against the appropriate `cis_interface` header library. Alternatively, users can provide the location of the model source code and let `cis_interface` handle the compilation, including linking against the appropriate `cis_interface` header library. `cis_interface` also has support for compiling models using Make (Stallman et al. 2004) and CMake

(Martin & Hoffman 2006) for models that already have a Makefile or CMakeLists.txt. To use these drivers, lines are added to the recipe in order to allow linking against `cis_interface`.

3.2. *Communication*

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

3.2.1. *Asynchronous Message Passing*

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

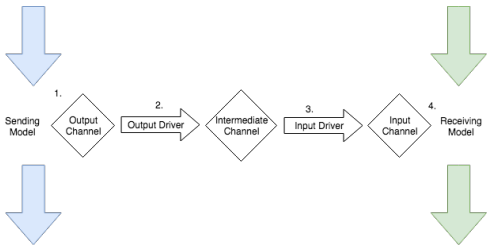

FIG. 1.— Diagram of how messages are passed asynchronously using input/output drivers and an intermediate channel.

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

In additional to communication between two models, users can also specify that a model should receive/send input/output from/to a file. This is specified in the integration network YAML and does not impact the way the model will receive/send messages.

3.2.2. *System V IPC Queues*

The first communication mechanism used by `cis_interface` was System V interprocess communication (IPC) message queues (Rusling 1999) on Posix (Linux and Mac OSX) systems. IPC message queues allow messages to be passed between models running on separate processes on the same machine. While IPC message queues are light weight, fast (See §4), and are part of Posix operating systems, they do not work in all situations. Sys V IPC queues are not natively supported by Windows operating systems and do not allow communication between remote processes. In addition, IPC queues also have relatively low default message size limits on Mac OSX systems (2048 bytes). While this can be handled by splitting large messages into multiple smaller messages (See §3.2.5), the time required to send a message increases with the number of message it must be broken into (See §4). As a result, Sys V IPC queues are considered by `cis_interface` to be a fallback on Posix systems if the necessary ZeroMQ libraries have not been installed.

3.2.3. *ZeroMQ*

The preferred communication mechanism used by `cis_interface` are ZeroMQ sockets (Akgul 2013). ZeroMQ provides broker-less communication via a number of protocols and patterns with bindings in a wide variety of languages that can be installed on Posix and Windows operating systems. ZeroMQ was adopted by `cis_interface` in order to allow support on Windows and for future target languages (See §5.2) that could not be accomplished using System V IPC queues. In addition, while ZeroMQ allows interprocess communication via IPC queues like System V IPC queues, ZeroMQ also supports protocols for distributed communication via an Internet Protocol (IP) network. While `cis_interface` does not currently support using these protocols for distributed integration networks, this one of the plans for future improvement.

3.2.4. *RabbitMQ*

While ZeroMQ provides broker-less communication means, `cis_interface` includes support for brokered communication via RabbitMQ (RMQ 2007). `cis_interface` does not currently use RabbitMQ for communication unless explicitly specified by the user in their integration network YAML. RabbitMQ support was originally added to `cis_interface` as a supplement to System V IPC queues in the case of future support for distributed integration networks. In future development,

RabbitMQ brokered communication will be used for establishing integration networks with remote models run a services (See §5.2).

3.2.5. *Sending/Receiving Large Messages*

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

3.2.6. *Interface*

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

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

3.3. *Transformation*

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

- Scalars (e.g. integers, floats)
- Arrays
- Tabular (e.g. CSV or tab-delimited)
- Pandas data frames
- PLY
- OBJ

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

4. RESULTS

In order to evaluate `cis_interface` performance tests were run on Linux, Mac OSX, and Windows for different communication mechanisms and language combinations. During each run, N_{msg} messages of size S_{msg} were sent from the source model (in one language) to the destination model (in another language) which then output the messages to file for verification. Performance tests were run using the perf package (Stinner 2018). Each run was repeated 20 times to rule out fluctuations due to external loads on the test machines.

4.1. *Communication Mechanism*

Figure 2 compares the execution times for the different communication mechanisms discussed in §3.2 for sending different numbers/sizes of messages from one Python model to another Python model. The tests were run on a Dell? tower with 10 quad-core Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz processors running Ubuntu 14.04. The left panel of Figure 2 shows the total time required to run both models and send a varying number of 1000 bytes messages from one model to the other. The right panel of Figure 2 shows the total time required to run the models and send 5 messages of varying lengths from one to the other. Although Sys V IPC queues are

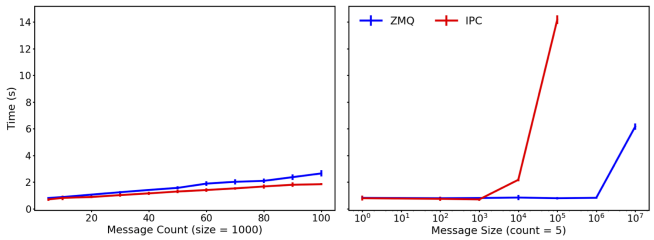

FIG. 2.— Comparison of communication time scaling across communication methods on Linux. Left: Scaling of execution time with number of 1000 byte messages sent. Right: Scaling of execution time with the size of the 5 messages sent.

slightly faster than ZeroMQ TCP sockets (0.012 s/msg

vs. 0.019 s/msg), IPC queues have a much smaller maximum size limit for messages. As discussed in §3.2.5, messages larger than this limit (2048 bytes) must be broken up into multiple messages. This compounds the time required to send these messages making IPC queues a poor choice for sending data over 10^5 bytes. This limit does not kick in for ZeroMQ sockets until much later ($\sim 10^6$ bytes).

4.2. *Language*

Figure 3 compares the execution times for integrations with communications between models in different combinations of languages. The tests were run on the Linux machine from above using ZeroMQ communication. For

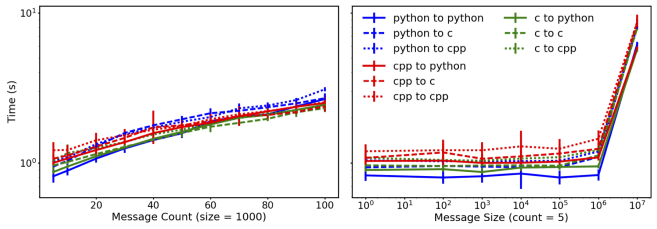

FIG. 3.— Comparison of communication time scaling between languages on Linux.

the most part, performances are not affected by the language of either model. Each language entails some overhead. For Python and Matlab, the overhead comes from starting the interpreter. For C and C++, the overhead comes from compiling the source code. results for Matlab

4.3. *Python Version*

Figure ?? compares the execution times when using different versions of Python. The performance tests for both Python 2.7 and Python 3.5 were performed on the Linux machine from above using ZeroMQ communication. figure and results

4.4. *Platform*

Figure ?? compares the execution times on different platforms. The performance tests reported for Linux were run on the machine from above. The tests for Mac OS X were run on a 2015 MacBook Pro with a 2.9 GHz Intel Core i5 processor running macOS High Sierra 10.13.4. The tests for Windows were run on the same machine from a dual boot of Windows 10. All platform dependent performance tests use ZeroMQ communication. figure and results

5. SUMMARY & DISCUSSION

5.1. *Current Status*

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

`cis_interface` has already been used to integrate several plant models within the Crops in Silico organization with ongoing work to add additional models to the integration network. Figure 4 shows the progress so far. Each square is a model and lines between the models represent connections where information is exchanged. Solid lines are connections we have already implemented, dashed lines are connections we are currently working on, and dotted lines are connections planned for the future. Results from one integration can be found in cite Kavya/Yu. more here? teaser results?

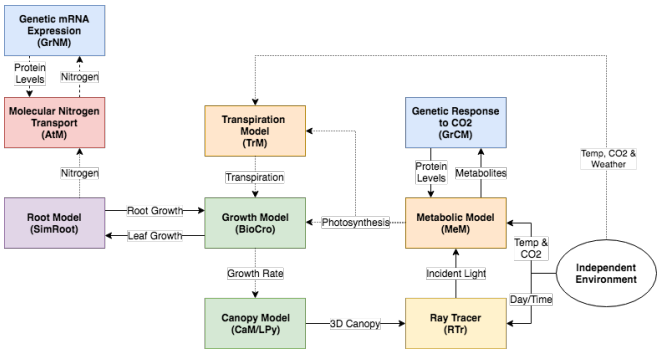

FIG. 4.— Progress towards full Crops in Silico integration network.

5.2. *Current/Future Improvements*

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

5.2.1. *Language Support*

The biggest barrier to running new models is language support. Plant models are written in many different languages. While the current language support covers many, additional languages must be added to unlock the full potential. Surveys of the plant modeling community have helped identify the core languages that models have been written in. Based on these results, we plan to expand `cis_interface` to support models written in R, Fortran, Java, and the SBML (Hucka et al. 2003) DSL. We also plan to add support for running Matlab models using Octave (Eaton 2002). Matlab models require a Matlab license to run. Given that plant modelers do not all use Matlab, it cannot be assumed that everyone will have access to a Matlab license. Octave is open source and provides much of the same functionality as Matlab and can run many Matlab codes. Support for Octave will improve modelers ability to collaborate without worrying about access to a Matlab license.

5.2.2. *Distributed Systems*

High performance computing (HPC) and cloud compute resources are powerful tools in the current computing ecosystem that could be used for running complex

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

5.2.3. *Control Flow*

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

5.2.4. *Data Aggregation*

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

5.2.5. *JSON Data Type Specification*

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

5.2.6. *Graphical User Interface*

In an effort to make `cis_interface` as user friendly as possible, we have also begun work on a graphical user interface (GUI) for entering model information, composing integration networks from an existing palette of models, and displaying basic output from an integration run. The current prototype of the GUI can only handle model ingestion and network composition; model execution must be done locally. However, our ultimate goal with the GUI is to provide a service where users can register their model, compose and run integration networks on dedicated cloud compute resources, and view output in real time all from the web browser. Mike and Craig as authors/acknowledgements? Provide link?

CODE

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

Fix code citations

REFERENCES

2007, RabbitMQ

2018, AppVeyor: Continuous Integration solution for Windows and Linux

2018, Celery

2018, JSON Schema — The home of JSON Schema

2018, Luigi

2018, Travis CI - Test and Deploy Your Code with Confidence

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

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

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

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

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

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

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

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

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

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

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

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

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

Stinner, V. 2018, Python perf module