Proceedings of the 11th Python in Science Conference

July 16–21, 2012 • Austin, Texas

Aron Ahmadia
Jarrod Millman
Stéfan van der Walt

PROCEEDINGS OF THE 11TH PYTHON IN SCIENCE CONFERENCE

Edited by Aron Ahmadia, Jarrod Millman, and Stéfan van der Walt.

SciPy 2012
Austin, Texas
July 16–21, 2012, 2012

ISSN:2575-9752
https://doi.org/10.25080/Majora-54c7f2c8-00d

ORGANIZATION

Conference Chairs

WARREN WECKESSER, Enthought, Inc.
STEFAN VAN DER WALT, UC Berkeley

Program Chairs

MATT MCCORMICK, Kitware, Inc.
ANDY TERREL, TACC, University of Texas

Program Committee

ARON AHMADIA, KAUST
W. NATHAN BELL, NVIDIA
JOHN HUNTER, TradeLink
MATTHEW G. KNEPLEY, University of Chicago
KYLE MANDLI, University of Texas
MIKE MCKERNS, Enthought, Inc.
DAN SCHULT, Colgate University
MATTHEW TERRY, LLNL
MATTHEW TURK, Columbia University
PETER WANG, Continuum Analytics

Tutorial Chairs

DHARHAS POTHINA, Texas Water Development Board
JONATHAN ROCHER, Enthought, Inc.

Sponsorship Chairs

CHRIS COLBERT, Enthought, Inc.
WILLIAM SPOTZ, Sandia National Laboratories

Program Staff

JODI HAVRANEK, Enthought, Inc.
JIM IVANOFF, Enthought, Inc.
LAUREN JOHNSON, Enthought, Inc.
LEAH JONES, Enthought, Inc.

Scholarship Recipients

,
,

Jump Trading and NumFOCUS Diversity Scholarship Recipients

,
,

CONTENTS

Parallel High Performance Bootstrapping in Python

Aakash Prasad‡*, David Howard‡, Shoaib Kamil‡, Armando Fox‡

We use a combination of code-generation, code lowering, and just-in-time compilation techniques called SEJITS (Selective Embedded JIT Specialization) to generate highly performant parallel code for Bag of Little Bootstraps (BLB), a statistical sampling algorithm that solves the same class of problems as general bootstrapping, but which parallelizes better. We do this by embedding a very small domain-specific language into Python for describing instances of the problem and using expert-created code generation strategies to generate code at runtime for a parallel multicore platform. The resulting code can sample gigabyte datasets with performance comparable to hand-tuned parallel code, achieving near-linear strong scaling on a 32-core CPU, yet the Python expression of a BLB problem instance remains source- and performance-portable across platforms. This work represents another case study in a growing list of algorithms we have "packaged" using SEJITS in order to make high-performance implementations of the algorithms available to Python programmers across diverse platforms.

Introduction

A common task domain experts are faced with is performing statistical analysis on data. The most prevalent methods for doing this task (e.g. coding in Python) often fail to take advantage of the power of parallelism, which restricts domain experts from performing analysis on much larger data sets, and doing it much faster than they would be able to with pure Python.

The rate of growth of scientific data is rapidly outstripping the rate of single-core processor speedup, which means that scientific productivity is now dependent upon the ability of domain expert, non-specialist programmers (productivity programmers) to harness both hardware and software parallelism. However, parallel programming has historically been difficult for productivity programmers, whose primary concern is not mastering platform specific programming frameworks. At the same time, the methods available to harness parallel hardware platforms become increasingly arcane and specialized in order to expose maximum performance potential to efficiency programming experts. Several methods have been proposed to bridge this disparity, with varying degrees of success.

High performance natively-compiled scientific libraries (such as SciPy) seek to provide a portable, high-performance interface for common tasks, but the usability and efficiency of an interface often varies inversely to its generality. In addition, SciPy's implementations are sequential, due to both the wide variety of parallel programming models and the difficulty of selecting parameters such as degree of concurrency, thread fan-out, etc.

SEJITS [SEJITS] provides the best of both worlds by allowing very compact Domain-Specific Embedded Languages (DSELs) to be embedded in Python. Specializers are mini-compilers for these DSELs, themselves implemented in Python, which perform code generation and compilation at runtime; the specializers only intervene during those parts of the Python program that use Python classes belonging to the DSEL. BLB is the latest such specializer in a growing collection.

ASP ("ASP is SEJITS for Python") is a powerful framework for bringing parallel performance to Python using targeted just-in-time code transformation. The ASP framework provides a skinny waist interface which allows multiple applications to be built and run upon multiple parallel frameworks by using a single run-time compiler, or specializer. Each specializer is a Python class which contains the tools to translate a function/functions written in Python into an equivalent function/functions written in one or more low-level efficiency languages. In addition to providing support for interfacing productivity code to multiple efficiency code back-ends, ASP includes several tools which help the efficiency programmer lower and optimize input code, as well as define the front-end DSL. Several specializers already use these tools to solve an array of problems relevant to scientific programmers [SEJITS].

Though creating a compiler for a DSL is not a new problem, it is one with which efficiency experts may not be familiar. ASP eases this task by providing accessible interfaces for AST transformation. The `NodeTransformer` interface in the ASP toolkit includes and expands upon CodePy's [CodePy] C++ AST structure, as well as providing automatic translation from Python to C++ constructs. By extending this interface, efficiency programmers can define their DSEL by modifying only those constructs which differ from standard python, or intercepting specialized constructs such as special function names. This frees the specializer writer from re-writing boilerplate for common constructs such as branches or arithmetic operations.

ASP also provides interfaces for managing source variants and platforms, to complete the task of code lowering. The ASP framework allows the specializer writer to specify Backends, which represent distinct parallel frameworks or platforms. Each backend may store multiple specialized source variants, and includes simple interfaces for selecting new or best-choice variants, as well as compiling and running the underlying efficiency source codes. Couple with the Mako templating language and ASP's

* Corresponding author: aprasad91@gmail.com
‡ University of California, Berkeley

AST transformation tools, efficiency programmers are relieved of writing and maintaining platform-specific boilerplate and tools, and can focus on providing the best possible performance for their specializer.

Related Work

Prior work on BLB includes a serial implementation of the algorithm, as described in "The Big Data Bootstrap" and a Scala implementation that runs on the Spark cluster computing framework, as described in "A Scalable Bootstrap for Massive Data". The first paper shows that the BLB algorithm produces statistically robust results on a small data set with a linear estimator function. The second paper describes how BLB scales with large data sets in distributed environments.

BLB

BLB ("Bag of Little Bootstraps") is a method to assess the quality of a statistical estimator, $\theta(X)$, based upon subsets of a sample distribution `X`. θ might represent such quantities as the parameters of a regressor, or the test accuracy of a machine learning classifier. In order to calculate θ, subsamples of size K^γ, where $K = |X|$ and γ is a real number between 0 and 1, are drawn n times without replacement from X, creating the independent subsets $X_1, X_2, ..., X_n$. Next, `K` elements are resampled with replacement from each subset X_i, m times. This procedure of resampling with replacement is referred to as bootstrapping. The estimator θ is applied to each bootstrap. These results are reduced using a statistical aggregator (e.g. mean, variance, margin of error, etc.) to form an intermediate estimate $\theta'(X_i)$.Finally, the mean of θ' for each subset is taken as the estimate for $\theta(X)$. This method is statistically rigorous, and in fact reduces bias in the estimate compared to other bootstrap methods [BLB]. In addition, its structural properties lend themselves to efficient parallelization.

DSEL for BLB

A BLB problem instance is defined by the estimators and reducers it uses, its sampling parameters, and its input data. Our BLB specializer exposes a simple but expressive interface which allows the user to communicate all of these elements using either pure Python or a simple DSEL.

The DSEL, which is formally specified in Appendix A, is designed to concisely express the most common features of BLB estimator computations: position-independent iteration over large data sets, and dense linear algebra. The BLB algorithm was designed for statistical and loss-minimization tasks. These tasks share the characteristic of position-independant computation; they depend only on the number and value of the unique elements of the argument data sets, and not upon the position of these data points within the set. For this reason, the DSEL provides a pythonic interface for iteration, instead of a position-oriented style (i.e., subscripts and incrementing index variables) which is common in lower-level languages. Because most data sets which BLB operates on will have high-dimensional data, the ability to efficiently express vector operations is an important feature of the DSEL. All arithmetic operations and function calls which operate on data are replaced in the final code with optimized, inlined functions which automatically handle data of any size without changes to the source code. In addition to these facilities, common dense linear algebra operations may also be accessed via special function calls in the DSEL.

The next set of problem parameters, the sampling parameters, are not represented directly in the DSEL; In fact, they are not referenced anywhere therein. This is because the sampling parameters, which comprise n, m, and γ, have pattern-level consequences, and have no direct bearing on the executrion of users' computations. These values can be passed as keyword arguments to the specializer object when it is created, or the specializer may be left to choose reasonable defaults.

The final components of a problem instance are the input data. Much of the necessary information about the input data is gleaned by the specializer without referring to the DSEL. However, a major component of what to do with the input data is expressed using the DSEL's annotation capability. Argument annotations, as seen in figure 1 below, are used to determine whether or not a given input should be subsampled as part of the BLB pattern. This is essential for many tasks, because it allows the user to pass in non-data information (e.g. a machine learning model vector) into the computation. Though the annotations are ultimately removed, the information they provide propagates as changes to the pattern within the execution template.

An example application of BLB is to do model verification. Suppose we have trained a classifier $\pi : \mathbb{R}^d \rightarrow C$ where `d` is the dimension of our feature vectors and `C` is the set of classes. We can define $\theta[Y]$ to be `error[Y]/|Y|`, where the error function is 1 if $\pi(y)$ is not the true class of `y`, and 0 elsewhere. If we then choose arithmetic mean as a statistical aggregator, the BLB method using the γ we defined will provide an estimate of the test error of our classifier.

```
import blb
import numpy

class SVMVerifierBLB( blb.BLB ):
    def compute_estimate( emails, tags, models = ('models', 'nosubsample') ):
    errors = 0.0
    for email, tag in emails, tags:
        choice = 0
        max_match = -1
        for model in models:
        match = dot( model, email )
        if match > max_match:
            choice = index() + 1
            max_match = match
        if choice != tag:
        errors += 1
    return errors / len( emails )

    def reduce_bootstraps( bootstraps ):
        mean = 0.0
        for bootstrap in bootstraps:
            mean += bootstrap
        return mean / len(bootstraps)

    def average( subsamples ):
        mean = 0.0
        for subsample in subsamples:
            mean += subsample
        return mean / len( subsamples )
```

Figure 1. User-supplied code for model verification application using BLB specializer.

The Specializer: A Compiler for the BLB DSEL

The BLB specializer combines various tools, as well as components of the ASP framework and a few thousand lines of custom code, to inspect and lower productivity code at run time.

The BLB DSEL is accessed by creating a new Python class which uses the base specializer class, `blb.BLB`, as a parent. Specific methods corresponding to the estimator and reducer functions are written with the DSEL, allowing the productivity programmer to easily express aspects of a BLB computation which can be difficult to write efficiently. Though much of this code is

converted faithfully from Python to C++ by the specializer, two important sets of constructs are intercepted and rewritten in an optimized way when they are lowered to efficiency code. The first such construct is the for loop. In the case of the estimator *theta*, these loops must be re-written to co-iterate over a weight set. As mentioned above, the bootstrap step of the algorithm samples with replacement a number of data points exponentially larger than the size of the set. A major optimization of this operation is to re-write the estimator to work with a weight set the same size as the subsample, who's weights sum to the size of the original data set. This is accomplished within the DSEL by automatically converting for loops over subsampled data sets into weighted loops, with weight sets drawn from an appropriate multinomial distribusion for each bootstrap. When this is done, the specializer converts all the operations in the interior of the loop to weighted operations, which is why only augmented assignments are permitted in the interior of loops Appendix A. The other set of constructs handled specially by the specializer are operators and function calls. These constructs are specialized as described in the previous section.

Introspection begins when a specializer object is instantiated. When this occurs, the specializer uses Python's inspect module to extract the source code from the specializer object's methods named `compute_estimate`, `reduce_bootstraps`, and `average`. The specializer then uses Python's ast module to generate a Python abstract syntax tree for each method.

The next stage of specialization occurs when the specialized function is invoked. When this occurs, the specializer extracts salient information about the problem, such as the size and data type of the inputs, and combines it with information about the platform gleaned using ASP's platform detector. Along with this information, each of the three estimator ASTs is passed to a converter object, which transforms the Python ASTs to C++ equivalents, as well as performing optimizations. The converter objects referred to above perform the most radical code transformations, and more so than any other part of the specializer might be called a run-time compiler (with the possible exception of the C++ compiler invoked later on). Once each C++ AST is produced, it is converted into a python string whose contents are a valid C++ function of the appropriate name. These functions-strings, along with platform and problem-specific data, are used as inputs to Mako templates to generate a C++ source file tailored for the platform and problem instance. Finally, CodePy is used to compile the generate source file and return a reference to the compiled function to Python, which can then be invoked.

In addition to code lowering and parallelization, the specializer is equipped to make pattern-level optimization decisions. These optimizations change the steps of the execution pattern, but do not affect the user's code. The best example of this in the BLB specializer is the decision of whether or not to load in subsamples. Subsamples of the full data set can be accessed by indirection to individual elements (a subsample is an array of pointers) or by loading the subsampled elements into a new buffer (loading in). Loading in subsamples encourages caching, and our experiments showed performance gains of up to 3x for some problem/platform combinations using this technique. However, as data sizes grow, the time spent moving data or contending for shared resources outweighs the caching benefit. Because the specializer has some knowledge of the platform and of the input data sizes, it is able to make predictions about how beneficial loading in will be, and can modify the efficiency level code to decide which inputs should be loaded in and which should not. The specializer determines this by comparing the size of a subsample to the size of the shared L2 cache; if the memory needed for a single thread would consume more than 40% of the resources, then subsamples will not be loaded in. The value of 40% is empirical, and determined for the particular experiments herein. In the future, this and other architecture-level optimizations will be made automatically by specializers by comparing the performance effects of such decisions on past problem instances.

The other major pattern-level decision for a BLB computation is choice of sampling parameters. These constitute the major efficiency/accuracy trade-off of the BLB approach. By default, the specializer sets these parameters conservatively, favoring accuracy heavily over efficiency; The default sampling parameters are n = 25 subsamples, m = 100 bootstraps per subsample, and $\gamma = 0.7$. Though each of these values has clear performance implications, the specializer does not adjust them based on platform parameters because it does not include a mechanism to evaluate acceptable losses in accuracy.

Empirical evidence shows that accuracy declines sharply using γ less than 0.5 [BLB], though does not increase much more using a higher value than 0.7. A change of .1 in this value leads to an order-of-magnitude change in subsample size for data sets in the 10-100 GB range, so the smallest value which will attain the desired accuracy should be chosen. The number of subsamples taken also has a major impact on performance. The run time of a specialized computation in these experiments could be approximated to within 5% error using the formula $t = \lceil \frac{n}{c} \rceil s$, where t is the total running time, c is the number of cores in use, and s is the time to compute the bootstraps of a single subsample in serial. Though the result from bootstraps of a given subsample will likely be close to the true estimate, at least 20 subsamples were needed in the experiments detailed here to reduce variance in the estimate to an acceptable level. Finally, the number of bootstraps per subsample determines how accurate an estimate is produced for each subsample. In the experiments described below, 40 bootstraps were used. In experiments not susceptible to noise, as few as 25 were used with acceptable results. Because the primary effect of additional bootstraps is to reduce the effect of noise and improve accuracy, care should be taken not to use too few.

Evaluation

We evaluated the performance gains from using our SEJITS specializer by performing model verification of a SVM classifier on a subset of the Enron email corpus [ENRON]. We randomly selected 10% (Approximately 120,000 emails) from the corpus to serve as our data set. From each email, we extracted the counts of all words in the email, as well as the user-defined directory the email was filed under. We then aggregated the word counts of all the emails to construct a Bag-of-Words model of our data set, and assigned classes based upon directory. In the interest of classification efficiency, we filtered the emails to use only those from the 20 most common classes, which preserved approximately 98% of our original data set. In the final count, our test data consisted of approximately 126,000 feature vectors and tags, with each feature vector composed of approximately 96,000 8-bit features. Using the SVM-Multiclass [SVM] library, we trained a SVM classifier to decide the likeliest storage directory for an email based upon its bag of words representation. We trained the

classifier on 10% of our data set, reserving the other 90% as a test set. We then applied the specialized code shown in figure 1 to estimate the accuracy of the classifier. We benchmarked the performance and accuracy of the specializer on a system using 4 Intel X7560 processors.

Our experiments indicate that our specialized algorithm was able to achieve performance gains of up to 31.6x with regards to the serial version of the same algorithm, and up to 22.1x with respect to other verification techniques. These gains did not come at the cost of greatly reduced accuracy; the results from repeated runs of the specialized code were both consistent and very close to the true population statistic.

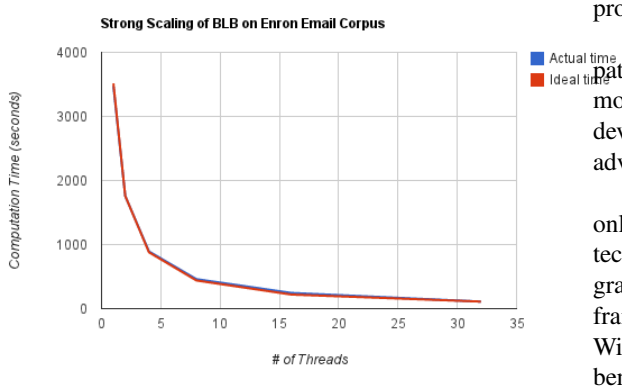


Figure 2. Efficiency gains from specialized code.

As is visible from figure 2 above, our specialized code achieved near-perfect strong scaling. In the serial case, the computation took approximately 3478 seconds. By comparison, when utilizing all 32 available hardware contexts, the exact same productivity level code returned in just under 110 seconds.

We also used SVM Multiclass' native verification utility to investigate the relative performance and accuracy of the specializer. SVM Multiclass' utility differs critically from our own in several ways: The former uses an optimized sparse linear algebra system, whereas the latter uses a general dense system; the former provides only a serial implementation; and the algorithm (traditional cross-validation) is different from ours. All of these factors should be kept in mind as results are compared. Nevertheless, the specializer garnered order-of-magnitude performance improvements once enough cores were in use. SVM Multiclass' utility determined the true population statistic in approximately 2200 seconds, making it faster than the serial incarnation of our specializer, but less efficient than even the dual-threaded version.

The native verification utility determined that the true error rate of the classifier on the test data was 67.86%. Our specializers estimates yielded a mean error rate of 67.24%, with a standard deviation of 0.36 percentage points. Though the true statistic was outside one standard deviation from our estimate's mean, the specializer was still capable of delivering a reasonably accurate estimate very quickly.

Limitations and Future Work

Some of the limitations of our current specializer are that the targets are limited to OpenMP and Cilk. We would like to implement a GPU and a cloud version of the BLB algorithm as additional targets for our specializer. We'd like to explore the performance of a GPU version implemented in CUDA. A cloud version will allow us to apply the BLB sepcializer to problems involving much larger data sets than are currently supported. Another feature we'd like to add is the ability for our specializer to automatically determine targets and parameters based on the input data size and platform specifications.

Conclusion

Using the SEJITS framework, productivity programmers are able to easily express high level computations while simultaneously gaining order-of-magnitude performance benefits. Because the parallelization strategy for a particular pattern of computation and hardware platform is often similar, efficiency expert programmers can make use of DSLs embedded in higher level languages, such as Python, to provide parallel solutions to large families of similar problems.

We were able to apply the ASP framework and the BLB pattern of computation to efficiently perform the high level task of model verification on a large data set. This solution was simple to develop with the help of the BLB specializer, and efficiently took advantage of all available parallel resources.

The BLB specializer provides the productivity programmer not only with performance, but with performance portability. Many techniques for bringing performance benefits to scientific programming, such as pre-compiled libraries, autotuning, or parallel framework languages, tie the user to a limited set of platforms. With SEJITS, productivity programmers gain the performance benefits of a wide variety of platforms without changes to source code.

This specializer is just one of a growing catalogue of such tools, which will bring to bear expert parallelization techniques to a variety of the most common computational patterns. With portable, efficient, high-level interfaces, domain expert programmers will be able to easily create and maintain code bases in the face of evolving parallel hardware and networking trends.

Acknowledgements

Armando Fox and Shoaib Kamil provided constant guidance in the development of this specializer, as well as the ASP project. Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael Jordan developed the BLB algorithm, and published the initial paper on the subject, *Bootstrapping Big Data*. They also consulted on effective parallelization strategies for that algorithm. John Duchi and Yuchen Zhang helped finalize the experiment plan and select appropriate test data sets. Richard Xia and Peter Birsinger developed the first BLB specializer interface, and continued work on the shared-nothing cloud version of this specializer.

Appendix A: Formal Specification of DSEL

```
## NAME indicates a valid python name, with the added
## stipulation it not start with '_blb_'
## INT and FLOAT indicate decimal representations of
## 64 bit integers and IEEE floating point numbers,
## respectively
## NEWLINE, INDENT, and DEDENT stand for the respective
## whitespace elements

P ::= OUTER_STMT* RETURN_STMT
AUG ::= '+=' \ '-=' | '*=' | '/='
NUM ::= INT | FLOAT
OP ::= '+' | '-' | '*' | '/' | '**'
COMP ::= '>' | '<' | '==' | '!=' | '<=' | '>='
BRANCH ::= 'if' NAME COMP NAME':'

RETURN_STMT ::= 'return' NAME | 'return' CALL
```

```
CALL ::= 'sqrt(' NAME ')'
       | 'len(' NAME ')'
       | 'mean(' NAME ')'
       | 'pow(' NAME',' INT ')'
       | 'dim(' NAME [',' INT ] ')'
       | 'dtype(' NAME ')'
       | 'MV_solve(' NAME',' NAME',' NAME ')'
       | NAME OP CALL | CALL OP NAME
       | CALL OP CALL | NAME OP NAME
       | NAME '*' NUM | CALL '*' NUM
       | NAME '/' NUM | CALL '/' NUM
       | NAME '**' NUM | CALL '**' NUM

INNER_STMT ::= NAME '=' NUM |
       | NAME = 'vector(' INT [',' INT]*', type='NAME ')'
       | NAME AUG CALL
       | NAME '=' 'index('[INT]')' OP NUM
       | NAME = NUM OP 'index('[INT]')'
       | BRANCH NEWLINE INDENT INNER_STMT* DEDENT
       | 'for' NAME[',' NAME]* 'in' NAME[',' NAME]*':' NEWLINE INDENT INNER_STMT* DEDENT

OUTER_STMT ::= NAME '=' NUM
       | NAME '=' 'vector(' INT [',' INT]*', type='NAME ')'
       | NAME '=' CALL | NAME AUG CALL
       | 'for' NAME[',' NAME]* 'in' NAME[',' NAME]*':' NEWLINE INDENT INNER_STMT* DEDENT
       | BRANCH NEWLINE INDENT OUTER_STMT* DEDENT
```

REFERENCES

[SEJITS] S. Kamil, D. Coetzee, A. Fox. "Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-In-Time Specialization". In SciPy 2011.

[BLB] A. Kleiner, A. Talwalkar, P. Sarkar, M. Jordan. "Bootstrapping Big Data". In NIPS 2011.

[CodePy] CodePy Homepage: http://mathema.tician.de/software/codepy

[ENRON] B. Klimt and Y. Yang. "The Enron corpus: A new dataset for email classification research". In ECML 2004.

[SVM] SVM-Multiclass Homepage: http://svmlight.joachims.org/svm_multiclass.html

[Spark] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In USENIX NSDI 2012.

A Computational Framework for Plasmonic Nanobiosensing

Adam Hughes‡*

Abstract—Basic principles in biosensing and nanomaterials precede the introduction of a novel fiber optic sensor. Software limitations in the biosensing domain are presented, followed by the development of a Python-based simulation environment. Finally, the current state of spectral data analysis within the Python ecosystem is discussed.

Index Terms—gold nanoparticles, fiber optics, biosensor, Python, immunoassay, plasmonics, proteins, metallic colloids, IPython, Traits, Chaco, Pandas, SEM,

Introduction

Because of their unique optical properties, metallic colloids, especially gold nanoparticles (AuNPs), have found novel applications in biology. They are utilized in the domain of *nanobiosensing* as platforms for biomolecule recognition. Nanobiosensing refers to the incorporation of nanomaterials into biosensing instrumentation. Sensors whose primary signal transduction mechanism is the interaction of light and metallic colloids are known as *plasmonic* sensors.[1]

Plasmonic sensors are constructed by depositing metallic layers (bulk or colloidal) onto a substrate such as glass, or in our case, onto a stripped optical fiber. Upon illumination, they relay continuous information about their surrounding physical and chemical environment. These sensors behave similarly to conventional assays with the added benefits of increased sensitivity, compact equipment, reduced sample size, low cost, and real-time data acquisition. Despite these benefits, nanobiosensing research in general is faced with several hinderances.

It is often difficult to objectively compare results between research groups, and sometimes even between experimental trials. This is mainly because the performance of custom sensors is highly dependent on design specifics as well as experimental conditions. The extensive characterization process found in commercial biosensors[2] exceeds the resources and capabilities of the average research group. This is partially due to a disproportionate investment in supplies and manpower; however, it is also due to a dearth of computational resources. The ad-hoc nature of empirical biosensor characterization often leads to asystematic experimental designs, implementations and conclusions between research groups. To compound matters, dedicated software is not evolving fast enough keep up with new biosensing technology. This lends an advantage to commercial biosensors, which use highly customized software to both control the experimental apparatus and extract underlying information from the data. Without a general software framework to develop similar tools, it is unreasonable to expect the research community to achieve the same breadth in application when pioneering new nanobiosensing technology.

Publications on novel biosensors often belaud improvement in sensitivity and cost over commercial alternatives; however, the aforementioned shortcomings relegate many new biosensors to prototype limbo. Until the following two freeware components are developed, new biosensors, despite any technical advantages over their commercial counterparts, will fall short in applicability:

1) A general and systematic framework for the development and objective quantification of nanobiosensors.
2) Domain-tailored software tools for conducting simulations and interpreting experimental data.

In regard to both points, analytical methods have been developed to interpret various aspects of plasmonic sensing; [R1] however, they have yet to be translated into a general software framework. Commercial software for general optical system design is available; however, it is expensive and not designed to encompass nanoparticles and their interactions with biomolecules. In the following sections, an effort to begin such computational endeavors is presented. The implications are relevant to plasmonic biosensing in general.

Optical Setup

We have developed an operational benchtop setup which records rapid spectroscopic measurements in the reflected light from the end of an AuNP-coated optical fiber. The nanoparticles are deposited on the flat endface of the fiber, in contrast to the commonly encountered method of depositing the AuNPs axially[3] along an etched region of the fiber [R2], [R3]. In either configuration, only the near-field interaction affects the signal, with no interference from far-field effects. The simple design is outlined in Fig. 1 (left). Broadband emission from a white LED is focused through a $10\times$ objective (not shown) into the 125μm core diameter of an optical fiber. AuNP-coated probes are connected into the setup via an

* Corresponding author: hugadams@gwmail.gwu.edu
‡ The George Washington University

1. This exposition is germane to plasmonic sensors, more so than to other nanobiosensor subgroups.

2. Biacore© and ForteBio© are examples of prominent nanobiosensing companies.

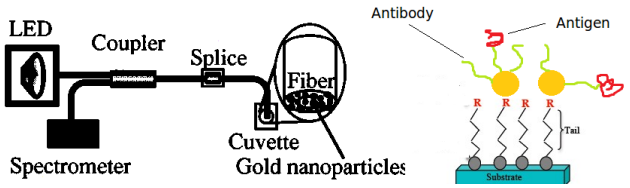


Fig. 1: *Left: Bench-top fiber optic configuration schematic, adapted from [R4]. Right: Depiction from bottom to top of fiber endface, APTMS monolayer, AuNPs, antibody-antigen coating.*

optical splice. The probes are dipped into solutions containing biomolecules, and the return light is captured by an OceanOptics© USB2000 benchtop spectrometer and output as ordered series data.

Fiber Surface Functionalization

16nm gold nanospheres are attached to the optical fiber via a linker molecule, (3-Aminoptopropyl)trimethoxysilane, or APTMS.[4] The surface chemistry of the gold may be further modified to the specifications of the experiment. One common modification is to covalently bind a ligand to the AuNPs using Dithiobis[succinimidyl propionate] (Lomant's reagent), and then use the fiber to study specificity in antibody-antigen interactions. This is depicted in Fig. 1 (right).

Modeling the Optical System in Python

The simulation codebase may be found at http://github.com/hugadams/fibersim.

Nanobiosensing resides at an intersection of optics, biology, and material science. To simulate such a system requires background in all three fields and new tools to integrate the pieces seamlessly. Nanobiosensor modeling must describe phenomena at three distinct length scales. In order of increasing length, these are:

1) A description of the optical properties of nanoparticles with various surface coatings.
2) The properties of light transmission through multilayered materials at the fiber endface.
3) The geometric parameters of the optics (e.g. fiber diameter, placement of nanoparticle monolayer, etc.).

The size regimes, shown in Fig. 2, will be discussed separately in the following subsections. It is important to note that the computational description of a *material* is identical at all three length scales. As such, general classes have been created and interfaced to accommodate material properties from datasets [R5] and models [R6]. This allows for a wide variety of experimental and theoretical materials to be easily incorporated into the simulation environment.

Modeling Nanoparticles

AuNPs respond to their surrounding environment through a phenomenon known as *surface plasmon resonance*. Incoming light couples to free electrons and induces surface oscillations on the

3. Axial deposition allows for more control of the fiber's optical properties; however, it makes probe creation more difficult and less reproducible.

4. APTMS is a heterobifunctional crosslinker that binds strongly to glass and gold respectively through silane and amine functional groups.

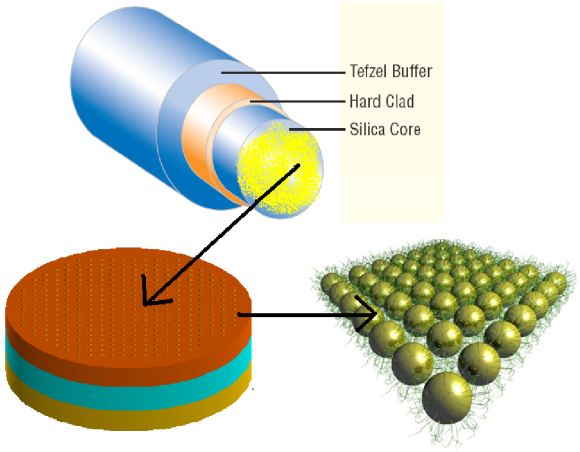


Fig. 2: *Three size regimes of the optical setup. Top: Optical fiber with an AuNP-coated endface. Left: Coarse approximation of a multilayered material. Right: Individual nanoparticles with protein shells.*

nanoparticle. The magnitude and dispersion of these oscillations is highly influenced by the dielectric media in direct contact with the particle's surface. As such, the scattering and absorption properties of the gold particles will change in response to changes in solution, as well as to the binding of biomolecules.

To model AuNPs, the complex dielectric function[5] of gold is imported from various sources, both from material models [R5] and datasets [R6]. The optical properties of bare and coated spheroids are described analytically by Mie theory [R7]. Scattering and absorption coefficients are computed using spherical Bessel functions from the *scipy.special* library of mathematical functions. Special routines and packages are available for computing the optical properties of non-spheroidal colloids; however, they have not yet been incorporated in this package.

AuNP modeling is straightforward; however, parametric analysis is uncommon. Enthought's `Traits` and `Chaco` packages are used extensively to provide interactivity. To demonstrate a use case, consider a gold nanoparticle with a shell of protein coating. The optical properties of the core-shell particle may be obtained analytically using Mie Theory;[6] however, analysis performed at a coarser scale requires this core-shell system to be approximated as a single composite particle (Fig. 3). With `Traits`, it is very easy for the user to interactively adjust the mixing parameters to ensure that the scattering properties of the approximated composite are as close as possible to those of the analytical core-shell particle. In this example, and in others, interactivity is favorable over complex optimization techniques.

Modeling Material Layers

The fiber endface at a more coarse resolution resembles a multilayered dielectric stack of homogeneous materials, also referred to as a thin film (Fig. 5). In the limits of this approximation, the reflectance, transmittance, and absorbance through the slab can

5. The dielectric function and shape of the particle are the only parameters required to compute its absorption and scattering cross sections.

6. Assuming that the shell is perfectly modeled; however, in practice the optical properties of protein mixtures are approximated by a variety of mixing models and methods.

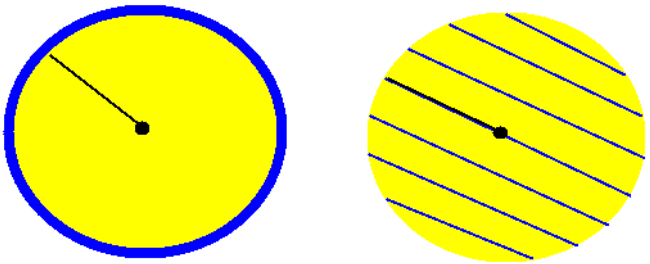


Fig. 3: *Left: A nanoparticle with heterogeneous core and shell dielectrics* $(\varepsilon_1, \varepsilon_2)$*, of radius,* $r = r_1 + r_2$*. Right: Composite approximation of a homogeneous material, with effective dielectric* ε'*, and radius,* r'*.*

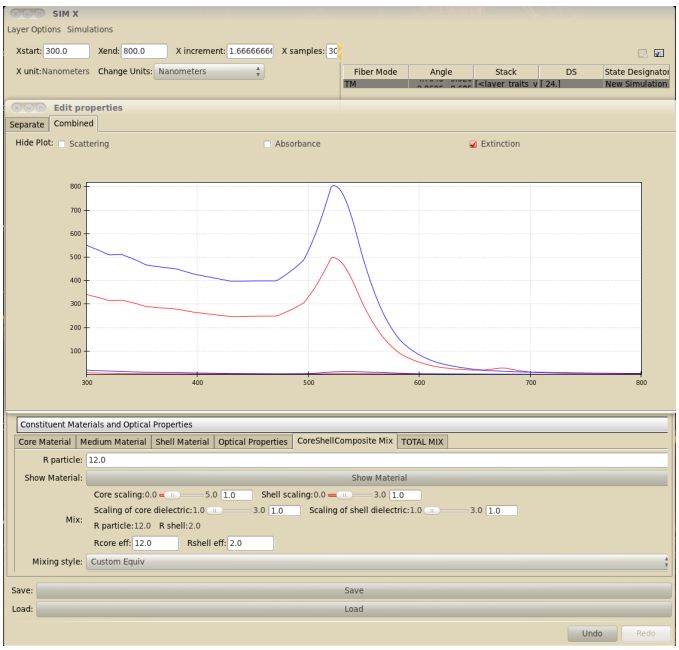


Fig. 4: *Screenshot of an interactive* `TraitsUI` *program for modeling the scenario in Fig. 3: the extinction spectra of a protein-coated AuNP (blue) compared to that of an equivalent core-shell composite (red).*

be calculated recursively for n-layered systems [R8]. Thin film optical software is commercially available and used extensively in optical engineering, for example, in designing coatings for sunglasses. Unfortunately, a free, user-friendly alternative is not available.[7] In addition, these packages are usually not designed for compatibility with nanomaterials; therefore, we have begun development of an extensible thin film Python API that incorporates nanomaterials. This is ideal, for example, in simulating a fiber immersed in a solvent with a variable refractive index (e.g. a solution with changing salinity). The program will ensure that as the solvent changes, the surrounding shell of the nanoparticle, and hence its extinction spectra, will update accordingly.

Optical Configurations and Simulation Environment

With the material and multilayer APIs in place, it is straightforward to incorporate an optical fiber platform. The light source and fiber parameters merely constrain the initial conditions of light

7. Open-source thin film software is often limited in scope and seldom provides a user-interface, making an already complex physical system more convoluted.

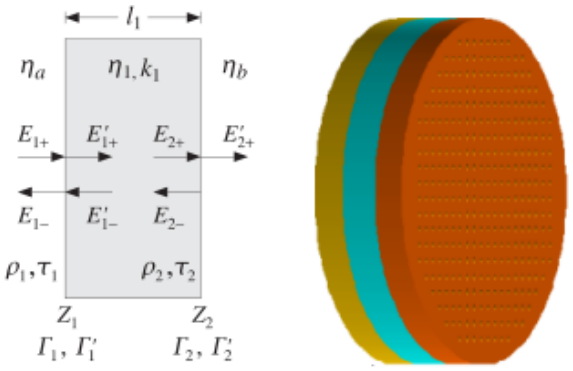


Fig. 5: *Left: Electromagnetic field components at each interface of a dielectric* slab *[R7]. Right: Illustration of a multilayered material whose optical properties would be described by such treatment.*

entering the multilayer interface; thus, once the correct multilayered environment is established, it easy to compare performance between different fiber optic configurations. Built-in parameters already account for the material makeup and physical dimensions of many commercially available optical fibers. A phase angle has been introduced to distinguish nanomaterial deposition on the fiber endface from axial deposition. This amounts to a 90° rotation of the incident light rays at the multilayered interface.[8]

The entire application was designed for exploratory analysis, so adjusting most parameters will automatically trigger system-wide updates. To run simulations, one merely automates setting `Trait` attributes in an iterative manner. For example, by iterating over a range of values for the index of refraction of the AuNP shells, one effectively simulates materials binding to the AuNPs. After each iteration, `Numpy` arrays are stored for the updated optical variables such as the extinction spectra of the particles, dielectric functions of the mixed layers, and the total light reflectance at the interface. All data output is formatted as ordered series to mimic the actual output of experiments; thus, simulations and experiments can be analyzed side-by-side without further processing. With this work flow, it is quite easy to run experiments and simulations in parallel as well as compare a variety of plasmonic sensors objectively.

Data Analysis

Our work flow is designed to handle ordered series spectra generated from both experiment and simulation. The Python packages `IPython`, `Traits`, and `Pandas` synergistically facilitate swift data processing and visualization. Biosensing results are information-rich, both in the spectral and temporal dimensions. Molecular interactions on the AuNP's surface have spectral signatures discernible from those of environmental changes. For example, the slow timescale of protein binding events is orders of magnitude less than the rapid temporal response to environmental changes.

Fig. 6 illustrates a fiber whose endface has been coated with gold nanoparticles and subsequently immersed in water. The top left plot shows the reflected light spectrum function of time. When submerged in water, the signal is very stable. Upon the addition of

8. The diameter of the optical fiber as well as the angle at which light rays interact with the material interface has a drastic effect on the system because each light mode contributes differently to the overall signal, which is the summation over all modes.

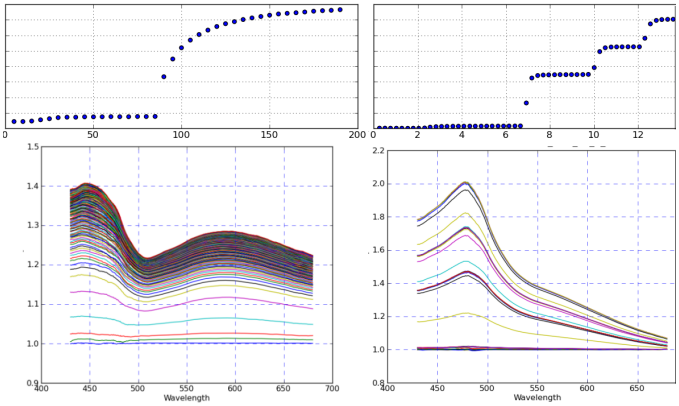


Fig. 6: *Temporal evolution (top) and spectral absorbance (bottom) of the light reflectance at the fiber endface due to a protein-protein interaction (left) as opposed to the stepwise addition of glycerin (right).*

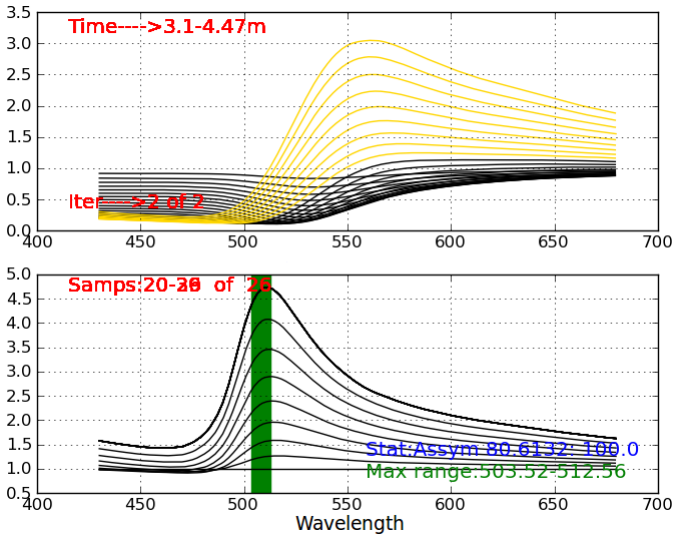


Fig. 7: *Top: Absorbance plot of the real-time deposition of AuNPs onto an optical fiber. Bottom: Time-slice later in the datasets shows that the signal is dominated by signal at the surface plasmon resonance peak for gold, $\lambda_{SPR} \approx 520$ nm. The exemplifies the correct timescale over which spectral events manifest.*

micromolar concentrations of Bovine Serum Albumin (BSA), the signal steadily increases as the proteins in the serum bind to the gold. About an hour after BSA addition, the nanoparticle binding sites saturate and the signal plateaus.

Fig. 6 (top right) corresponds to a different situation. Again, an AuNP-coated fiber is immersed in water. Instead of proteins, glycerin droplets are added. The fiber responds to these refractive index changes in an abrupt, stepwise fashion. Whereas the serum binding event evolves over a timescale of about two hours, the response to an abrupt environmental change takes mere seconds. This is a simple demonstration of how timescale provides insights to the physiochemical nature of the underlying process.

The dataset's spectral dimension can be used to identify physiochemical phenomena as well. Absorbance plots corresponding to BSA binding and glycerin addition are shown at the bottom of Fig. 6. These profiles tend to depend on the size of the biomolecules in the interaction. The spectral profile of BSA-AuNP binding, for example, is representative of other large proteins binding to gold. Similarly, index changes from saline, buffers and other viscous solutions are consistent with the dispersion profile of glycerin. Small biomolecules such as amino acids have yet another spectral signature (not shown), as well as a timestamp that splits the difference between protein binding and refractive index changes. This surprising relationship between the physiochemistry of an interaction and its temporal and spectral profiles aids in the interpretation of convoluted results in complex experiments.

Consistent binding profiles require similar nanoparticle coverage between fibers. If the coating process is lithographic, it is easier to ensure consistent coverage; however, many plasmonic biosensors are created through a *wet* crosslinking process similar to the APTMS deposition described here. Wet methods are more susceptible to extraneous factors; yet remarkably, we can use the binding profile as a tool to monitor and control nanoparticle deposition in realtime.

Fig. 7 (top) is an absorbance plot of the deposition of gold nanoparticles onto the endface of an optical fiber (dataset begins at $y = 1$). As the nanoparticles accumulate, they initially absorb signal, resulting in a drop in light reflectance; however, eventually the curves invert and climb rapidly. This seems to suggest the existence of a second process; however, simulations have confirmed that this inflection is merely a consequence of the nanoparticle film density and its orientation on the fiber. The spectral signature of the AuNP's may be observed by timeslicing the data (yellow curves) and renormalizing to the first curve in the subset. This is plotted in Fig. 7 (bottom), and clearly shows spectral dispersion with major weight around $\lambda = 520$ nm, the surface plasmon resonance peak of our gold nanoparticles.

This approach to monitoring AuNP deposition not only allows one to control coverage,[9] but also provides information on deposition quality. Depending on various factors, gold nanoparticles may tend to aggregate into clusters, rather than form a monolayer. When this occurs, red-shifted absorbance profiles appear in the timeslicing analysis. Because simple plots like Fig. 7 contain so much quantitative and qualitative information about nanoparticle coverage, we have begun an effort to calibrate these curves to measured particle coverage using scanning electron microscopy (SEM) (Fig. 8).

The benefits of such a calibration are two-fold. First, it turns out that the number of AuNP's on the fiber is a crucial parameter for predicting relevant biochemical quantities such as the binding affinity of two ligands. Secondly, it is important to find several coverages that optimize sensor performance. There are situations when maximum dynamic range at low particle coverage is desirable, for example in measuring non-equilibrium binding kinetics. Because of mass transport limitations, estimations of binding affinity tend to be in error for densely populated monolayers. In addition, there are coverages that impair dynamic range. Thus, it is important to optimize and characterize sensor performance at various particle coverages. Although simulations can estimate this relationship, it should also be confirmed experimentally.

Since most non-trivial biosensing experiments contain multiple phases (binding, unbinding, purging of the sensor surface, etc.), the subsequent data analysis requires the ability to rescale, resample and perform other manual curations on-the-fly. `Pandas` provides a great tool set for manipulating series data in such a

9. The user merely removes the fiber from AuNP when the absorbance reaches a preset value.

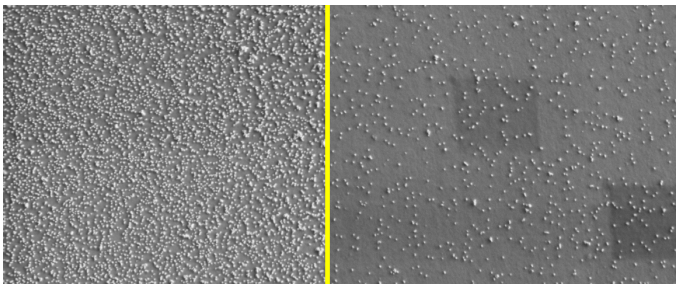


Fig. 8: *SEM images of fiber endfaces with 25% (left) and 5% (right) AuNP surface coverage at 30,000 X magnification.*

manner. For example, slicing a set of ordered series data by rows (spectral dimension) and columns (temporal dimension) is quite simple:

```
## Read series data from tab-delimited
## file into a pandas DataFrame object
from pandas import read_csv
data=read_csv('path to file', sep='\t')

## Select data by column index
data[['time1', 'time2']]

## Slice data by row label (wavelength)
data.ix[500.0:750.0]
```

By interfacing to `Chaco`, and to the `Pandas` plotting interface, one can slice, resample and visualize interesting regions in the dataspace quite easily. Through these packages, it is possible for non-computer scientists to not just visualize, but to dynamically *explore* the dataset. The prior examples of BSA and glycerin demonstrated just how much information could be extracted from the data using only simple, interactive methods.

our interactive approach is in contrast to popular *all-in-one* analysis methods. In Two-Dimensional Correlation Analysis (2DCA), [R9] for example, cross correlations of the entire dataset are consolidated into two contour plots. These plots tend to be difficult to interpret,[10] and become intractable for multi-staged events. Additionally, under certain experimental conditions they cannot be interpreted at all. It turns out that much of the same information provided by 2DCA can be ascertained using the simple, dynamic analysis methods presented here. This is not to suggest that techniques like 2DCA are disadvantageous, merely that some of the results may be obtained more simply. Perhaps in the future, transparent, interactive approaches will constitute the core of the spectral data analysis pipeline with sophisticated techniques like 2DCA adopting a complimentary role.

Conclusions

A benchtop nanobiosensor has been developed for the realtime detection of biomolecular interactions. It, as well as other emergent biosensing technologies, is hindered by a lack of dedicated open-source software. In an effort to remedy this, prototypical simulation and analysis tools have been developed to assist with our plasmonic sensor and certainly have the potential for wider applicability. Scientific Python libraries, especially `Chaco` and `Pandas`, reside at the core of our data analysis toolkit and are proving invaluable for interacting with and visualizing results. Unexpected physiochemical identifiers appear consistently within experimental results. These binding profiles not only provide new qualitative insights, but with the help of SEM imaging, may soon open new avenues towards the difficult task of quantifying biosensor output. Python has proven invaluable to our research, and just as it has suffused the domains of astronomy and finance, seems primed to emerge as the de-facto design platform in biosensing and its related fields.

10. 2DCA decomposes series data into orthogonal synchronous and asynchronous components. By applying the so-called Noda's rules, one can then analyze the resultant contour maps and infer information about events unfolding in the system.

Acknowledgements

I would like to thank my advisor, Dr. Mark Reeves, for his devoted guidance and support. I owe a great debt to Annie Matsko for her dedication in the lab and assistance in drafting this document. In regard to the programming community, I must foremost thank Enthought and the other sponsors of the SciPy2012 conference. Their generous student sponsorship program made it possible for me to attend for that I am gracious. Although indebted to countless members of the Python community, I must explicitly thank Jonathan March, Robert Kern and Stéfan van der Walt for their patience in helping me through various programming quandries. Thank you De-Hao Tsai and Vincent Hackley at the Material Measurement Laboratory at NIST for your helpful discussions and allowing our group to use your zeta-potential instrumentation. Finally, I must thank the George Gamow Research Fellowship Program, the Luther Rice Collaborative Research Fellowship program, and the George Washington University for the Knox fellowship for generous financial support.

References

[R1] Anuj K. Sharma B.D. Gupta. *Fiber Optic Sensor Based on Surface Plasmon Resonance with Nanoparticle Films.* Photonics and Nanostructures - Fundamentals and Applications, 3:30,37, 2005.

[R2] Ching-Te Huang Chun-Ping Jen Tzu-Chien Chao. *A Novel Design of Grooved Fibers for Fiber-optic Localized Plasmon Resonance Biosensors.*, Sensors, 9:15, August 2009.

[R3] Wen-Chi Tsai Pi-Ju Rini Pai. *Surface Plasmon Resonance-based Immunosensor with Oriented Immobilized Antibody Fragments on a Mixed Self-Assembled Monolayer for the Determination of Staphylococcal Enterotoxin B.*, MICROCHIMICA ACTA, 166(1-2):115–122, February 2009.

[R4] Mitsui Handa Kajikawa. *Optical Fiber Affinity Biosensor Based on Localized Surface Plasmon Resonance.*, Applied Physics Letters, 85(18):320–340, November 2004.

[R5] Etchegoin Ru Meyer. *An Analytic Model for the Optical Properties of Gold.* The Journal of Chemical Physics, 125, 164705, 2006.

[R6] Christy, Johnson. *Optical Constants of Noble Metals.* Physics Review, 6 B:4370-4379, 1972.

[R7] Bohren Huffman. *Absorption and Scattering of Light by Small Particles*, Wiley Publishing, 1983.

[R8] Orfanidis, Sophocles. *Electromagnetic Waves and Antennas.* 2008

[R9] Yukihiro Ozaki Isao Noda. *Two-Dimensional Correlation Spectroscopy.* Wiley, 2004.

A Tale of Four Libraries

Alejandro Weinstein‡*, Michael Wakin‡

Abstract—This work describes the use some scientific Python tools to solve information gathering problems using Reinforcement Learning. In particular, we focus on the problem of designing an agent able to learn how to gather information in linked datasets. We use four different libraries—RL-Glue, Gensim, NetworkX, and scikit-learn—during different stages of our research. We show that, by using NumPy arrays as the default vector/matrix format, it is possible to integrate these libraries with minimal effort.

Index Terms—reinforcement learning, latent semantic analysis, machine learning

Introduction

In addition to bringing efficient array computing and standard mathematical tools to Python, the NumPy/SciPy libraries provide an ecosystem where multiple libraries can coexist and interact. This work describes a success story where we integrate several libraries, developed by different groups, to solve some of our research problems.

Our research focuses on using Reinforcement Learning (RL) to gather information in domains described by an underlying linked dataset. We are interested in problems such as the following: given a Wikipedia article as a seed, find other articles that are interesting relative to the starting point. Of particular interest is to find articles that are more than one-click away from the seed, since these articles are in general harder to find by a human.

In addition to the staples of scientific Python computing NumPy, SciPy, Matplotlib, and IPython, we use the libraries RL-Glue [Tan09], NetworkX [Hag08], Gensim [Reh10], and scikit-learn [Ped11].

Reinforcement Learning considers the interaction between a given environment and an agent. The objective is to design an agent able to learn a policy that allows it to maximize its total expected reward. We use the RL-Glue library for our RL experiments. This library provides the infrastructure to connect an environment and an agent, each one described by an independent Python program.

We represent the linked datasets we work with as graphs. For this we use NetworkX, which provides data structures to efficiently represent graphs, together with implementations of many classic graph algorithms. We use NetworkX graphs to describe the environments implemented using RL-Glue. We also use these graphs to create, analyze and visualize graphs built from unstructured data.

* *Corresponding author: aweinste@mines.edu*
‡ *the EECS department of the Colorado School of Mines*

One of the contributions of our research is the idea of representing the items in the datasets as vectors belonging to a linear space. To this end, we build a Latent Semantic Analysis (LSA) [Dee90] model to project documents onto a vector space. This allows us, in addition to being able to compute similarities between documents, to leverage a variety of RL techniques that require a vector representation. We use the Gensim library to build the LSA model. This library provides all the machinery to build, among other options, the LSA model. One place where Gensim shines is in its capability to handle big data sets, like the entirety of Wikipedia, that do not fit in memory. We also combine the vector representation of the items as a property of the NetworkX nodes.

Finally, we also use the manifold learning capabilities of sckit-learn, like the ISOMAP algorithm [Ten00], to perform some exploratory data analysis. By reducing the dimensionality of the LSA vectors obtained using Gensim from 400 to 3, we are able to visualize the relative position of the vectors together with their connections.

Source code to reproduce the results shown in this work is available at https://github.com/aweinstein/a_tale.

Reinforcement Learning

The RL paradigm [Sut98] considers an agent that interacts with an environment described by a Markov Decision Process (MDP). Formally, an MDP is defined by a state space $\mathscr{X}$, an action space $\mathscr{A}$, a transition probability function P, and a reward function r. At a given sample time $t = 0, 1, \ldots$ the agent is at state $x_t \in \mathscr{X}$, and it chooses action $a_t \in \mathscr{A}$. Given the current state x and selected action a, the probability that the next state is x' is determined by $P(x, a, x')$. After reaching the next state x', the agent observes an immediate reward $r(x')$. Figure 1 depicts the agent-environment interaction. In an RL problem, the objective is to find a function $\pi : \mathscr{X} \mapsto \mathscr{A}$, called the *policy*, that maximizes the total expected reward

$$R = \mathbf{E}\left[\sum_{t=1}^{\infty} \gamma^t r(x_t)\right],$$

where $\gamma \in (0, 1)$ is a given discount factor. Note that typically the agent does not know the functions P and r, and it must find the optimal policy by interacting with the environment. See Szepesvári [Sze10] for a detailed review of the theory of MDPs and the different algorithms used in RL.

We implement the RL algorithms using the RL-Glue library [Tan09]. The library consists of the *RL-Glue Core* program and a set of codecs for different languages[1] to communicate with the library. To run an instance of a RL problem one needs to write three different programs: the *environment*, the *agent*, and the *experiment*. The environment and the agent programs match exactly the corresponding elements of the RL framework, while

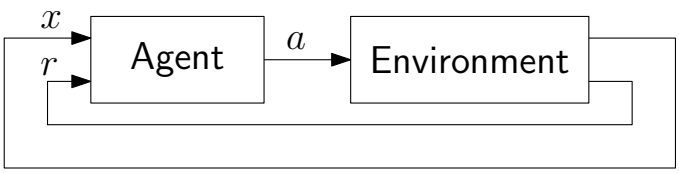


Fig. 1: *The agent-environment interaction. The agent observes the current state x and reward r; then it executes action $\pi(x) = a$.*

the experiment orchestrates the interaction between these two. The following code snippets show the main methods that these three programs must implement:

```
################# environment.py #################
class env(Environment):
    def env_start(self):
        # Set the current state

        return current_state

    def env_step(self, action):
        # Set the new state according to
        # the current state and given action.

        return reward

 ################### agent.py ###################
 class agent(Agent):
     def agent_start(self, state):
         # First step of an experiment

         return action

     def agent_step(self, reward, obs):
         # Execute a step of the RL algorithm

         return action

 ################ experiment.py #################
 RLGlue.init()
 RLGlue.RL_start()
 RLGlue.RL_episode(100) # Run an episode
```

Note that RL-Glue is only a thin layer among these programs, allowing us to use any construction inside them. In particular, as described in the following sections, we use a NetworkX graph to model the environment.

Computing the Similarity between Documents

To be able to gather information, we need to be able to quantify how relevant an item in the dataset is. When we work with documents, we use the similarity between a given document and the seed to this end. Among the several ways of computing similarities between documents, we choose the Vector Space Model [Man08]. Under this setup, each document is represented by a vector. The similarity between two documents is estimated by the *cosine similarity* of the document vector representations.

The first step in representing a piece of text as a vector is to build a *bag of words* model, where we count the occurrences of each term in the document. These word frequencies become the vector entries, and we denote the *term frequency* of term t in document d by $\mathrm{tf}_{t,d}$. Although this model ignores information related to the order of the words, it is still powerful enough to produce meaningful results.

In the context of a collection of documents, or corpus, word frequency is not enough to asses the importance of a term. For this reason, we introduce the quantity *document frequency* df_t, defined to be the number of documents in the collection that contain term t. We can now define the *inverse document frequency* (idf) as

$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t},$$

where N is the number of documents in the corpus. The idf is a measure of how unusual a term is. We define the tf-idf weight of term t in document d as

$$\text{tf-idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t\,.$$

This quantity is a good indicator of the discriminating power of a term inside a given document. For each document in the corpus we compute a vector of length M, where M is the total number of terms in the corpus. Each entry of this vector is the tf-idf weight for each term (if a term does not exist in the document, the weight is set to 0). We stack all the vectors to build the $M \times N$ *term-document matrix* C.

Note that since typically a document contains only a small fraction of the total number of terms in the corpus, the columns of the term-document matrix are sparse. The method known as Latent Semantic Analysis (LSA) [Dee90] constructs a low-rank approximation C_k of rank at most k of C. The value of k, also known as the *latent dimension*, is a design parameter typically chosen to be in the low hundreds. This low-rank representation induces a projection onto a k-dimensional space. The similarity between the vector representation of the documents is now computed after projecting the vectors onto this subspace. One advantage of LSA is that it deals with the problems of *synonymy*, where different words have the same meaning, and *polysemy*, where one word has different meanings.

Using the Singular Value Decomposition (SVD) of the term-document matrix $C = U\Sigma V^T$, the k-rank approximation of C is given by

$$C_k = U_k \Sigma_k V_k^T,$$

where U_k, Σ_k, and V_k are the matrices formed by the k first columns of U, Σ, and V, respectively. The tf-idf representation of a document q is projected onto the k-dimensional subspace as

$$q_k = \Sigma_k^{-1} U_k^T q.$$

Note that this projection transforms a sparse vector of length M into a dense vector of length k.

In this work we use the *Gensim* library [Reh10] to build the vector space model. To test the library we downloaded the top 100 most popular books from project Gutenberg.[2] After constructing the LSA model with 200 latent dimensions, we computed the similarity between *Moby Dick*, which is in the corpus used to build the model, and 6 other documents (see the results in Table 1). The first document is an excerpt from *Moby Dick*, 393 words long. The second one is an excerpt from the Wikipedia *Moby Dick* article. The third one is an excerpt, 185 words long, of *The Call of the Wild*. The remaining two documents are excerpts from Wikipedia articles not related to *Moby Dick*. The similarity values we obtain validate the model, since we can see high values (above 0.8) for the documents related to *Moby Dick*, and significantly smaller values for the remaining ones.

1. Currently there are codecs for Python, C/C++, Java, Lisp, MATLAB, and Go.

2. As per the April 20, 2011 list, http://www.gutenberg.org/browse/scores/top.

Text description	LSA similarity
Excerpt from Moby Dick	0.87
Excerpt from Wikipedia Moby Dick article	0.83
Excerpt from The Call of the Wild	0.48
Excerpt from Wikipedia Jewish Calendar article	0.40
Excerpt from Wikipedia Oxygen article	0.33

TABLE 1: *Similarity between Moby Dick and other documents.*

Next, we build the LSA model for Wikipedia that allows us to compute the similarity between Wikipedia articles. Although this is a lengthy process that takes more than 20 hours, once the model is built, a similarity computation is very fast (on the order of 10 milliseconds). Results in the next section make use of this model.

Note that although in principle it is simple to compute the LSA model of a given corpus, the size of the datasets we are interested in make doing this a significant challenge. The two main difficulties are that in general (i) we cannot hold the vector representation of the corpus in RAM memory, and (ii) we need to compute the SVD of a matrix whose size is beyond the limits of what standard solvers can handle. Here Gensim does stellar work by being able to handle both these challenges.

Representing the State Space as a Graph

We are interested in the problem of gathering information in domains described by linked datasets. It is natural to describe such domains by graphs. We use the NetworkX library [Hag08] to build the graphs we work with. NetworkX provides data structures to represent different kinds of graphs (undirected, weighted, directed, etc.), together with implementations of many graph algorithms. NetworkX allows one to use any hashable Python object as a node identifier. Also, any Python object can be used as a node, edge, or graph attribute. We exploit this capability by using the LSA vector representation of a Wikipedia article, which is a NumPy array, as a node attribute.

The following code snippet shows a function[3] used to build a directed graph where nodes represent Wikipedia articles, and the edges represent links between articles. Note that we compute the LSA representation of the article (line 11), and that this vector is used as a node attribute (line 13). The function obtains up to `n_max` articles by breadth-first crawling the Wikipedia, starting from the article defined by `page`.

```
1  def crawl(page, n_max):
2      G = nx.DiGraph()
3      n = 0
4      links = [(page, -1, None)]
5      while n < n_max:
6          link = links.pop()
7          page = link[0]
8          dist = link[1] + 1
9          page_text = page.edit().encode('utf-8')
10         # LSI representation of page_text
11         v_lsi = get_lsi(page_text)
12         # Add node to the graph
13         G.add_node(page.name, v=v_lsi)
14         if link[2]:
15             source = link[2]
16             dest = page.name
17             if G.has_edge(source, dest):
18                 # Link already exists
19                 continue
20             else:
21                 sim = get_similarity(page_text)
22                 self.G.add_edge(source,
23                                 dest,
24                                 weight=sim,
25                                 d=dist)
26         new_links = [(l, dist, page.name)
27                      for l in page.links()]
28         links = new_links + links
29         n += 1
30
31     return G
```

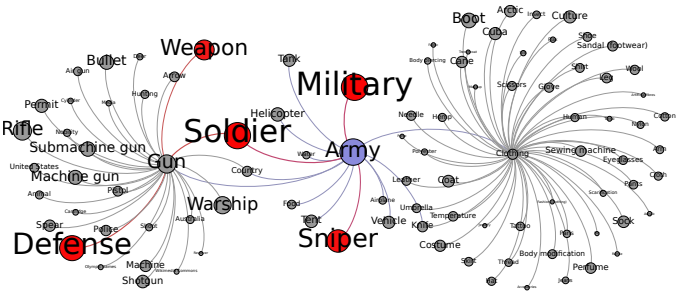


Fig. 2: *Graph for the "Army" article in the simple Wikipedia with 97 nodes and 99 edges. The seed article is in light blue. The size of the nodes (except for the seed node) is proportional to the similarity. In red are all the nodes with similarity greater than 0.5. We found two articles ("Defense" and "Weapon") similar to the seed three links ahead.*

We now show the result of running the code above for two different setups. In the first instance we crawl the *Simple English Wikipedia*[4] using "Army" as the seed article. We set the limit on the number of articles to visit to 100. The result is depicted[5] in Fig. 2, where the node corresponding to the seed article is in light blue and the remaining nodes have a size proportional to the similarity with respect to the seed. Red nodes are the ones with similarity bigger than 0.5. We observe two nodes, "Defense" and "Weapon", with similarities 0.7 and 0.53 respectively, that are three links away from the seed.

In the second instance we crawl Wikipedia using the article "James Gleick"[6] as seed. We set the limit on the number of articles to visit to 2000. We show the result in Fig. 3, where, as in the previous example, the node corresponding to the seed is in light blue and the remaining nodes have a size proportional to the similarity with respect to the seed. The eleven red nodes are the ones with similarity greater than 0.7. Of these, 9 are more than one link away from the seed. We see that the article with the biggest similarity, with a value of 0.8, is about "Robert Wright (journalist)", and it is two links away from the seed (passing through the "Slate magazine" article). Robert Wright writes books about sciences, history and religion. It is very reasonable to consider him an author similar to James Gleick.

Another place where graphs can play an important role in the RL problem is in finding basis functions to approximate the value-

3. The parameter `page` is a mwclient page object. See http://sourceforge.net/apps/mediawiki/mwclient/.

4. The Simple English Wikipedia (http://simple.wikipedia.org) has articles written in *simple English* and has a much smaller number of articles than the standard Wikipedia. We use it because of its simplicity.

5. To generate this figure, we save the NetworkX graph in GEXF format, and create the diagram using Gephi (http://gephi.org/).

6. James Gleick is "an American author, journalist, and biographer, whose books explore the cultural ramifications of science and technology".

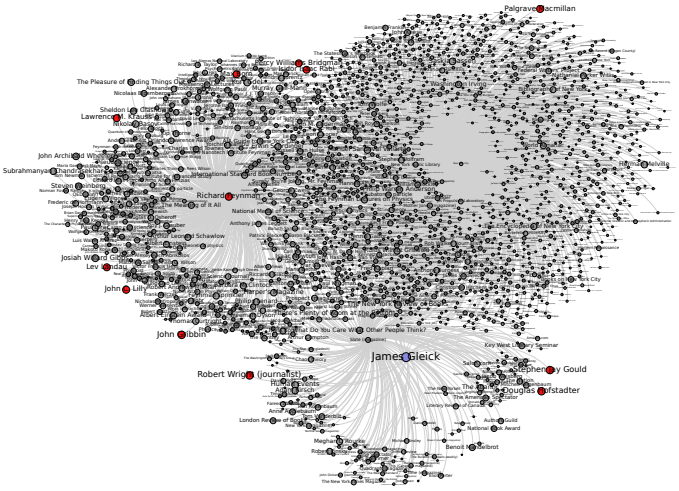


Fig. 3: *Graph for the "James Gleick" Wikipedia article with 1975 nodes and 1999 edges. The seed article is in light blue. The size of the nodes (except for the seed node) is proportional to the similarity. In red are all the nodes with similarity bigger than 0.7. There are several articles with high similarity more than one link ahead.*

function. The value-function is the function $V^\pi : \mathscr{X} \mapsto \mathbb{R}$ defined as

$$V^\pi(x) = \mathbf{E}\left[\sum_{t=1}^{\infty} \gamma^t r(x_t) \;\middle|\; x_0 = x, a_t = \pi(x_t)\right],$$

and plays a key role in many RL algorithms [Sze10]. When the dimension of $\mathscr{X}$ is significant, it is common to approximate $V^\pi(x)$ by

$$V^\pi \approx \hat{V} = \Phi w,$$

where Φ is an n-by-k matrix whose columns are the basis functions used to approximate the value-function, n is the number of states, and w is a vector of dimension k. Typically, the basis functions are selected by hand, for example, by using polynomials or radial basis functions. Since choosing the right functions can be difficult, Mahadevan and Maggioni [Mah07] proposed a framework where these basis functions are learned from the topology of the state space. The key idea is to represent the state space by a graph and use the k smoothest eigenvectors of the graph laplacian, dubbed *Proto-value* functions, as basis functions. Given the graph that represents the state space, it is very simple to find these basis functions. As an example, consider an environment consisting of three 16×20 grid-like rooms connected in the middle, as shown in Fig. 4. Assuming the graph is stored in `G`, the following code[7] computes the eigenvectors of the laplacian:

```
L = nx.laplacian(G, sorted(G.nodes()))
evalues, evec = np.linalg.eigh(L)
```

Figure 5 shows[8] the second to fourth eigenvectors. Since in general value-functions associated to this environment will exhibit a fast change rate close to the room's boundaries, these eigenvectors provide an efficient approximation basis.

7. We assume that the standard `import numpy as np` and `import networkx as nx` statements were previously executed.

8. The eigenvectors are reshaped from vectors of dimension $3 \times 16 \times 20 = 960$ to a matrix of size 16-by-60. To get meaningful results, it is necessary to build the laplacian using the nodes in the grid in a row major order. This is why the `nx.laplacian` function is called with `sorted(G.nodes())` as the second parameter.

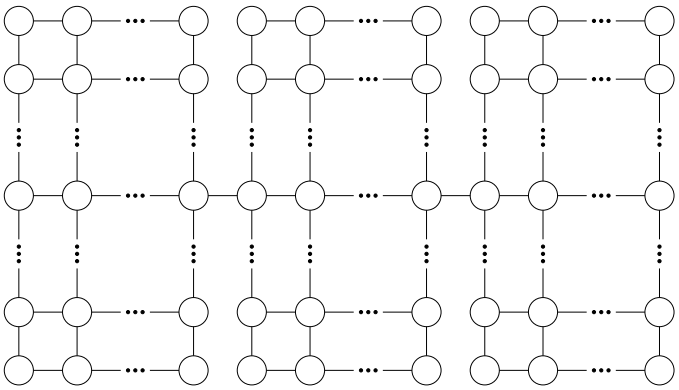


Fig. 4: *Environment described by three* 16×20 *rooms connected through the middle row.*

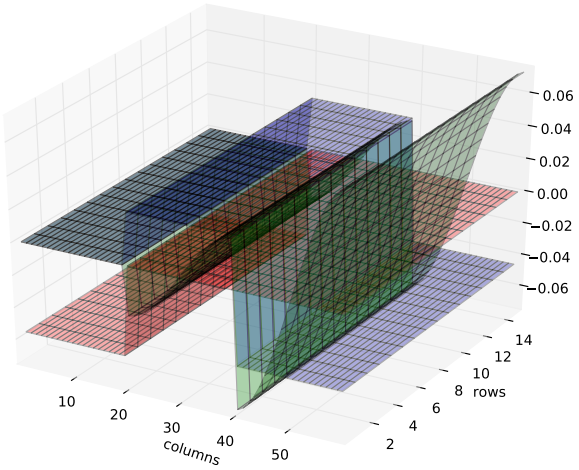


Fig. 5: *Second to fourth eigenvectors of the laplacian of the three rooms graph. Note how the eigendecomposition automatically captures the structure of the environment.*

Visualizing the LSA Space

We believe that being able to work in a vector space will allow us to use a series of RL techniques that otherwise we would not be available to use. For example, when using Proto-value functions, it is possible to use the Nyström approximation to estimate the value of an eigenvector for out-of-sample states [Mah06]; this is only possible if states can be represented as points belonging to a Euclidean space.

How can we embed an entity in Euclidean space? In the previous section we showed that LSA can effectively compute the similarity between documents. We can take this concept one step forward and use LSA not only for computing similarities, but also for embedding documents in Euclidean space.

To evaluate the soundness of this idea, we perform an exploratory analysis of the simple Wikipedia LSA space. In order to be able to visualize the vectors, we use ISOMAP [Ten00] to reduce the dimension of the LSA vectors from 200 to 3 (we use the ISOMAP implementation provided by scikit-learn [Ped11]). We show a typical result in Fig. 6, where each point represents the LSA embedding of an article in $\mathbb{R}^3$, and a line between two points represents a link between two articles. We can see how the points close to the "Water" article are, in effect, semantically related ("Fresh water", "Lake", "Snow", etc.). This result confirms that the

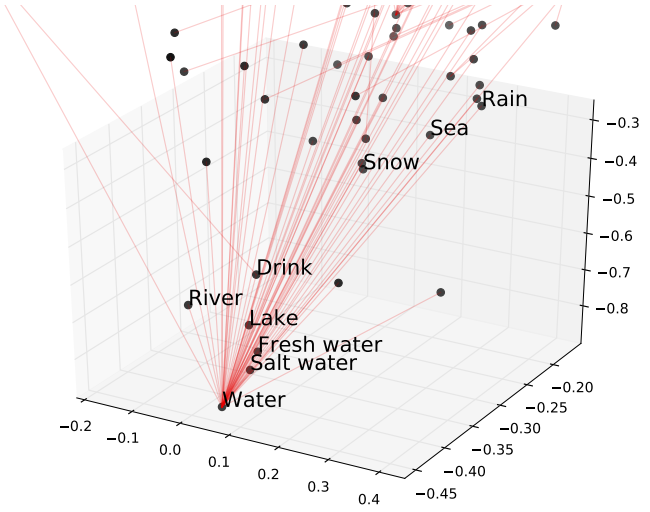


Fig. 6: *ISOMAP projection of the LSA space. Each point represents the LSA vector of a Simple English Wikipedia article projected onto $\mathbb{R}^3$ using ISOMAP. A line is added if there is a link between the corresponding articles. The figure shows a close-up around the "Water" article. We can observe that this point is close to points associated to articles with a similar semantic.*

LSA representation is not only useful for computing similarities between documents, but it is also an effective mechanism for embedding the information entities into a Euclidean space. This result encourages us to propose the use of the LSA representation in the definition of the state.

Once again we emphasize that since Gensim vectors are NumPY arrays, we can use its output as an input to scikit-learn without any effort.

Conclusions

We have presented an example where we use different elements of the scientific Python ecosystem to solve a research problem. Since we use libraries where NumPy arrays are used as the standard vector/matrix format, the integration among these components is transparent. We believe that this work is a good success story that validates Python as a viable scientific programming language.

Our work shows that in many cases it is advantageous to use general purposes languages, like Python, for scientific computing. Although some computational parts of this work might be somewhat simpler to implement in a domain specific language,[9] the breadth of tasks that we work with could make it hard to integrate all of the parts using a domain specific language.

Acknowledgment

This work was partially supported by AFOSR grant FA9550-09-1-0465.

References

[Tan09] B. Tanner and A. White. *RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments*, Journal of Machine Learning Research, 10(Sep):2133-2136, 2009

[Hag08] A. Hagberg, D. Schult and P. Swart, *Exploring Network Structure, Dynamics, and Function using NetworkX*, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11-15, Aug 2008

9. Examples of such languages are MATLAB, Octave, SciLab, etc.

[Ped11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay. *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 12:2825-2830, 2011

[Reh10] R. Řehůřek and P. Sojka. *Software Framework for Topic Modelling with Large Corpora*, in Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45-50 May 2010

[Sze10] C. Szepesvári. *Algorithms for Reinforcement Learning*. San Rafael, CA, Morgan and Claypool Publishers, 2010.

[Sut98] R.S. Sutton and A.G. Barto. *Reinforcement Learning*. Cambridge, Massachusetts, The MIT press, 1998.

[Mah07] S. Mahadevan and M. Maggioni. *Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes*. Journal of Machine Learning Research, 8:2169-2231, 2007.

[Man08] C.D. Manning, P. Raghavan and H. Schutze. *An introduction to information retrieval*. Cambridge, England. Cambridge University Press, 2008

[Ten00] J.B Tenenbaum, V. de Silva, and J.C. Langford. *A global geometric framework for nonlinear dimensionality reduction* . Science, 290(5500), 2319-2323, 2000

[Mah06] S. Mahadevan, M. Maggioni, K. Ferguson and S.Osentoski. *Learning representation and control in continuous Markov decision processes*. National Conference on Artificial Intelligence, 2006.

[Dee90] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer and R. Harshman, R. (1990). *Indexing by latent semantic analysis.* Journal of the American Society for Information Science, 41(6), 391-407.

Total Recall: flmake and the Quest for Reproducibility

Anthony Scopatz‡*

Abstract—FLASH is a high-performance computing (HPC) multi-physics code which is used to perform astrophysical and high-energy density physics simulations. To run a FLASH simulation, the user must go through three basic steps: setup, build, and execution. Canonically, each of these tasks are independently handled by the user. However, with the recent advent of flmake - a Python workflow management utility for FLASH - such tasks may now be performed in a fully reproducible way. To achieve such reproducibility a number of developments and abstractions were needed, some only enabled by Python. These methods are widely applicable outside of FLASH. The process of writing flmake opens many questions to what precisely is meant by reproducibility in computational science. While posed here, many of these questions remain unanswered.

Index Terms—FLASH, reproducibility

Introduction

FLASH is a high-performance computing (HPC) multi-physics code which is used to perform astrophysical and high-energy density physics simulations [FLASH]. It runs on the full range of systems from laptops to workstations to 100,000 processor super computers, such as the Blue Gene/P at Argonne National Laboratory.

Historically, FLASH was born from a collection of unconnected legacy codes written primarily in Fortran and merged into a single project. Over the past 13 years major sections have been rewritten in other languages. For instance, I/O is now implemented in C. However building, testing, and documentation are all performed in Python.

FLASH has a unique architecture which compiles *simulation specific* executables for each new type of run. This is aided by an object-oriented-esque inheritance model that is implemented by inspecting the file system directory tree. This allows FLASH to compile to faster machine code than a compile-once strategy. However it also places a greater importance on the Python build system.

To run a FLASH simulation, the user must go through three basic steps: setup, build, and execution. Canonically, each of these tasks are independently handled by the user. However with the recent advent of flmake - a Python workflow management utility for FLASH - such tasks may now be performed in a repeatable way [FLMAKE].

Previous workflow management tools have been written for FLASH. (For example, the "Milad system" was implemented entirely in Makefiles.) However, none of the prior attempts have placed reproducibility as their primary concern. This is in part because fully capturing the setup metadata required alterations to the build system.

The development of flmake started by rewriting the existing build system to allow FLASH to be run outside of the mainline subversion repository. It separates out a project (or simulation) directory independent of the FLASH source directory. This directory is typically under its own version control.

For each of the important tasks (setup, build, run, etc), a sidecar metadata *description* file is either initialized or modified. This is a simple dictionary-of-dictionaries JSON file which stores the environment of the system and the state of the code when each flmake command is run. This metadata includes the version information of both the FLASH mainline and project repositories. However, it also may include all local modifications since the last commit. A patch is automatically generated using standard posix utilities and stored directly in the description.

Along with universally unique identifiers, logging, and Python run control files, the flmake utility may use the description files to fully reproduce a simulation by re-executing each command in its original state. While `flmake reproduce` makes a useful debugging tool, it fundamentally increases the scientific merit of FLASH simulations.

The methods described herein may be used whenever source code itself is distributed. While this is true for FLASH (uncommon amongst compiled codes), most Python packages also distribute their source. Therefore the same reproducibility strategy is applicable and highly recommended for Python simulation codes. Thus flmake shows that reproducibility - which is notably absent from most computational science projects - is easily attainable using only version control, Python standard library modules, and ever-present command line utilities.

New Workflow Features

As with many predictive science codes, managing FLASH simulations may be a tedious task for both new and experienced users. The flmake command line utility eases the simulation burden and shortens the development cycle by providing a modular tool which implements many common elements of a FLASH workflow. At each stage this tool captures necessary metadata about the task which it is performing. Thus flmake encapsulates the following operations:

- setup/configuration,
- building,
- execution,
- logging,

* *Corresponding author: scopatz@flash.uchicago.edu*
‡ *The FLASH Center for Computational Science, The University of Chicago*

- analysis & post-processing,
- and others.

It is highly recommended that both novice and advanced users adopt flmake as it *enables* reproducible research while simultaneously making FLASH easier to use. This is accomplished by a few key abstractions from previous mechanisms used to set up, build, and execute FLASH. The implementation of these abstractions are critical flmake features and are discussed below. Namely they are the separation of project directories, a searchable source path, logging, dynamic run control, and persisted metadata descriptions.

Independent Project Directories

Without flmake, FLASH must be setup and built from within the FLASH source directory (`FLASH_SRC_DIR`) using the setup script and make [GMAKE]. While this is sufficient for single runs, such a strategy fails to separate projects and simulation campaigns from the source code. Moreover, keeping simulations next to the source makes it difficult to track local modifications independent of the mainline code development.

Because of these difficulties in running suites of simulations from within `FLASH_SRC_DIR`, flmake is intended to be run external to the FLASH source directory. This is known as the project directory. The project directory should be managed by its own version control systems. By doing so, all of the project-specific files are encapsulated in a repository whose history is independent from the main FLASH source. Here this directory is called `proj/` though in practice it takes the name of the simulation campaign. This directory may be located anywhere on the user's file system.

Source & Project Paths Searching

After creating a project directory, the simulation source files must be assembled using the flmake setup command. This is analogous to executing the traditional setup script. For example, to run the classic Sedov problem:

```
~/proj $ flmake setup Sedov -auto
[snip]
SUCCESS
~/proj $ ls
flash_desc.json  setup/
```

This command creates symbolic links to the the FLASH source files in the `setup/` directory. Using the normal FLASH setup script, all of these files must live within `${FLASH_SRC_DIR}/source/`. However, the flmake setup command searches additional paths to find potential source files.

By default if there is a local `source/` directory in the project directory then this is searched first for any potential FLASH units. The structure of this directory mirrors the layout found in `${FLASH_SRC_DIR}/source/`. Thus if the user wanted to write a new or override an existing driver unit, they could place all of the relevant files in `~/proj/source/Driver/`. Units found in the project source directory take precedence over units with the same name in the FLASH source directory.

The most commonly overridden units, however, are simulations. Yet specific simulations live somewhat deep in the file system hierarchy as they reside within `source/Simulation/SimulationMain/`. To make accessing simulations easier a local project `simulations/` directory is first searched for any possible simulations. Thus `simulations/` effectively aliases `source/Simulation/SimulationMain/`. Continuing with the previous Sedov example the following directories are searched in order of precedence for simulation units, if they exist:

1) `~/proj/simulations/Sedov/`
2) `~/proj/source/Simulation/SimulationMain/Sedov/`
3) `${FLASH_SRC_DIR}/source/Simulation/SimulationMain/Sedov/`

Therefore, it is common for a project directory to have the following structure if the project requires many modifications to FLASH that are - at least in the short term - inappropriate for mainline inclusion:

```
~/proj $ ls
flash_desc.json  setup/  simulations/  source/
```

Logging

In many ways computational simulation is more akin to experimental science than theoretical science. Simulations are executed to test the system at hand in analogy to how physical experiments probe the natural world. Therefore, it is useful for computational scientists to adopt the time-tested strategy of a keeping a lab notebook or its electronic analogy.

Various example of virtual lab notebooks exist [VLABNB] as a way of storing information about how an experiment was conducted. The resultant data is often stored in conjunction with the notebook. Arguably the corollary concept in software development is logging. Unfortunately, most simulation science makes use of neither lab notebooks nor logging. Rather than using an external rich- or web-client, flmake makes use of the built-in Python logger.

Every flmake command has the ability to log a message. This follows the `-m` convention from version control systems. These messages and associated metadata are stored in a `flash.log` file in the project directory.

Not every command uses logging; for trivial commands which do not change state (such as listing or diffing) log entries are not needed. However for more serious commands (such as building) logging is a critical component. Understanding that many users cannot be bothered to create meaningful log messages at each step, sensible and default messages are automatically generated. Still, it is highly recommended that the user provide more detailed messages as needed. *E.g.*:

```
~/proj $ flmake -m "Run with 600 J laser" run -n 10
```

The `flmake log` command may then be used to display past log messages:

```
~/proj $ flmake log -n 1
Run id: b2907415
Run dir: run-b2907415
Command: run
User: scopatz
Date: Mon Mar 26 14:20:46 2012
Log id: 6b9e1a0f-cfdc-418f-8c50-87f66a63ca82

    Run with 600 J laser
```

The `flash.log` file should be added to the version control of the project. Entries in this file are not typically deleted.

Dynamic Run Control

Many aspects of FLASH are declared in a static way. Such declarations happen mainly at setup and runtime. For certain build and run operations several parameters may need to be altered in a consistent way to actually have the desired effect. Such repetition can become tedious and usually leads to less readable inputs.

To make the user input more concise and expressive, flmake introduces a run control `flashrc.py` file in the project directory. This is a Python module which is executed, if it exists, in an empty namespace whenever flmake is called. The flmake commands may then choose to access specific data in this file. Please refer to individual command documentation for an explanation on if/how the run control file is used.

The most important example of using `flashrc.py` is that the run and restart commands will update the `flash.par` file with values from a `parameters` dictionary (or function which returns a dictionary).

Initial `flash.par`

```
order = 3
slopeLimiter = "minmod"
charLimiting = .true.
RiemannSolver = "hll"
```

Run Control `flashrc.py`

```
parameters = {"slopeLimiter": "mc",
              "use_flattening": False}
```

Final `flash.par`

```
RiemannSolver = "hll"
charLimiting = .true.
order = 3
slopeLimiter = "mc"
use_flattening = .false.
```

Description Sidecar Files

As a final step, the setup command generates a `flash_desc.json` file in the project directory. This is the description file for the FLASH simulation which is currently being worked on. This description is a sidecar file whose purpose is to store the following metadata at execution of each flmake command:

- the environment,
- the version of both project and FLASH source repository,
- local source code modifications (diffs),
- the run control files (see above),
- run ids and history,
- and FLASH binary modification times.

Thus the description file is meant to be a full picture of the way FLASH code was generated, compiled, and executed. Total reproducibility of a FLASH simulation is based on having a well-formed description file.

The contents of this file are essentially a persisted dictionary which contains the information listed above. The top level keys include setup, build, run, and merge. Each of these keys gets added when the corresponding flmake command is called. Note that restart alters the run value and does not generate its own top-level key.

During setup and build, `flash_desc.json` is modified in the project directory. However, each run receives a copy of this file in the run directory with the run information added. Restarts and merges inherit from the file in the previous run directory.

These sidecar files enable the flmake reproduce command which is capable of recreating a FLASH simulation from only the `flash_desc.json` file and the associated source and project repositories. This is useful for testing and verification of the same simulation across multiple different machines and platforms. It is generally not recommended that users place this file under version control as it may change often and significantly.

Example Workflow

The fundamental flmake abstractions have now been explained above. A typical flmake workflow which sets up, builds, runs, restarts, and merges a fork of a Sedov simulation is now demonstrated. First, construct the project repository:

```
~ $ mkdir my_sedov
~ $ cd my_sedov/
~/my_sedov $ mkdir simulations/
~/my_sedov $ cp -r ${FLASH_SRC_DIR}/source/\
             Simulation/SimulationMain/Sedov
             simulations/
~/my_sedov $ # edit the simulation
~/my_sedov $ nano simulations/Sedov/\
             Simulation_init.F90
~/my_sedov $ git init .
~/my_sedov $ git add .
~/my_sedov $ git commit -m "My Sedov project"
```

Next, create and run the simulation:

```
~/my_sedov $ flmake setup -auto Sedov
~/my_sedov $ flmake build -j 20
~/my_sedov $ flmake -m "First run of my Sedov" \
                                      run -n 10
~/my_sedov $ flmake -m "Oops, it died." restart \
                           run-5a4f619e/ -n 10
~/my_sedov $ flmake -m "Merging my first run." \
                    merge run-fc6c9029 first_run
~/my_sedov $ flmake clean 1
```

Why Reproducibility is Important

True to its part of speech, much of 'scientific computing' has the trappings of science in that it is code produced to solve problems in (big-'S') Science. However, the process by which said programs are written is not typically itself subject to the rigors of the scientific method. The vaulted method contains components of prediction, experimentation, duplication, analysis, and openness [GODFREY-SMITH]. While software engineers often engage in such activities when programming, scientific developers usually forgo these methods, often to their detriment [WILSON].

Whatever the reason for this may be - ignorance, sloth, or other deadly sins - the impetus for adopting modern software development practices only increases every year. The evolution of tools such as version control and environment capturing mechanisms (such as virtual machines/hypervisors) enable researchers to more easily retain information about software during and

after production. Furthermore, the apparent end of Silicon-based Moore's Law has necessitated a move to more exotic architectures and increased parallelism to see further speed increases [MIMS]. This implies that code that runs on machines now may not be able to run on future processors without significant refactoring.

Therefore the scientific computing landscape is such that there are presently the tools and the need to have fully reproducible simulations. However, most scientists choose to not utilize these technologies. This is akin to a chemist not keeping a lab notebook. The lack of reproducibility means that many solutions to science problems garnered through computational means are relegated to the realm of technical achievements. Irreproducible results may be novel and interesting but they are not science. Unlike the current paradigm of computing-about-science, or *periscientific computing*, reproducibility is a keystone of *diacomputational science* (computing-throughout-science).

In periscientific computing there may exist a partition between expert software developers and expert scientists, each of whom must learn to partially speak the language of the other camp. Alternatively, when expert software engineers are not available, canonically ill-equipped scientists perform only the bare minimum development to solve computing problems.

On the other hand, in diacomputational science, software exists as a substrate on top of which science and engineering problems are solved. Whether theoretical, simulation, or experimental problems are at hand the scientist has a working knowledge of computing tools available and an understanding of how to use them responsibly. While the level of education required for diacomputational science may seem extreme in a constantly changing software ecosystem, this is in fact no greater than what is currently expect from scientists with regard to Statistics [WILSON].

With the extreme cases illustrated above, there are some notable exceptions. The first is that there are researchers who are cognizant and respectful of these reproducibility issues. The efforts of these scientists help paint a less dire picture than the one framed here.

The second exception is that while reproducibility is a key feature of fundamental science it is not the only one. For example, openness is another point whereby the statement "If a result is not produced openly then it is not science" holds. Open access to results - itself is a hotly contested issue [VRIEZE] - is certainly a component of computational science. Though having open and available code is likely critical for pure science, it often lies outside the scope of normal research practice. This is for a variety of reasons, including the fear that releasing code too early or at all will negatively impact personal publication records. Tackling the openness issue must be left for another paper.

In summary, reproducibility is important because without it any results generated are periscientific. To achieve diacomputational science there exist computational tools to aid in this endeavor, as in analogue science there are physical solutions. Though it is not the only criticism to be levied against modern research practices, irreproducibility is one that affects computation acutely and uniquely as compared to other spheres.

The Reproduce Command

The `flmake reproduce` command is the key feature enabling the total reproducibility of a FLASH simulation. This takes a description file (e.g. `flash_desc.json`) and implicitly the FLASH source and project repositories and replays the setup, build, and run commands originally executed. It has the following usage string:

```
flmake reproduce [options] <flash_descr>
```

For each command, reproduction works by cloning both source and project repositories at a the point in history when they were run into temporary directories. Then any local modifications which were present (and not under version control) are loaded from the description file and applied to the cloned repository. It then copies the original run control file to the cloned repositories and performs any command-specific modifications needed. Finally, it executes the appropriate command *from the cloned repository* using the original arguments provided on the command line. Figure 1 presents a flow sheet of this process.

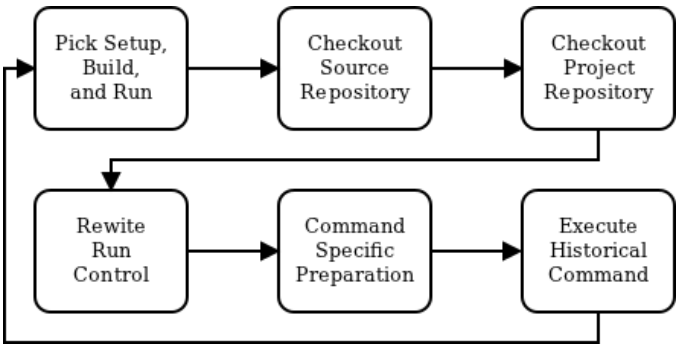


Fig. 1: *The reproduce command workflow.*

Thus the `flmake reproduce` recreates the original simulation using the original commands (and not the versions currently present). The reproduce command has the following limitations:

1) Source directory must be version controlled,
2) Project directory must be version controlled,
3) The FLASH run must depend on only the runtime parameters file, the FLASH executable and FLASH datafiles,
4) and the FLASH executable must not be modified between build and run steps.

The above restrictions enforce that the run is not considered reproducible if at any point FLASH depends on externalities or alterations not tracked by version control. Critical to this process are version control abstractions and the capability to execute historical commands. These will be discussed in the following subsections.

Meta-Version Control

Every user and developer tends towards one version control system or another. The mainline FLASH development team operates in subversion [SVN] though individual developers may prefer git [GIT] or mercurial [HG]. As mentioned previously, some users do not employ any source control management software.

In the case where the user lacks a sophisticated version control system, it is still possible to obtain reproducibility *if* a clean directory tree of a recent release is available. This clean tree must be stored in a known place, typically the `.clean/` subdirectory of the `FLASH_SRC_DIR`. This is known as the 'release' versioning system and is managed entirely by flmake.

To realize reproducibility in this environment, it is necessary for the reproduce command to abstract core source control management features away from the underlying technology (or absence of technology). For flmake, the following operations define version control in the context of reproducibility:

- info,
- checkout or clone,
- diff,
- and patch.

The info operation provides version control information that points to the current state of the repository. For all source control management schemes this includes a unique string id for the versioning type (e.g. 'svn' for subversion). In centralized version control this contains the repository version number, while for for distributed systems info will return the branch name and the hash of the current HEAD. In the release system, info simply returns the release version number. The info data that is found is then stored in the description file for later use.

The checkout (or sometimes clone) operation is effectively the inverse operation to info. This operation takes a point in history, as described by the data garnered from info, and makes a temporary copy of the whole repository at this point. Thus no matter what evolution the code has undergone since the description file was written, checkout rolls back the source to its previous incarnation. For centralized version control this operation copies the existing tree, reverts it to a clean working tree of HEAD, and performs a reverse merge on all commits from HEAD to the historical target. For distributed systems this clones the current repository, checkouts or updates to the historical position, and does a hard reset to clean extraneous files. The release system is easiest in that checkout simply copies over the clean subdirectory. This operation is performed for the setup, build, and run commands at reproduce time.

The diff operation may seem less than fundamental to version control. Here however, diff is used to capture local modifications to the working trees of the source and project directories. This diffing is in place as a fail-safe against uncommitted changes. For centralized and distributed systems, diffing is performed through the selfsame command name. In the release system (where committing is impossible), diffing takes on the heavy lifting not provided by a more advanced system. Thus for the release system diff is performed via the posix `diff` tool with the recursive switch between the `FLASH_SRC_DIR` and the clean copy. The diff operation is executed when the commands are originally run. The resultant diff string is stored in the description file, along with the corresponding info.

The inverse operation to diff is patch. This is used at reproduce time after checkout to restore the working trees of the temporary repositories to the same state they were in at the original execution of setup, build, and run. While each source control management system has its own patching mechanism, the output of diff always returns a string which is compatible with the posix `patch` utility. Therefore, for all systems the `patch` program is used.

The above illustrates how version control abstraction may be used to define a set of meta-operations which capture all versioning information provided. This even included the case where no formal version control system was used. It also covers the case of the 'forgetful' user who may not have committed every relevant local change to the repository prior to running a simulation. What is more is that the flmake implementation of these abstractions is only a handful of functions. These total less than 225 lines of code in Python. Though small, this capability is vital to the reproduce command functioning as intended.

Command Time Machine

Another principal feature of flmake reproducibility is its ability to execute historical versions of the key commands (setup, build, and run) as reincarnated by the meta-version control. This is akin to the bootstrapping problem whereby all of the instruction needed to reproduce a command are contained in the original information provided. Without this capability, the most current versions of the flmake commands would be acting on historical versions of the repository. While such a situation would be a large leap forward for the reproducibility of FLASH simulations, it falls well short of total reproducibility. In practice, therefore, historical flmake commands acting on historical source are needed. This maybe be termed the 'command time machine,' though it only travels into the past.

The implementation of the command time machine requires the highly dynamic nature of Python, a bit of namespace slight-of-hand, and relative imports. First note that module and package which are executing the flmake reproduce command may not be deleted from the `sys.modules` cache. (Such a removal would cause sudden and terrifying runtime failures.) This effectively means that everything under the `flash` package name may not be modified.

Nominally, the historical version of the package would be under the `flash` namespace as well. However, the name `flash` is only given at install time. Inside of the source directory, the package is located in `tools/python/`. This allows the current reproduce command to add the checked out and patched `{temp-flash-src-dir}/tools/` directory to the front of `sys.path` for setup, build, and run. Then the historical flmake may be imported via `python.flmake` because `python/` is a subdirectory of `{temp-flash-src-dir}/tools/`.

Modules inside of `python` or `flmake` are guaranteed to import other modules in their own package because of the exclusive use of relative imports. This ensures that the old commands import old commands rather then mistakenly importing newer iterations.

Once the historical command is obtained, it is executed with the original arguments from the description file. After execution, the temporary source directory `{temp-flash-src-dir}/tools/` is removed from `sys.path`. Furthermore, any module whose name starts with `python` is also deleted from `sys.modules`. This cleans the environment for the next historical command to be run in its own temporal context.

In effect, the current version of flmake is located in the `flmake` namespace and should remain untouched while the reproduce command is running. Simultaneously, the historic flmake commands are given the namespace `python`. The time value of `python` changes with each command reproduced but is fully independent from the current flmake code. This method of renaming a package namespace on the file system allows for one version of flmake to supervise the execution of another in a manner relevant to reproducibility.

A Note on Replication

A weaker form of reproducibility is known as *replication* [SCHMIDT]. Replication is the process of recreating a result when "you take all the same data and all the same tools" [GRAHAM] which were used in the original determination. Replication is a weaker determination than reproduction because at minimum the original scientist should be able to replicate their own work. Without replication, the same code executed twice will produce distinct results. In this case no trust may be placed in the conclusions whatsoever.

Much as version control has given developers greater control over reproducibility, other modern tools are powerful instruments of replicability. Foremost among these are hypervisors. The ease-of-use and ubiquity of virtual machines (VM) in the software ecosystem allows for the total capture and persistence of the environment in which any computation was performed. Such environments may be hosted and shared with collaborators, editors, reviewers, or the public at large. If the original analysis was performed in a VM context, shared, and rerun by other scientists then this is replicability. Such a strategy has been proposed by C. T. Brown as a stop-gap measure until diacomputational science is realized [BROWN].

However, as Brown admits (see comments), the delineation between replication and reproduction is fuzzy. Consider these questions which have no clear answers:

- Are bit-identical results needed for replication?
- How much of the environment must be reinstated for replication versus reproduction?
- How much of the hardware and software stack must be recreated?
- What precisely is meant by 'the environment' and how large is it?
- For codes depending on stochastic processes, is reusing the same random seed replication or reproduction?

Without justifiable answers to the above, ad hoc definitions have governed the use of replicability and reproducibility. Yet to the quantitatively minded, an I-know-reproducibility-when-I-see-it approach falls short. Thus the science of science, at least in the computational sphere, has much work remaining.

Even with the reproduction/replication dilemma, the flmake reproduce command *is* a reproducibility tool. This is because it takes the opposite approach to Brown's VM-based replication. Though the environment is captured within the description file, flmake reproduce does not attempt to recreate this original environment at all. The previous environment information is simply there for posterity, helping to uncover any discrepancies which may arise. User specific settings on the reproducing machine are maintained. This includes but is not limited to which compiler is used.

The claim that Brown's work and flmake reproduce represent paragons of replicability and reproducibility respectively may be easily challenged. The author, like Brown himself, does not presuppose to have all - or even partially satisfactory - answers. What is presented here is an attempt to frame the discussion and bound the option space of possible meanings for these terms. Doing so with concrete code examples is preferable to debating this issue in the abstract.

Conclusions & Future Work

By capturing source code and the environment at key stages - setup, build, and run - FLASH simulations may be fully reproduced in the future. Doing so required a wrapper utility called flmake. The writing of this tool involved an overhaul of the existing system. Though portions of flmake took inspiration from previous systems none were as comprehensive. Additionally, to the author's knowledge, no previous system included a mechanism to non-destructively execute previous command incarnations similar to flmake reproduce.

The creation of flmake itself was done as an exercise in reproducibility. What has been shown here is that it is indeed possible to increase the merit of simulation science through a relatively small, though thoughtful, amount of code. It is highly encouraged that the methods described here be copied by other software-in-science project*.

Moreover, in the process of determining what flmake *should* be, several fundamental questions about reproducibility itself were raised. These point to systemic issues within the realm of computational science. With the increasing importance of computing, soon science as a whole will also be forced to reconcile these reproducibility concerns. Unfortunately, there does not appear to be an obvious and present solution to the problems posed.

As with any software development project, there are further improvements and expansions that will continue to be added to flmake. More broadly, the questions posed by reproducibility will be the subject of future work on this project and others. Additional issues (such as openness) will also figure into subsequent attempts to bring about a global state of diacomputational science.

Acknowledgements

Dr. Milad Fatenejad provided a superb sounding board in the conception of the flmake utility and aided in outlining the constraints of reproducibility.

The software used in this work was in part developed by the DOE NNSA-ASC OASCR Flash Center at the University of Chicago.

REFERENCES

[BROWN] C. Titus Brown, "Our approach to replication in computational science," Living in an Ivory Basement, April 2012, http://ivory.idyll.org/blog/replication-i.html.

[FLASH] FLASH Center for Computational Science, *FLASH User's Guide, Version 4.0-beta,* http://flash.uchicago.edu/site/flashcode/user_support/flash4b_ug.pdf, University of Chicago, February 2012.

[FLMAKE] A. Scopatz, *flmake: the flash workflow utility,* http://flash.uchicago.edu/site/flashcode/user_support/tools4b/usersguide/flmake/index.html, The University of Chicago, June 2012.

[GIT] Scott Chacon, "Pro Git," Apress (2009) DOI: 10.1007/978-1-4302-1834-0

[GMAKE] Free Software Foundation, The GNU Make Manual for version 3.82, http://www.gnu.org/software/make/, 2010.

[GODFREY-SMITH] Godfrey-Smith, Peter (2003), *Theory and Reality: An introduction to the philosophy of science*, University of Chicago Press, ISBN 0-226-30063-3.

[GRAHAM] Jim Graham, "What is 'Reproducibility,' Anyway?", Scimatic, April 2010, http://www.scimatic.com/node/361.

*. Please contact the author if you require aid in any reproducibility endeavours.

[HG] Bryan O'Sullivan, "Mercurial: The Definitive Guide," O'Reilly Media, Inc., 2009.

[MIMS] C. Mims, *Moore's Law Over, Supercomputing "In Triage," Says Expert,* http://www.technologyreview.com/view/427891/moores-law-over-supercomputing-in-triage-says/ May 2012, Technology Review, MIT.

[SCHMIDT] Gavin A. Schmidt, "On replication," RealClimate, Feb 2009, http://www.realclimate.org/index.php/archives/2009/02/on-replication/langswitch_lang/in/.

[SVN] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato (2011). "Version Control with Subversion: For Subversion 1.7". O'Reilly.

[VLABNB] Rubacha, M.; Rattan, A. K.; Hosselet, S. C. (2011). *A Review of Electronic Laboratory Notebooks Available in the Market Today*. Journal of Laboratory Automation 16 (1): 90–98. DOI:10.1016/j.jala.2009.01.002. PMID 21609689.

[VRIEZE] Jop de Vrieze, *Thousands of Scientists Vow to Boycott Elsevier to Protest Journal Prices,* Science Insider, February 2012.

[WILSON] G.V. Wilson, *Where's the real bottleneck in scientific computing?* Am Sci. 2005;94:5.

Python's Role in VisIt

Cyrus Harrison[‡*], Harinarayan Krishnan[§]

Abstract—VisIt is an open source, turnkey application for scientific data analysis and visualization that runs on a wide variety of platforms from desktops to petascale class supercomputers. VisIt's core software infrastructure is written in C++, however Python plays a vital role in enabling custom workflows. Recent work has extended Python's use in VisIt beyond scripting, enabling custom Python UIs and Python filters for low-level data manipulation. The ultimate goal of this work is to evolve Python into a true peer to our core C++ plugin infrastructure. This paper provides an overview of Python's role in VisIt with a focus on use cases of scripted rendering, data analysis, and custom application development.

Index Terms—visualization, hpc, python

Introduction

VisIt [VisIt05], like EnSight [EnSight09] and ParaView [ParaView05], is an application designed for post processing of mesh based scientific data. VisIt's core infrastructure is written in C++ and it uses VTK [VTK96] for its underlying mesh data model. Its distributed-memory parallel architecture is tailored to process domain decomposed meshes created by simulations on large-scale HPC clusters.

Early in development, the VisIt team adopted Python as the foundation of VisIt's primary scripting interface. The scripting interface is available from both a standard Python interpreter and a custom command line client. The interface provides access to all features available through VisIt's GUI. It also includes support for macro recording of GUI actions to Python snippets and full control of windowless batch processing.

While Python has always played an important scripting role in VisIt, two recent development efforts have greatly expanded VisIt's Python capabilities:

1) We now support custom UI development using Qt via PySide [PySide]. This allows users to embed VisIt's visualization windows into their own Python applications. This provides a path to extend VisIt's existing GUI and for rapid development of streamlined UIs for specific use cases.
2) We recently enhanced VisIt by embedding Python interpreters into our data flow network pipelines. This provides fine grained access, allowing users to write custom algorithms in Python that manipulate mesh data via VTK's Python wrappers and leverage packages such as NumPy [NumPy] and SciPy [SciPy]. Current support includes the ability to create derived mesh quantities and execute data summarization operations.

* Corresponding author: cyrush@llnl.gov
‡ Lawrence Livermore National Laboratory
§ Lawrence Berkeley National Laboratory

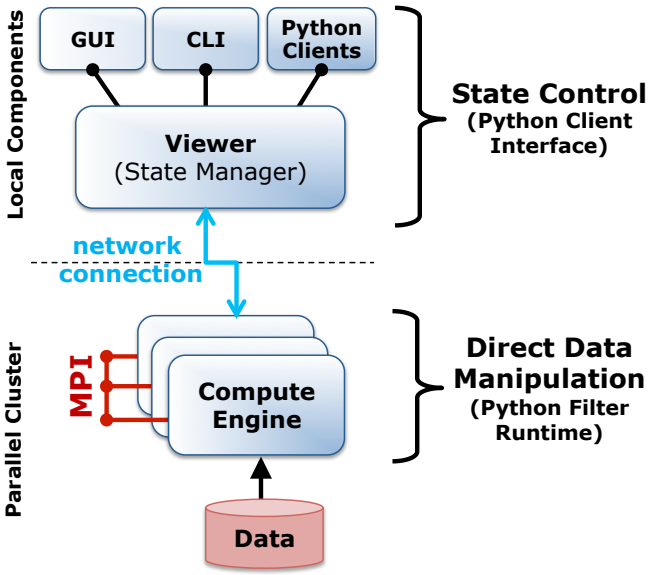


Fig. 1: *Python integration with VisIt's components.*

This paper provides an overview of how VisIt leverages Python in its software architecture, outlines these two recent Python feature enhancements, and introduces several examples of use cases enabled by Python.

Python Integration Overview

VisIt employs a client-server architecture composed of several interacting software components:

- A *viewer* process coordinates the state of the system and provides the visualization windows used to display data.
- A set of *client* processes, including a Qt-based GUI and Python-based command line interface (CLI), are used to setup plots and direct visualization operations.
- A parallel *compute engine* executes the visualization pipelines. This component employs a data flow network design and uses MPI for communication in distributed-memory parallel environments.

Client and *viewer* proceses are typically run on a desktop machine and connect to a parallel *compute engine* running remotely

on a HPC cluster. For smaller data sets, a local serial or parallel *compute engine* is also commonly used.

Figure 1 outlines how Python is integrated into VisIt's components. VisIt both extends and embeds Python. State control of the *viewer* is provided by a Python Client Interface, available as Python/C extension module. This interface is outlined in the Python Client Interface section, and extensions to support custom UIs written in Python are described in the Custom Python UIs section. Direct access to low-level mesh data structures is provided by a Python Filter Runtime, embedded in VisIt's *compute engine* processes. This runtime is described in the Python Filter Runtime section.

Python Client Interface

VisIt clients interact with the *viewer* process to control the state of visualization windows and data processing pipelines. Internally the system uses a collection of state objects that rely on a publish/subscribe design pattern for communication among components. These state objects are wrapped by a Python/C extension module to expose a Python state control API. The function calls are typically imperative: *Add a new plot*, *Find the maximum value of a scalar field*, etc. The client API is documented extensively in the VisIt Python Interface Manual [VisItPyRef]. To introduce the API in this paper we provide a simple example script, Listing 1, that demonstrates VisIt's five primary visualization building blocks:

- **Databases**: File readers and data sources.
- **Plots**: Data set renderers.
- **Operators**: Filters implementing data set transformations.
- **Expressions**: Framework enabling the creation of derived quantities from existing mesh fields.
- **Queries**: Data summarization operations.

Listing 1: Trace streamlines along the gradient of a scalar field.

```
# Open an example file
OpenDatabase("noise.silo")
# Create a plot of the scalar field 'hardyglobal'
AddPlot("Pseudocolor","hardyglobal")
# Slice the volume to show only three
# external faces.
AddOperator("ThreeSlice")
tatts = ThreeSliceAttributes()
tatts.x = -10
tatts.y = -10
tatts.z = -10
SetOperatorOptions(tatts)
DrawPlots()
# Find the maximum value of the field 'hardyglobal'
Query("Max")
val = GetQueryOutputValue()
print "Max value of 'hardyglobal' = ", val
# Create a streamline plot that follows
# the gradient of 'hardyglobal'
DefineVectorExpression("g","gradient(hardyglobal)")
AddPlot("Streamline","g")
satts = StreamlineAttributes()
satts.sourceType = satts.SpecifiedBox
satts.sampleDensity0 = 7
satts.sampleDensity1 = 7
satts.sampleDensity2 = 7
satts.coloringMethod = satts.ColorBySeedPointID
SetPlotOptions(satts)
DrawPlots()
```

In this example, the Silo database reader is automatically selected to read meshes from the input file 'noise.silo'. A *Pseudocolor* plot is created to display the scalar field named 'hardyglobal'. The mesh is transformed by a *ThreeSlice* operator to limit the volume displayed by the *Pseudocolor* plot to three external faces. We use a query to obtain and print the maximum value of the 'hardyglobal' field. An expression is defined to extract the gradient of the 'hardyglobal' scalar field. Finally, this gradient vector is used as the input field for a second plot, which traces streamlines. Figure 2 shows the resulting visualization which includes both the *Pseudocolor* and *Streamline* plots.

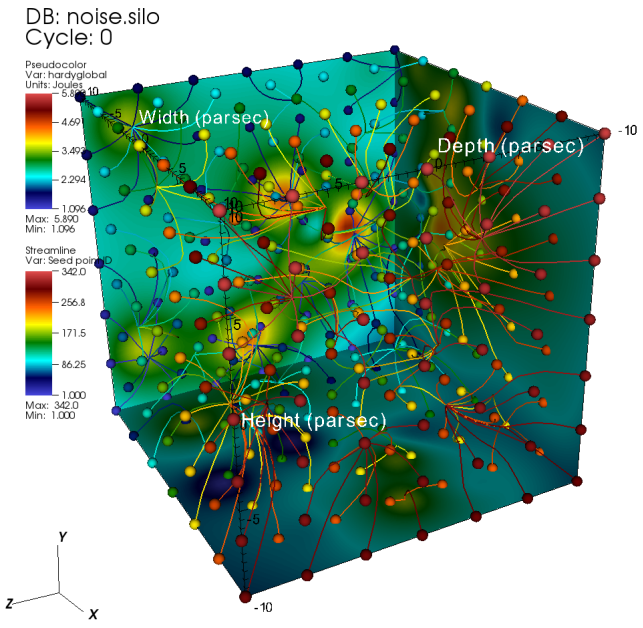


Fig. 2: Pseudocolor and Streamline plots setup using the script in Listing 1.

Accessing the Python Client Interface

For convenience, you can access the client interface from a custom binary or a standalone Python interpreter.

VisIt provides a command line interface (CLI) binary that embeds a Python interpreter and automatically imports the client interface module. There are several ways to access this binary:

- From VisIt's GUI, you can start a CLI instance from the "Launch CLI" entry in the "Options" menu.
- Invoking VisIt from the command line with the `-cli` option starts the CLI and launches a connected *viewer* process:

  ```
  >visit -cli
  ```

  For batch processing, the `-nowin` option launches the viewer in an offscreen mode and you can select a Python script file to run using the `-s` option:
- ```
  >visit -cli -nowin -s
  <script_file.py>
  ```

You can also import the interface into a standalone Python interpreter and use the module to launch and control a new instance of VisIt. Listing 2 provides example code for this use case. The core implementation of the VisIt module is a Python/C extension module, so normal caveats for binary compatibly with your Python interpreter apply.
  ```

The features of the VisIt interface are dependent on the version of VisIt selected, so the import process is broken into two steps. First, a small front end module is imported. This module allows you to select the options used to launch VisIt. Examples include: using `-nowin` mode for the *viewer* process, selecting a specific version of VisIt, `-v 2.5.1`, etc. After these options are set the *Launch()* method creates the appropriate Visit components. During the launch, the interfaces to the available state objects are enumerated and dynamically imported into the *visit* module.

Listing 2: Launch and control VisIt from a standalone Python interpreter.

```
import sys
import os
from os.path import join as pjoin
vpath = "path/to/visit/<ver>/<arch>/"
# or for an OSX bundle version
# "path/to/VisIt.app/Contents/Resources/<ver>/<arch>"
vpath = pjoin(vpath,"lib","site-packages")
sys.path.insert(0,vpath)
import visit
visit.Launch()
# use the interface
visit.OpenDatabase("noise.silo")
visit.AddPlot("Pseudocolor","hardyglobal")
```

Macro Recording

VisIt's GUI provides a *Commands* window that allows you to record GUI actions into short Python snippets. While the client interface supports standard Python introspection methods (`dir()`, `help()`, etc), the *Commands* window provides a powerful learning tool for VisIt's Python API. You can access this window from the "Commands" entry in the "Options" menu. From this window you can record your actions into one of several source scratch pads and convert common actions into macros that can be run using the *Marcos* window.

Custom Python UIs

VisIt provides 100+ database readers, 60+ operators, and over 20 different plots. This toolset makes it a robust application well suited to analyze problem sets from a wide variety of scientific domains. However, in many cases users would like to utilize only a specific subset of VisIt's features and understanding the intricacies of a large general purpose tool can be a daunting task. For example, climate scientists require specialized functionality such as viewing information on Lat/Long grids bundled with computations of zonal averages. Whereas, scientists in the fusion energy science community require visualizations of interactions between magnetic and particle velocity fields within a tokomak simulation. To make it easier to target specific user communities, we extended VisIt with ability to create custom UIs in Python. Since we have an investment in our existing Qt user interface, we choose PySide, an LGPL Python Qt wrapper, as our primary Python UI framework. Leveraging our existing Python Client Interface along with new PySide support allows us to easily and quickly create custom user interfaces that provide specialized analysis routines and directly target the core needs of specific user communities. Using Python allows us to do this in a fraction of the time it would take to do so using our C++ APIs.

VisIt provides two major components to its Python UI interface:

- The ability to embed VisIt's render windows.
- The ability to reuse VisIt's existing set of GUI widgets.

The ability to utilize renderers as Qt widgets allows VisIt's visualization windows to be embedded in custom PySide GUIs and other third party applications. Re-using VisIt's existing generic widget toolset, which provides functionally such as remote filesystem browsing and a visualization pipeline editor, allows custom applications to incorporate advanced features with little difficulty.

One important note, a significant number of changes went into adding Python UI support into VisIt. Traditionally, VisIt uses a component-based architecture where the Python command line interface, the graphical user interface, and the *viewer* exist as separate applications that communicate over sockets. Adding Python UI functionality required these three separate components to work together as single unified application. This required components that once communicated only over sockets to also be able to directly interact with each other. Care is needed when sharing data in this new scenario, we are still refactoring parts of VisIt to better support embedded use cases.

To introduce VisIt's Python UI interface, we start with Listing 3, which provides a simple PySide visualization application that utilizes VisIt under the hood. We then describe two complex applications that use VisIt's Python UI interface with several embedded renderer windows.

Listing 3: Custom application that animates an Isosurface with a sweep across Isovalues.

```
class IsosurfaceWindow(QWidget):
    def __init__(self):
        super(IsosurfaceWindow,self).__init__()
        self.__init_widgets()
        # Setup our example plot.
        OpenDatabase("noise.silo")
        AddPlot("Pseudocolor","hardyglobal")
        AddOperator("Isosurface")
        self.update_isovalue(1.0)
        DrawPlots()
    def __init_widgets(self):
        # Create Qt layouts and widgets.
        vlout = QVBoxLayout(self)
        glout = QGridLayout()
        self.title   = QLabel("Iso Contour Sweep Example")
        self.title.setFont(QFont("Arial", 20, bold=True))
        self.sweep   = QPushButton("Sweep")
        self.lbound  = QLineEdit("1.0")
        self.ubound  = QLineEdit("99.0")
        self.step    = QLineEdit("2.0")
        self.current = QLabel("Current % =")
        f = QFont("Arial",bold=True,italic=True)
        self.current.setFont(f)
        self.rwindow = pyside_support.GetRenderWindow(1)
        # Add title and main render winodw.
        vlout.addWidget(self.title)
        vlout.addWidget(self.rwindow,10)
        glout.addWidget(self.current,1,3)
        # Add sweep controls.
        glout.addWidget(QLabel("Lower %"),2,1)
        glout.addWidget(QLabel("Upper %"),2,2)
        glout.addWidget(QLabel("Step %"),2,3)
        glout.addWidget(self.lbound,3,1)
        glout.addWidget(self.ubound,3,2)
        glout.addWidget(self.step,3,3)
        glout.addWidget(self.sweep,4,3)
        vlout.addLayout(glout,1)
        self.sweep.clicked.connect(self.exe_sweep)
        self.resize(600,600)
    def update_isovalue(self,perc):
        # Change the % value used by
        # the isosurface operator.
        iatts = IsosurfaceAttributes()
        iatts.contourMethod = iatts.Percent
```

```
        iatts.contourPercent = (perc)
        SetOperatorOptions(iatts)
        txt = "Current % = "  + "%0.2f" % perc
        self.current.setText(txt)
    def exe_sweep(self):
        # Sweep % value accoording to
        # the GUI inputs.
        lbv  = float(self.lbound.text())
        ubv  = float(self.ubound.text())
        stpv = float(self.step.text())
        v = lbv
        while v < ubv:
            self.update_isovalue(v)
            v+=stpv

# Create and show our custom window.
main = IsosurfaceWindow()
main.show()
```

In this example, a VisIt render window is embedded in a QWidget to provide a *Pseudocolor* view of an *Isosurface* of the scalar field 'hardyglobal'. We create a set of UI controls that allow the user to select values that control a sweep animation across a range of Isovalues. The *sweep* button initiates the animation. To run this example, the `-pysideviewer` flag is passed to VisIt at startup to select a unified *viewer* and CLI process.

```
> visit -cli -pysideviewer
```

This example was written to work as standalone script to illustrate the use of the PySide API for this paper. For most custom applications, developers are better served by using QtDesigner for UI design, in lieu of hand coding the layout of UI elements. Listing 4 provides a small example showing how to load a QtDesigner UI file using PySide.

Listing 4: Loading a custom UI file created with Qt Designer.

```
from PySide.QtUiTools import *
# example slot
def on_my_button_click():
    print "myButton was clicked"

# Load a UI file created with QtDesigner
loader = QUiLoader()
uifile = QFile("custom_widget.ui")
uifile.open(QFile.ReadOnly)
main = loader.load(uifile)
# Use a string name to locate
# objects from Qt UI file.
button = main.findChild(QPushButton, "myButton")
# After loading the UI, we want to
# connect Qt slots to Python functions
button.clicked.connect(on_my_button_click)
main.show()
```

Advanced Custom Python UI Examples

To provide more context for VisIt's Python UI interface, we now discuss two applications that leverage this new infrastructure: the Global Cloud Resolving Model (GCRM) [GCRM] Viewer and Ultra Visualization - Climate Data Analysis Tools (UV-CDAT) [UVCDAT],

VisIt users in the climate community involved with the global cloud resolving model project (GCRM) mainly required a custom NetCDF reader and a small subset of domain specific plots and operations. Their goal for climate analysis was to quickly visualize models generated from simulations, and perform specialized analysis on these modules. Figure 3 shows two customized skins for the GCRM community developed in QtDesigner and loaded using

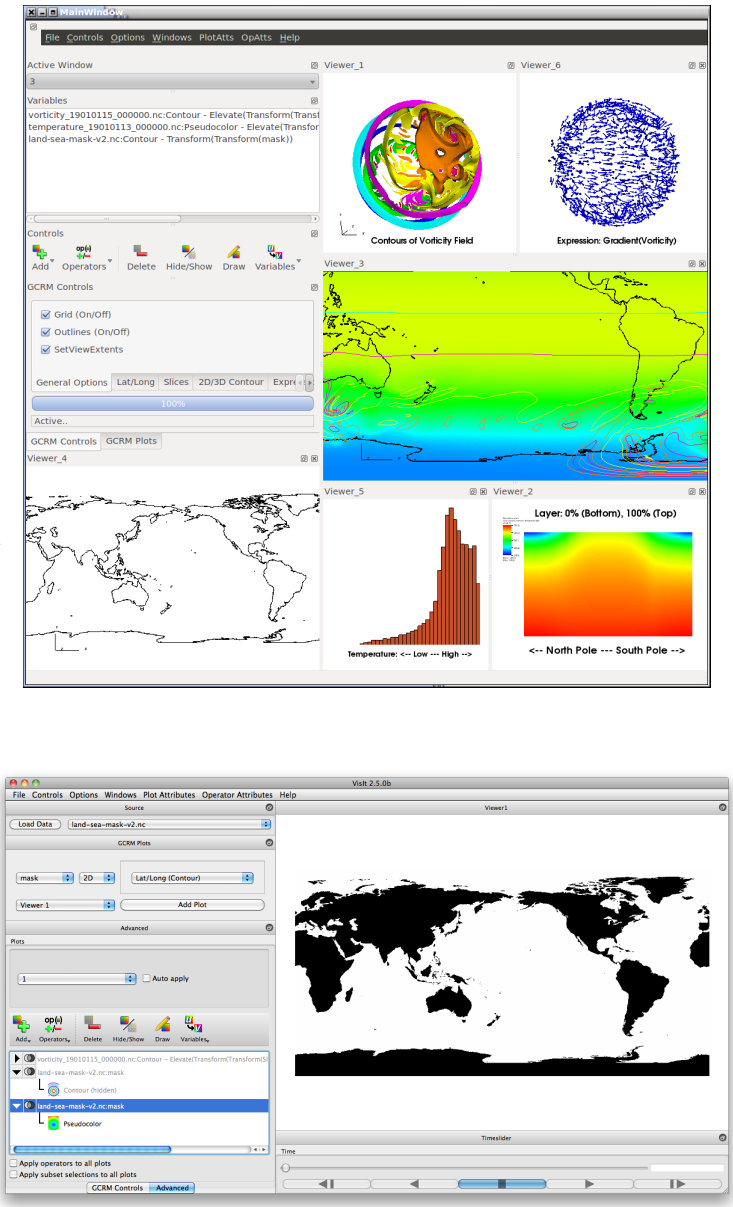


Fig. 3: *Climate Skins for the Global Cloud Resolving Model Viewer.*

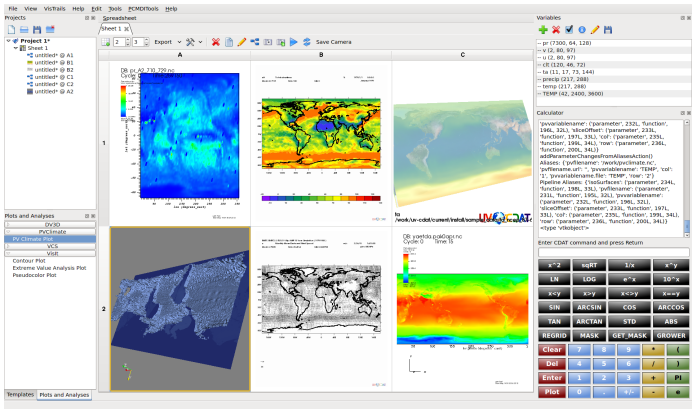


Fig. 4: *Example showing integration of VisIt's components in UV-CDAT.*

PySide from VisIt's Python UI client. The customized skins embed VisIt rendering windows and reuse several of VisIt's GUI widgets. We also wrote several new analysis routines in Python for custom visualization and analysis tasks targeted for the climate community. This included providing Lat/Long grids with continental outlines and computing zonal means. Zonal mean computation was achieved by computing averages across the latitudes for each layer of elevation for a given slice in the direction of the longitude.

UV-CDAT is a multi-institutional project geared towards addressing the visualization needs of climate scientists around the world. Unlike the GCRM project which was targeted towards one specific group and file format, for UV-CDAT all of VisIt's functionality needs to be exposed and embedded alongside several other visualization applications. The goal of UV-CDAT is to bring together all the visualization and analysis routines provided within several major visualization frameworks inside one application. This marks one of the first instances where several separate fully-featured visualization packages, including VisIt, ParaView, DV3D, and VisTrails all function as part of one unified application. Figure 4 shows an example of using VisIt plots, along with plots from several other packages, within UV-CDAT. The core UV-CDAT application utilizes PyQt [PyQt] as its central interface and Python as the intermediate bridge between the visualization applications. The infrastructure changes made to VisIt to support custom Python UIs via PySide also allowed us to easily interface with PyQt. Apart from creating PyQt wrappers for the project, we also made significant investments in working out how to effectively share resources created within Python using NumPy & VTK Python data objects.

Python Filter Runtime

The Python Client Interface allows users to assemble visualization pipelines using VisIt's existing building blocks. While VisIt provides a wide range of filters, there are of course applications that require special purpose algorithms or need direct access to low-level mesh data structures. VisIt's Database, Operator, and Plot primitives are extendable via a C++ plugin infrastructure. This infrastructure allows new instances of these building blocks to be developed against an installed version of VisIt, without access to VisIt's full source tree. Whereas, creating new Expression and Query primitives in C++ currently requires VisIt's full source tree. To provide more flexibility for custom work flows and special purpose algorithms, we extended our data flow network pipelines with a Python Filter Runtime. This extension provides two important benefits:

- Enables runtime prototyping/modification of filters.
- Reduces development time for special purpose/one-off filters.

To implement this runtime, each MPI process in VisIt's *compute engine* embeds a Python interpreter. The interpreter coordinates with the rest of the pipeline using Python/C wrappers for existing pipeline control data structures. These data structures also allow requests for pipeline optimizations, for example a request to generate ghost zones. VisIt's pipelines use VTK mesh data structures internally, allowing us to pass VTK objects zero-copy between C++ and the Python interpreter using Kitware's existing VTK Python wrapper module. Python instances of VTK data arrays can also be wrapped zero-copy into *ndarrays*, opening up access to the wide range of algorithms available in NumPy and SciPy.

To create a custom filter, the user writes a Python script that implements a class that extends a base filter class for the desired VisIt building block. The base filter classes mirror VisIt's existing C++ class hierarchy. The exact execution pattern varies according the to selected building block, however they loosely adhere to the following basic data-parallel execution pattern:

- Submit requests for pipeline constraints or optimizations.
- Initialize the filter before parallel execution.
- Process mesh data sets in parallel on all MPI tasks.
- Run a post-execute method for cleanup and/or summarization.

To support the implementation of distributed-memory algorithms, the Python Filter Runtime provides a simple Python MPI wrapper module, named *mpicom*. This module includes support for collective and point-to-point messages. The interface provided by *mpicom* is quite simple, and is not as optimized or extensive as other Python MPI interface modules, as such *mpi4py* [Mpi4Py]. We would like to eventually adopt *mpi4py* for communication, either directly or as a lower-level interface below the existing *mpicom* API.

VisIt's Expression and Query filters are the first constructs exposed by the Python Filter Runtime. These primitives were selected because they are not currently extensible via our C++ plugin infrastructure. Python Expressions and Queries can be invoked from VisIt's GUI or the Python Client Interface. To introduce these filters, this paper will outline a simple Python Query example and discuss how a Python Expression was used to research a new OpenCL Expression Framework.

Listing 5: Python Query filter that calculates the average of a cell centered scalar field.

```
class CellAverageQuery(SimplePythonQuery):
    def __init__(self):
        # basic initialization
        super(CellAverageQuery,self).__init__()
        self.name = "Cell Average Query"
        self.description = "Calculating scalar average."
    def pre_execute(self):
        # called just prior to main execution
        self.local_ncells = 0
        self.local_sum    = 0.0
    def execute_chunk(self,ds_in,domain_id):
        # called per mesh chunk assigned to
        # the local MPI task.
        ncells = ds_in.GetNumberOfCells()
        if ncells == 0:
            return
        vname  = self.input_var_names[0]
        varray = ds_in.GetCellData().GetArray(vname)
        self.local_ncells += ncells
        for i in range(ncells):
            self.local_sum += varray.GetTuple1(i)
    def post_execute(self):
        # called after all mesh chunks on all
        # processors have been processed.
        tot_ncells = mpicom.sum(self.local_ncells)
        tot_sum    = mpicom.sum(self.local_sum)
        avg = tot_sum  / float(tot_ncells)
        if mpicom.rank() == 0:
            vname = self.input_var_names[0]
            msg = "Average value of %s = %s"
            msg = msg % (vname,str(avg))
            self.set_result_text(msg)
            self.set_result_value(avg)

# Tell the Python Filter Runtime which class to use
# as the Query filter.
py_filter = CellAverageQuery
```

Listing 6: Python Client Interface code to invoke the Cell Average Python Query on a example data set.

```
# Open an example data set.
OpenDatabase("multi_rect3d.silo")
# Create a plot to query
AddPlot("Pseudocolor","d")
DrawPlots()
# Execute the Python query
PythonQuery(file="listing_5_cell_average.vpq",
            vars=["default"])
```

Listing 5 provides an example Python Query script, and Listing 6 provides example host code that can be used to invoke the Python Query from VisIt's Python Client Interface. In this example, the *pre_execute* method initializes a cell counter and a variable to hold the sum of all scalar values provided by the host MPI task. After initialization, the *execute_chunk* method is called for each mesh chunk assigned to the host MPI task. *execute_chunk* examines these meshes via VTK's Python wrapper interface, obtaining the number of cells and the values from a cell centered scalar field. After all chunks have been processed by the *execute_chunk* method on all MPI tasks, the *post_execute* method is called. This method uses MPI reductions to obtain the aggregate number of cells and total scalar value sum. It then calculates the average value and sets an output message and result value on the root MPI process.

Using a Python Expression to host a new OpenCL Expression Framework.

The HPC compute landscape is quickly evolving towards accelerators and many-core CPUs. The complexity of porting existing codes to the new programming models supporting these architectures is a looming concern. We have an active research effort exploring OpenCL [OpenCL] for visualization and analysis applications on GPU clusters.

One nice feature of OpenCL is the that it provides runtime kernel compilation. This opens up the possibility of assembling custom kernels that dynamically encapsulate multiple steps of a visualization pipeline into a single GPU kernel. A subset of our OpenCL research effort is focused on exploring this concept, with the goal of creating a framework that uses OpenCL as a backend for user defined expressions. This research is joint work with Maysam Moussalem and Paul Navrátil at the Texas Advanced Computing Center, and Ming Jiang at Lawrence Livermore National Laboratory.

For productivity reasons we chose Python to prototype this framework. We dropped this framework into VisIt's existing data parallel infrastructure using a Python Expression. This allowed us to test the viability of our framework on large data sets in a distributed-memory parallel setting. Rapid development and testing of this framework leveraged the following Python modules:

- *PLY* [PLY] was used to parse our expression language grammar. PLY provides an easy to use Python lex/yacc implementation that allowed us to implement a front-end parser and integrate it with the Python modules used to generate and execute OpenCL kernels.
- *PyOpenCL* [PyOpenCL] was used to interface with OpenCL and launch GPU kernels. PyOpenCL provides a wonderful interface to OpenCL that saved us an untold amount of time over the OpenCL C-API. PyOpenCL also uses *ndarrays* for data transfer between the CPU and GPU, and this was a great fit because we can easily access our data arrays as *ndarrays* using the VTK Python wrapper module.

We are in the process of conducting performance studies and writing a paper with the full details of the framework. For this paper we provide a high-level execution overview and a few performance highlights:

Execution Overview:

An input expression, defining a new mesh field, is parsed by a PLY front-end and translated into a data flow network specification. The data flow network specification is passed to a Python data flow module that dynamically generates a single OpenCL kernel for the operation. By dispatching a single kernel that encapsulates several pipeline steps to the GPU, the framework mitigates CPU-GPU I/O bottlenecks and boosts performance over both existing CPU pipelines and a naive dispatch of several small GPU kernels.

Performance Highlights:

- Demonstrated speed up of up to ~20x vs an equivalent VisIt CPU expression, including transfer of data arrays to and from the GPU.
- Demonstrated use in a distributed-memory parallel setting, processing a 24 billion zone rectilinear mesh using 256 GPUs on 128 nodes of LLNL's Edge cluster.

Python, PLY, PyOpenCL, and VisIt's Python Expression capability allowed us to create and test this framework with a much faster turn around time than would have been possible using C/C++ APIs. Also, since the bulk of the processing was executed on GPUs, we were able to demonstrate impressive speedups.

Conclusion

In this paper we have presented an overview of the various roles that Python plays in VisIt's software infrastructure and a few examples of visualization and analysis use cases enabled by Python in VisIt. Python has long been an asset to VisIt as the foundation of VisIt's scripting language. We have recently extended our infrastructure to enable custom application development and low-level mesh processing algorithms in Python.

For future work, we are refactoring VisIt's component infrastructure to better support unified process Python UI clients. We also hope to provide more example scripts to help developers bootstrap custom Python applications that embed VisIt. We plan to extend our Python Filter Runtime to allow users to write new Databases and Operators in Python. We would also like to provide new base classes for Python Queries and Expressions that encapsulate the VTK to *ndarray* wrapping process, allowing users to write streamlined scripts using NumPy.

For more detailed info on VisIt and its Python interfaces, we recommend: the VisIt Website [VisItWeb], the VisIt Users' Wiki [VisItWiki], VisIt's user and developer mailing lists, and the VisIt Python Client reference manual [VisItPyRef].

Acknowledgments

This work performed under the auspices of the U.S. DOE by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-564292.

REFERENCES

[VisIt05] Childs, H. et al, 2005. A Contract Based System For Large Data Visualization. *VIS '05: Proceedings of the conference on Visualization '05*

[ParaView05] Ahrens, J. et al, 2005. Visualization in the ParaView Framework. *The Visualization Handbook*, 162-170

[EnSight09] EnSight User Manual. Computational Engineering International, Inc. Dec 2009.

[OpenCL] Kronos Group, OpenCL parallel programming framework. http://www.khronos.org/opencl/

[PLY] Beazley, D., Python Lex and Yacc. http://www.dabeaz.com/ply/

[NumPy] Oliphant, T., NumPy Python Module. http://numpy.scipy.org

[SciPy] Scientific Tools for Python. http://www.scipy.org.

[PyOpenCL] Klöckner, A., Python OpenCL Module. http://mathema.tician.de/software/pyopencl

[PySide] PySide Python Bindings for Qt. http://www.pyside.org/

[Mpi4Py] Dalcin, L., mpi4py: MPI for Python. http://mpi4py.googlecode.com/

[VTK96] Schroeder, W. et al, 1996. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. *VIS '96: Proceedings of the 7th conference on Visualization '96*

[VisItPyRef] Whitlock, B. et al. VisIt Python Reference Manual. http://portal.nersc.gov/svn/visit/trunk/releases/2.3.0/VisItPythonManual.pdf

[PyQt] PyQt Python Bindings for Qt. http://www.riverbankcomputing.co.uk/software/pyqt/

[UVCDAT] Ultra Visualization - Climate Data Analysis Tools. http://uv-cdat.llnl.gov

[GCRM] Global Cloud Resolving Model. http://kiwi.atmos.colostate.edu/gcrm/

[VisItWiki] VisIt Users' Wiki. http://www.visitusers.org/

[VisItWeb] VisIt Website. https://wci.llnl.gov/codes/visit/

PythonTeX: Fast Access to Python from within LaTeX

Geoffrey M. Poore[‡*]

Abstract—PythonTeX is a new LaTeX package that provides access to the full power of Python from within LaTeX documents. It allows Python code entered within a LaTeX document to be executed, and provides access to the output. PythonTeX also provides syntax highlighting for any language supported by the Pygments highlighting engine.

PythonTeX is fast and user-friendly. Python code is separated into user-defined sessions. Each session is only executed when its code is modified. When code is executed, sessions run in parallel. The contents of stdout and stderr are synchronized with the LaTeX document, so that printed content is easily accessible and error messages have meaningful line numbering.

PythonTeX simplifies scientific document creation with LaTeX. Plots can be created with matplotlib and then customized in place. Calculations can be performed and automatically typeset with NumPy. SymPy can be used to automatically create mathematical tables and step-by-step mathematical derivations.

Index Terms—LaTeX, document preparation, document automation, matplotlib, NumPy, SymPy, Pygments

Introduction

Scientific documents and presentations are often created with the LaTeX document preparation system. Though some LaTeX tools exist for creating figures and performing calculations, external scientific software is typically required for these tasks. This can result in an inefficient workflow. Every time a calculation requires modification or a figure needs tweaking, the user must switch between LaTeX and the scientific software. The user must locate the code that created the calculation or figure, modify it, and execute it before returning to LaTeX.

One way to streamline this process is to include non-LaTeX code within LaTeX documents, with a means to execute this code and access the output. That approach has connections to Knuth's concept of literate programming, in which code and its documentation are combined in a single document [Knuth]. The noweb literate programming tool extended Knuth's work to additional document formats and arbitrary programming languages [Ramsey]. Sweave subsequently built on noweb by allowing the output of individual chunks of R code to be accessed within the document [Leisch]. This made possible dynamic reports that are reproducible since they contain the code that generated their results. As such, Sweave and similar tools represent an additional, complementary approach to reproducibility compared to makefile-based approaches [Schwab].

* *Corresponding author: gpoore@uu.edu*
‡ *Union University*

Several methods of including executable code in LaTeX documents ultimately function as preprocessors or templating systems. A document might contain a mix of LaTeX and code, and the preprocessor replaces the code with its output. The original document would not be valid LaTeX; only the preprocessed document would be. Sweave, knitr [Xie], the Python-based Pweave [Pastell], and template libraries such as Mako [MK] function in this manner. More recently, the IPython notebook has provided an interactive browser-based interface in which text, code, and code output may be interspersed [IPY]. Since the notebook can be exported as LaTeX, it functions similarly to the preprocessor-style approach.

The preprocessor/templating-style approach has a significant advantage. All of the examples mentioned above are compatible with multiple document formats, not just LaTeX. This is particularly true in the case of templating libraries. One significant drawback is that the line numbers of the preprocessed document, which LaTeX receives, do not correspond to those of the original document. This makes it difficult to debug LaTeX errors, particularly in longer documents. It also breaks standard LaTeX tools such as forward and inverse search between a document and its PDF (or other) output; only Sweave and knitr have systems to work around this. An additional issue is that it is difficult for LaTeX code to interact with code in other languages, when the code in other languages has already been executed and removed before LaTeX runs.

In an alternate approach to including executable code in LaTeX documents, the original document is valid LaTeX, containing code wrapped in special commands and environments. The code is extracted by LaTeX itself during compilation, then executed and replaced by its output. Such approaches with Python go back to at least 2007, with Martin R. Ehmsen's python.sty style file [Ehmsen]. Since 2008, SageTeX has provided access to the Sage mathematics system from within LaTeX [Drake]. Because Sage is largely based on Python, it also provides Python access. SympyTeX (2009) is based on SageTeX [Molteno]. Though SympyTeX is primarily intended for accessing the SymPy library for symbolic mathematics [SymPy], it provides general access to Python. Since these packages begin with a valid LaTeX document, they automatically work with standard LaTeX editing tools and also allow LaTeX code to interact with Python.

Python.sty, SageTeX, and SympyTeX illustrate the potential of a close Python-LaTeX integration. At the same time, they leave much of the possible power of the Python-LaTeX combination untapped. Python.sty requires that all Python code be executed every time the document is compiled. SageTeX and SympyTeX separate code execution from document compilation, but because all code is executed in a single session, everything must be executed whenever anything changes. None of these packages provides

comprehensive syntax highlighting. SageTeX and SympyTeX do not provide access to stdout or stderr. They do synchronize error messages with the document, but synchronization is performed by executing a `try/except` statement on every line of the user's code. This reduces performance and fails in the case of syntax errors.

PythonTeX is a new LaTeX package that provides access to Python from within LaTeX documents. It emphasizes performance and usability.

- Python-generated content is always saved, so that the LaTeX document can be compiled without running Python.
- Python code is divided into user-defined sessions. Each session is only executed when it is modified. When code is executed, sessions run in parallel.
- Both stdout and stderr are easily accessible.
- All Python error messages are synchronized with the LaTeX document, so that error line numbers correctly correspond to the document.
- Code may be typeset with highlighting provided by Pygments [Pyg]—this includes any language supported by Pygments, not just Python. Unicode is supported.
- Native Python code is fully supported, including imports from `__future__`. No changes to Python code are required to make it compatible with PythonTeX.

While PythonTeX lacks the rapid interactivity of the IPython notebook, as a LaTeX package it offers much tighter Python-LaTeX integration. It also provides greater control over what is displayed (code, stdout, or stderr) and allows executable code to be included inline within normal text.

This paper presents the main features of PythonTeX and considers several examples. It also briefly discusses the internal workings of the package.

PythonTeX environments and commands

PythonTeX provides four LaTeX environments and four LaTeX commands for accessing Python. These environments and commands save code to an external file and then bring back the output once the code has been processed by PythonTeX.

The **code** environment simply executes the code it contains. By default, any printed content is brought in immediately after the end of the environment and interpreted as LaTeX code. For example, the LaTeX code

```
\begin{pycode}
myvar = 123
print('Greetings from Python!')
\end{pycode}
```

creates a variable `myvar` and prints a string, and the printed content is automatically included in the document:

Greetings from Python!

The **block** environment executes its contents and also typesets it. By default, the typeset code is highlighted using Pygments. Reusing the Python code from the previous example,

```
\begin{pyblock}
myvar = 123
print('Greetings from Python!')
\end{pyblock}
```

creates

```
myvar = 123
print('Greetings from Python!')
```

The printed content is not automatically included. Typically, the user wouldn't want the printed content immediately after the typeset code—explanation of the code, or just some space, might be desirable before showing the output. Two equivalent commands are provided for including the printed content generated by a block environment: `\printpythontex` and `\stdoutpythontex`. These bring in any printed content created by the most recent PythonTeX environment and interpret it as LaTeX code. Both commands also take an optional argument to bring in content as verbatim text. For example, `\printpythontex[v]` brings in the content in a verbatim form suitable for inline use, while `\printpythontex[verb]` brings in the content as a verbatim block.

All code entered within code and block environments is executed within the same Python session (unless the user specifies otherwise, as discussed below). This means that there is continuity among environments. For example, since `myvar` has already been created, it can now be modified:

```
\begin{pycode}
myvar += 4
print('myvar = ' + str(myvar))
\end{pycode}
```

This produces

myvar = 127

The **verb** environment typesets its contents, without executing it. This is convenient for simply typesetting Python code. Since the verb environment has a parallel construction to the code and block environments, it can also be useful for temporarily disabling the execution of some code. Thus

```
\begin{pyverb}
myvar = 123
print('Greetings from Python!')
\end{pyverb}
```

results in the typeset content

```
myvar = 123
print('Greetings from Python!')
```

without any code actually being executed.

The final environment is different. The **console** environment emulates a Python interactive session, using Python's `code` module. Each line within the environment is treated as input to an interactive interpreter. The LaTeX code

```
\begin{pyconsole}
myvar = 123
myvar
print('Greetings from Python!')
\end{pyconsole}
```

creates

```
>>> myvar = 123
>>> myvar
123
>>> print('Greetings from Python!')
Greetings from Python!
```

PythonTeX provides options for showing and customizing a banner at the beginning of console environments. The content of all console environments is executed within a single Python session, providing continuity, unless the user specifies otherwise.

While the PythonTeX environments are useful for executing and typesetting large blocks of code, the PythonTeX commands are intended for inline use. Command names are based on abbreviations of environment names. The **code** command simply executes its contents. For example, `\pyc{myvar = 123}`. Again, any printed content is automatically included by default. The **block** command typesets and executes the code, but does not automatically include printed content (`\printpythontex` is required). Thus, `\pyb{myvar = 123}` would typeset

```
myvar = 123
```

in a form suitable for inline use, in addition to executing the code. The **verb** command only typesets its contents. The command `\pyv{myvar = 123}` would produce

```
myvar=123
```

without executing anything. If Pygments highlighting for inline code snippets is not desired, it may be turned off.

The final inline command, `\py`, is different. It provides a simple way to typeset variable values or to evaluate short pieces of code and typeset the result. For example, `\py{myvar}` accesses the previously created variable `myvar` and brings in a string representation: 123. Similarly, `\py{2**8 + 1}` converts its argument to a string and returns 257.

It might seem that the effect of `\py` could be achieved using `\pyc` combined with `print`. But `\py` has significant advantages. First, it requires only a single external file per document for bringing in content, while `print` requires an external file for each environment and command in which it is used. This is discussed in greater detail in the discussion of PythonTeX's internals. Second, the way in which `\py` converts its argument to a valid LaTeX string can be specified by the user. This can save typing when several conversions or formatting operations are needed. The examples below using SymPy illustrate this approach.

All of the examples of inline commands shown above use opening and closing curly brackets to delimit the code. This system breaks down if the code itself contains an unmatched curly bracket. Thus, all inline commands also accept arbitrary matched characters as delimiters. This is similar to the behavior of LaTeX's `\verb` macro. For example, `\pyc!myvar = 123!` and `\pyc#myvar = 123#` are valid. No such consideration is required for environments, since they are delimited by `\begin` and `\end` commands.

Options: Sessions and Fancy Verbatims

PythonTeX commands and environments take optional arguments. These determine the session in which the code is executed and provide additional formatting options.

By default, all code and block content is executed within a single Python session, and all console content is executed within a separate session. In many cases, such behavior is desired because of the continuity it provides. At times, however, it may be useful to isolate some independent code in its own session. A long calculation could be placed in its own session, so that it only runs when its code is modified, independently of other code.

PythonTeX provides such functionality through user-defined sessions. All commands and environments take a session name as an optional argument. For example, `\pyc[slowsession]{myvar = 123}` and

```
\begin{pycode}[slowsession]
myvar = 123
print('Greetings from Python!')
\end{pycode}
```

Each session is only executed when its code has changed, and sessions run in parallel (via Python's `multiprocessing` package), so careful use of sessions can significantly increase performance.

All PythonTeX environments also accept a second optional argument. This consists of settings for the LaTeX `fancyvrb` (Fancy Verbatims) package [FV], which PythonTeX uses for typesetting code. These settings allow customization of the code's appearance. For example, a block of code may be surrounded by a colored frame, with a title. Or line numbers may be included.

Plotting with matplotlib

The PythonTeX commands and environments can greatly simplify the creation of scientific documents and presentations. One example is the inclusion of plots created with matplotlib [MPL].

All of the commands and environments discussed above begin with the prefix `py`. PythonTeX provides a parallel set of commands and environments that begin with the prefix `pylab`. These behave identically to their `py` counterparts, except that matplotlib's `pylab` module is automatically imported via `from pylab import *`. The `pylab` commands and environments can make it easier to keep track of code dependencies and separate content that would otherwise require explicit sessions; the default `pylab` session is separate from the default `py` session.

Combining PythonTeX with matplotlib significantly simplifies plotting. The commands for creating a plot may be included directly within the LaTeX source, and the plot may be edited in place to get the appearance just right. Matplotlib's LaTeX option may be used to keep fonts consistent between the plot and the document. The code below illustrates this approach. Notice that the plot is created in its own session, to increase performance.

```
\begin{pylabcode}[plotsession]
rc('text', usetex=True)
rc('font', **{'family':'serif', 'serif':['Times']})
rc('font', size=10.0)
rc('legend', fontsize=10.0)
x = linspace(0, 3*pi)
figure(figsize=(3.25,2))
plot(x, sin(x), label='$\sin(x)$')
plot(x, sin(x)**2, label='$\sin^2(x)$',
     linestyle='dashed')
xlabel(r'$x$-axis')
ylabel(r'$y$-axis')
xticks(arange(0, 4*pi, pi), ('$0$',
       '$\pi$', '$2\pi$', '$3\pi$'))
axis([0, 3*pi, -1, 1])
legend(loc='lower right')
savefig('myplot.pdf', bbox_inches='tight')
\end{pylabcode}
```

The plot may be brought in and positioned using the standard LaTeX commands:

```
\begin{figure}
\centering
\includegraphics{myplot}
\caption{\label{fig:matplotlib} A plot
created with PythonTeX.}
\end{figure}
```

The end result is shown in Figure 1.

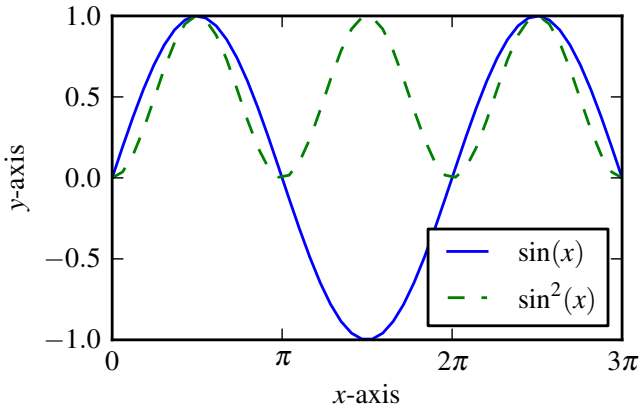


Fig. 1: *A matplotlib plot created with PythonTeX.*

Solving equations with NumPy

PythonTeX didn't require any special modifications to the Python code in the previous example with matplotlib. The code that created the plot was the same as it would have been had an external script been used to generate the plot. In some situations, however, it can be beneficial to acknowledge the LaTeX context of the Python code. This may be illustrated by solving an equation with NumPy [NP].

Perhaps the most obvious way to solve an equation using PythonTeX is to separate the Python solving from the LaTeX typesetting. Consider finding the roots of a polynomial using NumPy.

```
\begin{pylabcode}
coeff = [4, 2, -4]
r = roots(coeff)
\end{pylabcode}

The roots of $4x^2 + 2x - 4 = 0$ are
$\pylab{r[0]}$ and $\pylab{r[1]}$.
```

This yields

The roots of $4x^2+2x-4=0$ are -1.2807764064 and 0.780776406404.

Such an approach works, but the code must be modified significantly whenever the polynomial changes. A more sophisticated approach automatically generates the LaTeX code and perhaps rounds the roots as well, for an arbitrary polynomial.

```
\begin{pylabcode}
coeff = [4, 2, -4]
# Build a string containing equation
eq = ''
for n, c in enumerate(coeff):
    if n == 0 or str(c).startswith('-'):
        eq += str(c)
    else:
        eq += '+' + str(c)
    if len(coeff) - n - 1 == 1:
        eq += 'x'
    elif len(coeff) - n - 1 > 1:
        eq += 'x^' + str(len(coeff) - n - 1)
eq += '=0'
# Get roots and format for LaTeX
r = ['{0:+.3f}'.format(root)
     for root in roots(coeff)]
latex_roots = ','.join(r)
\end{pylabcode}

The roots of $\pylab{eq}$ are
$[\pylab{latex_roots}]$.
```

This yields

The roots of $4x^2+2x-4=0$ are $[-1.281,+0.781]$.

The automated generation of LaTeX code on the Python side begins to demonstrate the full power of PythonTeX.

Solving equations with SymPy

Several examples with SymPy further illustrate the potential of Python-generated LaTeX code [SymPy].

To simplify SymPy use, PythonTeX provides a set of commands and environments that begin with the prefix `sympy`. These are identical to their `py` counterparts, except that SymPy is automatically imported via `from sympy import *`.

SymPy is ideal for PythonTeX use, because its `LatexPrinter` class and the associated `latex()` function provide LaTeX representations of objects. For example, returning to solving the same polynomial,

```
\begin{sympycode}
x = symbols('x')
myeq = Eq(4*x**2 + 2*x - 4)
print('The roots of the equation ')
print(latex(myeq, mode='inline'))
print(' are ')
print(latex(solve(myeq), mode='inline'))
\end{sympycode}
```

creates

The roots of the equation $4x^2+2x-4=0$ are $\left[-\frac{1}{4}\sqrt{17}-\frac{1}{4},\quad -\frac{1}{4}+\frac{1}{4}\sqrt{17}\right]$

Notice that the printed content appears as a single uninterrupted line, even though it was produced by multiple prints. This is because the printed content is interpreted as LaTeX code, and in LaTeX an empty line is required to end a paragraph.

The `\sympy` command provides an alternative to printing. While the `\py` and `\pylab` commands attempt to convert their arguments directly to a string, the `\sympy` command converts its argument using SymPy's `LatexPrinter` class. Thus, the output from the last example could also have been produced using

```
\begin{sympycode}
x = symbols('x')
myeq = Eq(4*x**2 + 2*x - 4)
\end{sympycode}

The roots of the equation $\sympy{myeq}$
are $\sympy{solve(myeq)}$.
```

The `\sympy` command uses a special interface to the `LatexPrinter` class, to allow for context-dependent `LatexPrinter` settings. PythonTeX includes a utilities class, and an instance of this class called `pytex` is created within each PythonTeX session. The `formatter()` method of this class is responsible for converting objects into strings for `\py`, `\pylab`, and `\sympy`. In the case of SymPy, `pytex.formatter()` provides an interface to `LatexPrinter`, with provision for context-dependent customization. In LaTeX, there are four possible math styles: displaystyle (regular equations), textstyle (inline), scriptstyle (superscripts and subscripts), and scriptscriptstyle (superscripts and subscripts, of superscripts and subscripts). Separate

`LatexPrinter` settings may be specified for each of these styles individually, using a command of the form

```
pytex.set_sympy_latex(style, **kwargs)
```

For example, by default `\sympy` is set to create normal-sized matrices in displaystyle and small matrices elsewhere. Thus, the following code

```
\begin{sympycode}
m = Matrix([[1,0], [0,1]])
\end{sympycode}

The matrix in inline is small:  $\sympy{m}$

The matrix in an equation is of normal size:
\[ \sympy{m} \]
```

produces

The matrix in inline is small: $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$
The matrix in an equation is of normal size:

$$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

As another example, consider customizing the appearance of inverse trigonometric functions based on their context.

```
\begin{sympycode}
x = symbols('x')
sineq = Eq(asin(x/2)-pi/3)
pytex.set_sympy_latex('display',
                      inv_trig_style='power')
pytex.set_sympy_latex('text',
                      inv_trig_style='full')
\end{sympycode}

Inline:  $\sympy{sineq}$

Equation:  \[ \sympy{sineq} \]
```

This creates

Inline: $\arcsin\left(\frac{1}{2}x\right) - \frac{1}{3}\pi = 0$
Equation:

$$\sin^{-1}\left(\frac{1}{2}x\right) - \frac{1}{3}\pi = 0$$

Notice that in both examples above, the `\sympy` command is simply used—no information about context must be passed to Python. On the Python side, the context-dependent `LatexPrinter` settings are used to determine whether the LaTeX representation of some object is context-dependent. If not, Python creates a single LaTeX representation of the object and returns that. If the LaTeX representation is context-dependent, then Python returns multiple LaTeX representations, wrapped in LaTeX's `\mathchoice` macro. The `\mathchoice` macro takes four arguments, one for each of the four LaTeX math styles display, text, script, and scriptscript. The correct argument is typeset by LaTeX based on the current math style.

Step-by-step derivations with SymPy

With SymPy's LaTeX functionality, it is simple to automate tasks that could otherwise be tedious. Instead of manually typing step-by-step mathematical solutions, or copying them from an external program, the user can generate them automatically from within LaTeX.

```
\begin{sympycode}
x, y = symbols('x, y')
f = x + sin(y)
step1 = Integral(f, x, y)
step2 = Integral(Integral(f, x).doit(), y)
step3 = step2.doit()
\end{sympycode}

\begin{align*}
\sympy{step1} &= \sympy{step2} \\
              &= \sympy{step3}
\end{align*}
```

This produces

$$\begin{aligned}\iint x+\sin(y)\,dx\,dy &= \int \frac{1}{2}x^2 + x\sin(y)\,dy\\ &= \frac{1}{2}x^2 y - x\cos(y)\end{aligned}$$

Automated mathematical tables with SymPy

The creation of mathematical tables is another traditionally tedious task that may be automated with PythonTeX and SymPy. Consider the following code, which automatically creates a small integral and derivative table.

```
\begin{sympycode}
x = symbols('x')
funcs = ['sin(x)', 'cos(x)', 'sinh(x)', 'cosh(x)']
ops = ['Integral', 'Derivative']
print('\\begin{align*}')
for func in funcs:
    for op in ops:
        obj = eval(op + '(' + func + ', x)')
        left = latex(obj)
        right = latex(obj.doit())
        if op != ops[-1]:
            print(left + '&=' + right + '&')
        else:
            print(left + '&=' + right + r'\\')
print('\\end{align*}')
\end{sympycode}
```

$$\begin{aligned}\int \sin(x)\,dx &= -\cos(x) & \frac{\partial}{\partial x}\sin(x) &= \cos(x)\\ \int \cos(x)\,dx &= \sin(x) & \frac{\partial}{\partial x}\cos(x) &= -\sin(x)\\ \int \sinh(x)\,dx &= \cosh(x) & \frac{\partial}{\partial x}\sinh(x) &= \cosh(x)\\ \int \cosh(x)\,dx &= \sinh(x) & \frac{\partial}{\partial x}\cosh(x) &= \sinh(x)\end{aligned}$$

This code could easily be modified to generate a page or more of integrals and derivatives by simply adding additional function names to the `funcs` list.

Debugging and access to stderr

PythonTeX commands and environments save the Python code they contain to an external file, where it is processed by PythonTeX. When the Python code is executed, errors may occur. The line numbers for these errors do not correspond to the document line numbers, because only the Python code contained in the document is executed; the LaTeX code is not present. Furthermore, the error line numbers do not correspond to the line numbers that would be obtained by only counting the Python code in

the document, because PythonTeX must execute some boilerplate management code in addition to the user's code. This presents a challenge for debugging.

PythonTeX addresses this issue by tracking the original LaTeX document line number for each piece of code. All error messages are parsed, and Python code line numbers are converted to LaTeX document line numbers. The raw stderr from the Python code is interspersed with PythonTeX messages giving the document line numbers. For example, consider the following code, with a syntax error in the last line:

```
\begin{pyblock}[errorsession]
x = 1
y = 2
z = x + y +
\end{pyblock}
```

The error occurred on line 3 of the Python code, but this might be line 104 of the actual document and line 47 of the combined code and boilerplate. In this case, running the PythonTeX script that processes Python code would produce the following message, where `<temp file name>` would be the name of a temporary file that was executed:

```
* PythonTeX code error on line 104:
    File "<temp file name>", line 47
      z = x + y +
                ^
  SyntaxError: invalid syntax
```

Thus, finding code error locations is as simple as it would be if the code were written in separate files and executed individually. PythonTeX is the first Python-LaTeX solution to provide such comprehensive error line synchronization.

In general, errors are something to avoid. In the context of writing about code, however, they may be created intentionally for instructional purposes. Thus, PythonTeX also provides access to error messages in a form suitable for typesetting. If the PythonTeX package option `stderr` is enabled, any error message created by the most recent PythonTeX command or environment is available via `\stderrpythontex`. By default, stderr content is brought in as LaTeX verbatim content; this preserves formatting and prevents issues caused by stderr content not being valid LaTeX.

Python code and the error it produces may be typeset next to each other. Reusing the previous example,

```
\begin{pyblock}[errorsession]
x = 1
y = 2
z = x + y +
\end{pyblock}
```

creates the following typeset code:

```
x = 1
y = 2
z = x + y +
```

The stderr may be brought in via `\stderrpythontex`:

```
  File "<file>", line 3
    z = x + y +
              ^
SyntaxError: invalid syntax
```

Two things are noteworthy about the form of the stderr. First, in the case shown, the file name is given as `"<file>"`. PythonTeX provides a package option `stderrfilename` for controlling this name. The actual name of the temporary file that was executed may be shown, or simply a name based on the session (`"errorsession.py"` in this case), or the more generic `"<file>"` or `"<script>"`. Second, the line number shown corresponds to the code that was actually entered in the document, not to the document line number or to the line number of the code that was actually executed (which would have included PythonTeX boilerplate). To accomplish this, PythonTeX parses the stderr and corrects the line number, so that the typeset code and the typeset stderr are in sync.

General code highlighting with Pygments

The primary purpose of PythonTeX is to execute Python code included in LaTeX documents and provide access to the output. Once support for Pygments highlighting of Python code was added [Pyg], however, it was simple to add support for general code highlighting.

PythonTeX provides a `\pygment` command for typesetting inline code snippets, a `pygments` environment for typesetting blocks of code, and an `\inputpygments` command for bringing in and highlighting an external file. All of these have a mandatory argument that specifies the Pygments lexer to be used. For example, `\pygment{latex}{\pygment}` produces

```
\pygment
```

in a form suitable for inline use while

```
\begin{pygments}{python}
def f(x):
    return x**3
\end{pygments}
```

creates

```
def f(x):
    return x**3
```

The `pygments` environment and the `\inputpygments` command accept an optional argument containing `fancyvrb` settings.

As far as the author is aware, PythonTeX is the only LaTeX package that provides Pygments highlighting with Unicode support under the standard pdfTeX engine. The `listings` package [LST], probably the most prominent non-Pygments highlighting package, does support Unicode—but only if the user follows special procedures that could become tedious. PythonTeX requires no special treatment of Unicode characters, so long as the `fontenc` and `inputenc` packages are loaded and used correctly. For example, PythonTeX can correctly highlight the following snippet copied and pasted from a Python 3 console session, without any modification.

```
>>> var1 = 'âæéöø'
>>> var2 = 'ßçñðŠ'
>>> var1 + var2
'âæéöøßçñðŠ'
```

Implementation

A brief overview of the internal workings of PythonTeX is provided below. For additional details, please consult the documentation.

When a LaTeX document is compiled, the PythonTeX commands and environments write their contents to a single shared

external file. The command and environment contents are interspersed with delimiters, which contain information about the type of command or environment, the session in which the code is to be executed, the document line number where the code originated, and similar tracking information. A single external file is used to minimize the number of temporary files created, and because TeX has a very limited number of output streams.

During compilation, each command and environment also checks for any Python-generated content that belongs to it, and brings in this content if it exists. Python-generated content is brought in via LaTeX macros and via separate external files. At the beginning of the LaTeX document, the PythonTeX package brings in two files of LaTeX macros that were created on the Python side, if these files exist. One file consists of macros containing the Python content accessed by `\py`, `\pylab`, and `\sympy`. The other file contains highlighted Pygments content. The files are separate for performance reasons. In addition to content that is brought in via macros, content may be brought in via separate external files. Each command or environment that uses the print statement/function must bring in an external file containing the printed content. The printed content cannot be brought in as LaTeX macros, because in general printed content need not be valid LaTeX code. In contrast, `\py`, `\pylab`, and `\sympy` should return valid LaTeX, and of course Pygments-highlighted content is valid LaTeX as well.

On the Python side, the file containing code and delimiters must be processed. All code is hashed, to determine what has been modified since the previous run so that only new and modified code may be executed. Code that must be executed is divided by session, and each session (plus some PythonTeX management code) is saved to its own external file. The highlighting settings for Pygments content are compared with the settings for the last run, to determine what needs to be highlighted again with new settings.

Next, Python's `multiprocessing` package is used to perform all necessary tasks. Each of the session code files is executed within a separate process. The process executes the file, parses the stdout into separate files of printed content based on the command or environment from which it originated, and parses the stderr to synchronize it with the document line numbers. If specified by the user, a modified version of the stderr is created and saved in an external file for inclusion in the document via `\stderrpythontex`. Two additional processes are used, one for highlighting code with Pygments and one for evaluating and highlighting all console content (using Python's `code` module).

Finally, all LaTeX macros created by all processes are saved in one of two external files, depending on whether they contain general content or content highlighted by Pygments (again, this is for performance reasons). All information that will be needed the next time the Python side runs is saved. This includes the hashes for each session. Any session that produced errors is automatically set to be executed the next time the Python side runs. A list of all files that were automatically created by PythonTeX is also saved, so that future runs can clean up outdated and unused files.

PythonTeX consists of a LaTeX package and several Python scripts. A complete compilation cycle for a PythonTeX document involves running LaTeX to create the file of code and delimiters, running the PythonTeX script to create Python content, and finally running LaTeX again to compile the document with Python-generated content included. Since all Python-generated content is saved, the PythonTeX script only needs to be run when the document's PythonTeX commands or environments are modified. By default, all files created by PythonTeX are kept in a subdirectory within the document directory, keeping things tidy.

Conclusion

PythonTeX provides access to the full power of Python from within LaTeX documents. This can greatly simplify the creation of scientific documents and presentations.

One of the potential drawbacks of using a special LaTeX package like PythonTeX is that publishers may not support it. Since PythonTeX saves all Python-generated content, it already provides document compilation without the execution of any Python code, so that aspect will not be an issue. Ideally, a PythonTeX document and its Python output could be merged into a single, new document that does not require the PythonTeX package. This feature is being considered for an upcoming release.

PythonTeX provides many features not discussed here, including a number of formatting options and methods for adding custom code to all sessions. PythonTeX is also under active development. For additional information and the latest code, please visit https://github.com/gpoore/pythontex.

References

[Leisch] F. Leisch. *Sweave: Dynamic generation of statistical reports using literate data analysis*, in Wolfgang Härdle and Bernd Rönz, editors, Compstat 2002 - Proceedings in Computational Statistics, pages 575-580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9. http://www.statistik.lmu.de/~leisch/Sweave/.

[Ehmsen] M. R. Ehmsen. "Python in LaTeX." http://www.ctan.org/pkg/python.

[Drake] D. Drake. "The SageTeX package." https://bitbucket.org/ddrake/sagetex/.

[Molteno] T. Molteno. "The sympytex package." https://github.com/tmolteno/SympyTeX/.

[SymPy] SymPy Development Team. "SymPy." http://sympy.org/.

[Pyg] The Pocoo Team. "Pygments: Python Syntax Highlighter." http://pygments.org/.

[FV] T. Van Zandt, D. Girou, S. Rahtz, and H. Voß. "The 'fancyvrb' package: Fancy Verbatims in LaTeX." http://www.ctan.org/pkg/fancyvrb.

[MPL] J. D. Hunter. *Matplotlib: A 2D Graphics Environment*, in Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95. http://matplotlib.sourceforge.net/.

[NP] Numpy developers. "NumPy." http://numpy.scipy.org/.

[LST] C. Heinz and B. Moses. "The Listings Package." http://www.ctan.org/tex-archive/macros/latex/contrib/listings/.

[IPY] The IPython development team. "The IPython Notebook." http://ipython.org/notebook.html.

[Pastell] M. Pastell. "Pweave - reports from data with Python." http://mpastell.com/pweave/.

[Knuth] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. Stanford, California: Center for the Study of Language and Information, 1992.

[Ramsey] N. Ramsey. *Literate programming simplified*. IEEE Software, 11(5):97-105, September 1994. http://www.cs.tufts.edu/~nr/noweb/.

[Schwab] M. Schwab, M. Karrenbach, and J. Claerbout. *Making scientific computations reproducible*. Computing in Science & Engineering, 2(6):61-67, Nov/Dec 2000.

[Xie] Y. Xie. "knitr: Elegant, flexible and fast dynamic report generation with R." http://yihui.name/knitr/.

[MK] M. Bayer. "Mako Templates for Python." http://www.makotemplates.org/.

Self-driving Lego Mindstorms Robot

Iqbal Mohomed[‡*]

Abstract—In this paper, I describe the workings of my personal hobby project - a self-driving lego mindstorms robot. The body of the robot is built with Lego Mindstorms. An Android smartphone is used to capture the view in front of the robot. A user first teaches the robot how to drive; this is done by making the robot go around a track a small number of times. The image data, along with the user action is used to train a Neural Network. At run-time, images of what is in front of the robot are fed into the neural network and the appropriate driving action is selected. This project showcases the power of python's libraries, as they enabled me to put together a sophisticated working system in a very short amount of time. Specifically, I made use of the Python Image Library to downsample images, as well as the PyBrain neural network library. The robot was controlled using the nxt-python library.

Index Terms—self-driving, neural networks, robotics

Introduction

Recently, there has been significant interest in building self-driving cars. Starting in 2004, a series of competitions were held as part of the DARPA Grand Challenge, wherein driverless vehicles outfitted with sophisticated sensing equipment navigated real-world terrain. While none of the entrants in the 2004 iteration of the competition made it to the finish line, in the 2005 iteration, five driverless vehicles successfully completed the course. More recently, there have been many exciting developments in this area, with the development of Google's Driverless Car, and the US state of Nevada beginning the process to legalize and issue licenses for self-driving cars.

Building a self-driving car requires expensive sensing equipment. For example, the stanford entry in the DARPA grand challenge had 5 different laser measurement system [Mon08]. It is interesting to consider, if it is possible to create a self-driving car only using data from a camera. Around 1993, CMU created a learning system called "ALVINN" (Autonomous Land Vehicle In a Neural Network) [Pom93], which could control a testbed vehicle to drive on a variety of road surfaces. ALVINN worked by first "watching" a human driver's response to road conditions . After just 5 minutes of such training data in new situations, ALVINN could be trained to drive on a variety of road surfaces, and at speeds of unto 55 mies per hour. At first blush, it is starting that simply feeding image data and driver response to train a neural network would lead to a working autonomous vehicle. Earlier this year, David Singleton put up a blog post describing his weekend project – a self-driving RC car [Sin12]. As the project dealt with a small RC vehicle in an indoor environment, the technique was simpler than that used in ALVINN. I was inspired by David's post and decided to independently replicate this project using a Lego Mindstorms robot instead of an RC car. While I started out with limited experience using Neural Networks, I succeeded in my endeavor to create a self-driving robot that can navigate a track in an indoor environment. Figure 1 shows the lego robot in action.

The purpose of this paper is to share details of how I utilized a set of Python libraries - nxt-python to control the Lego robot, Python Image Library (PIL) to process camera images, and the pyBrain library to train and use an artificial neural network - to build a self-driving Lego Mindstorms robot.

Robot Construction

I used the Lego Mindstorms NXT 2.0 set to construct the robot. The set consists of various construction elements, motors and sensors. A key element of the set is a microcomputer called the NXT Intelligent Brick. The NXT brick contains a 32-bit ARM7 microprocessor, flash memory, a battery holder, USB 2.0 port, supports Bluetooth communication and also has ports for connecting sensors and motors. While the Lego Mindstorms set contains a variety of sensors, I did not utilize any of them for this project. The motors in the Lego set are Interactive Servo Motors. Unlike regular hobbyist motors, these motors can be rotated a specific number of degrees (the Lego motors are precise to 1 degree).

The robot I constructed has two independent tracks (each controlled by a separate motor). The motors are powered and controlled by an NXT brick, which is mounted on top of the tracks. The most challenging part of the robot build was creating a secure holder for my smart phone. While the phone has two cameras (front and back), I made the glass of the phone face backwards. This resuled in an acceptable camera mounting. I took care to have the images coming from the camera show what is directly in front of the robot and minimize views that are far away. That said, I did not have to spend too much effort in optimizing the camera view. The mount I created also had a mechanism to quickly release the phone, which was useful in debugging, charging the phone, and other activities. Figure 2 shows a close up of the phone mount on the robot, and figure 3 shows a schematic of the mount.

Setup

All my code (written in Python) runs on a Windows 7 PC. The PC communicates with the Lego NXT brick via Bluetooth. An Android camera phone (Google Nexus S) is attached to the Lego robot. The phone is connected to my home wireless network, as is

* *Corresponding author: iqbal@us.ibm.com*
‡ *IBM Research*

my PC. Figure 4 shows a diagram of communication between the various components.

Driving the Lego Mindstorms robot

I used the nxt-python library to interface my PC to the Lego Mindstorms robot. While the NXT brick does possess flash memory to allow programs to reside on the robot itself, nxt-python works by executing on a PC and sending short commands to the NXT brick via Bluetooth or USB. As I want the robot to be untethered, I make use of Bluetooth.

I made use of an experimental class in nxt-python called "SynchrnonizedMotors" that makes the motors controlling the left and right track to move in unision. If care were not taken to move the two motors together, the robot could drift to one side when the intent is to move straight ahead. Ultimately, the key requirement for the robot's motion is consistency. In my implementation, the robot had three movement options: it could move straight ahead, turn left or turn right. Each action went on for a short period of time. The right motor (which ran the right-side track) is connected to PORT A of the NXT brick. Analagously, the left motor is connected to PORT B. In nxt-python, we can create a Motor object that represents the motor and provides a function like turn() to move the interactive servo motor to a specific position using a given amount of power.

```
import nxt

def initBrick():
    # Define some globals to simplify code
    global b, r, l, m, mls, mrs
    # Search and connect to NXT brick (via BT)
    b = nxt.find_one_brick()
    # Create objects to control motors
    r = nxt.Motor(b, nxt.PORT_A)
    l = nxt.Motor(b, nxt.PORT_B)
    # Create objects for synchronized motors
    # We specify the leader, follower and turn ratio
    m = nxt.SynchronizedMotors(r,l, 0)
    mls = nxt.SynchronizedMotors(l, r, 20)
    mrs = nxt.SynchronizedMotors(r, l, 20)

# The first parameter to turn() indicates
# 100% or full power. To run the motor backwards,
# a negative value can be provided. Amt indicates the
# number of degrees to turn.
def go(dev,amt):
    dev.turn(100,amt);
```

To facilitate the collection of training data, I implemented a "keyboard teleop" mode, wherein I type commands into a python CLI and get my robot to make the appropriate movement (i.e. go straight, go left or go right).

```
# cmd param is the character typed by the user
def exec_cmd(cmd):
    if cmd == 'f':
        go(m,250)
    elif cmd == 'l':
        go(mls,250)
    elif cmd == 'r':
        go(mrs,250)
    elif cmd == 'x':
        b.sock.close()
```

Getting images from the camera phone

I initially thought about writing my own app to capture images from my phone (an Android Nexus S). However, I found a free app called IP Webcam that allowed me to take snapshots from

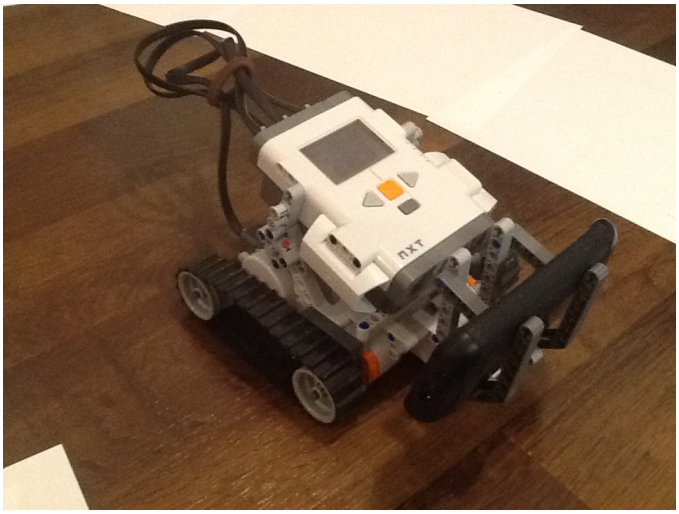


Fig. 1: *An image of the lego robot as it is driving along its course.*

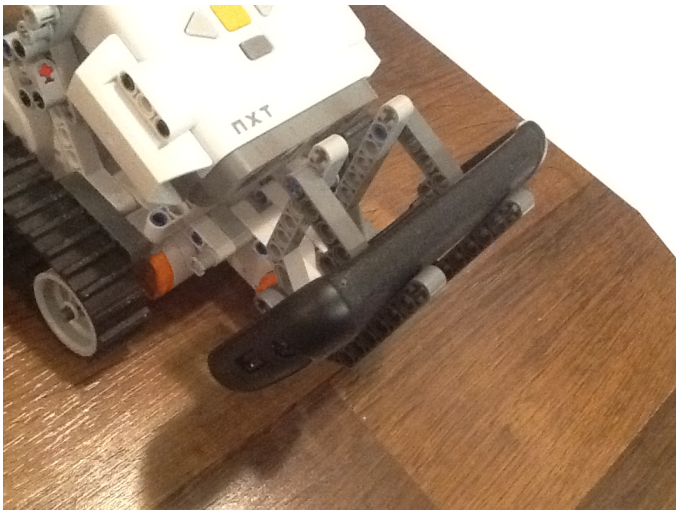


Fig. 2: *A close up look at the holder mechanism for the Android phone.*

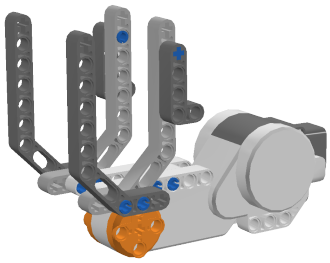


Fig. 3: *A schematic of the holder mechanism for the Android phone. Credit goes to Saira Karim for drawing the diagram using the free Lego Digital Designer software*

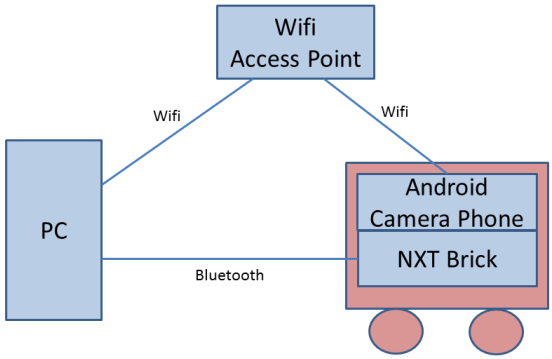


Fig. 4: *A diagram showing communication between various components.*

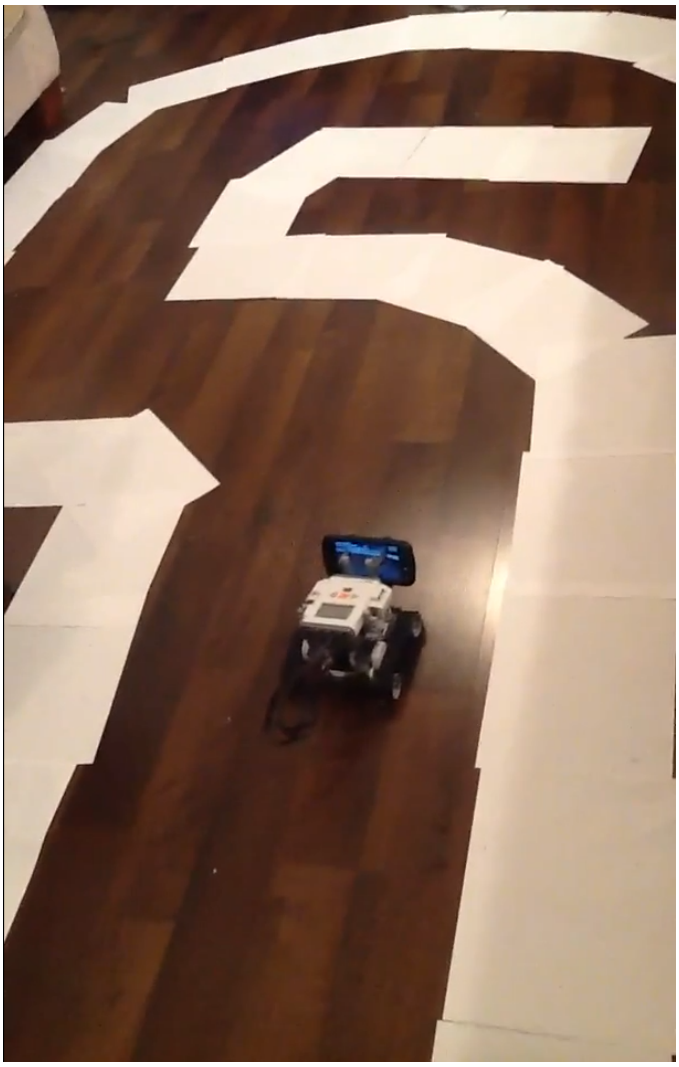


Fig. 5: *A view of the robot driving on the track.*

the phone via HTTP. Note that the IP address in the URL used to retrieve the image corresponds to the address assigned to the Android phone by the wireless AP. The lowest resolution at which I could get images was 176×144; I processed these images on the desktop before sending them to the neural network.

```
import urllib
res=urllib.urlretrieve('http://192.168.1.12:8080/shot.jpg')
```

Processing the images on desktop

I used the Python Imaging Library to first convert the images from the camera phone to greyscale and then lower their resolution to 100×100.

```
from PIL import Image
im = Image.open(res[0])
nim = im.convert('L')
nim2 = nim.resize((100,100))
```

I combine the two code fragments above into a function called take_pic(), which captures an image from the Android phone, transforms it and returns the result.

Obtaining training data

In order to teach the Lego robot how to drive, one must first obtain training data. Each sample of the training data consists of a low resolution greyscale image showing what is in front of the robot, and a human driver's action in that situation.

```
# This function accepts a command from the
# user via the keyboard, and executes it on the robot
def accept_execute_cmd():
    cmd = '';
    gotCmd = False;
    print "CMD: "
    while gotCmd == False:
        cmd = getch();
        #cmd = raw_input('CMD: ')
        if cmd == 'f' or cmd == 'l' or cmd == 'r':
            exec_cmd(cmd)
            gotCmd = True;
        elif cmd == 'x':
            b.sock.close()
            gotCmd = False;
            exit();
    print cmd + "\n";
    return cmd;


def trainer():
    while True:
        # download pic from camera and downsample
        im=take_pic()
        # get cmd from user and run it
        cmd = accept_execute_cmd()
        # record the image and cmd
        record_data(im,cmd)
```

Enter the Neural Network

This was the key part of the project. To learn about Neural Networks, I went through Professor Andrew Ng's lectures on Neural Networks, and played around with the assignments on the topic (recognizing hand-written digits using Neural Networks). Luckily, I found the pyBrain project, which provides a very easy interface for using Neural Nets in Python. Similar to David Singleton, I used a three level network. The first layer had 100×100 nodes. Each input node corresponds to a greyscale image captured from the camera phone. The hidden layer had 64 units (I tried other

values, but like David, 64 hidden units worked well for me too). Unlike David, I only had three output units – forward, left and right.

```
from pybrain.tools.shortcuts import buildNetwork
from pybrain.datasets import SupervisedDataSet
from pybrain.supervised.trainers import BackpropTrainer
net = buildNetwork(10000,64,3,bias=True)
ds = SupervisedDataSet(10000,3)
```

Training the brain

I built a "driving course" in my living room (shown in Figure 5). I drove around the course only 10 times and trained network for about an hour.

```
def train(net,ds,p=500):
   trainer = BackpropTrainer(net,ds)
   trainer.trainUntilConvergence(maxEpochs=p)
   return trainer
```

Auto-drive mode

The code for auto-drive mode was pretty similar to training mode. I took an image from the camera phone, processed it (greyscale and lowered the res to 100×100) and activated it against the neural net I had trained. The output is one of three commands (forward, left or right), which I send to the same "drive(cmd)" function I used in training mode. I put a short sleep between each command to ensure the robot had enough time to complete its motion.

```
# The following function takes the Neural Network
# and the processed image as input. It returns
# the action selected by activating the neural
# net.
def use_nnet(nnet,im):
   cmd = ''
   lst = list(im.getdata())
   res=nnet.activate(lst)
   val = res.argmax()
      if val == 0:
         cmd = 'f'
      elif val == 1:
         cmd = 'l'
      elif val == 2:
         cmd = 'r'
   return cmd

# The auto() function takes a trained
# neural network as input, and drives
# the robot. Each time through the loop,
# it obtains an image from the phone (and
# downsamples it). The image data is used
# to activate the Neural Network, the
# output of which is executed on the robot.
def auto(nnet):
   while True:
   im=take_pic()
   cmd=use_nnet(nnet,im)
   exec_cmd(cmd)
   print "executing .." + cmd
   time.sleep(3)
```

The Self-Driving Lego Mindstorms Robot comes to life!

It worked! Mostly. About 2/3 of the time, the robot could go through the entire course without any "accidents". About 1/3 of the time, the robot's motion takes it to a point where it can only see the track (sheets of white paper). When it gets to that state, it keeps going forward instead of making a turn. I have posted videos to YouTube as well as a blog post on my attempts [Moh12]. Implementing a "spin-90-degrees" command might help the robot get out of that situation. But all-in-all, I'm pretty happy with the results.

Conclusions

In this paper, I detail the workings of my self-driving Lego Mindstorms robot. The heart of the project is a neural network, which is trained with camera images of the "road" ahead and user input. At run time, the same camera images are used to activate the neural network, and the resulting action is executed on the robot. While a simple vision-based system cannot be expected to perform flawlessly, acceptable performance was achieved. My experience suggests that Python programmers can utilize neural networks and camera images to quickly build other interesting applications.

Source Code

All the source code I wrote for this project is publicly available on GitHub (https://github.com/iqbalmohomed/selfdrivingrobot.git).

Acknowledgements

Big thanks go to my wife (Saira Karim) for helping me with the project. I'd like to reiterate that this project was inspired by David Singleton's self-driving RC car and was an independent implementation of his work. Many thanks go to him. A big thank you to Prof. Andrew Ng for the Stanford Machine Learning class that is freely provided online. And a thanks to the following projects that made mine possible: nxt-python, pybrain, python-imaging-library, and the free IP Webcam Android App.

Disclaimers

This work was done as part of a personal hobby project. The views and opinions expressed are my own, and not related to my employer. LEGO, LEGO Mindstorms, ARM and Bluetooth are trademarks owned by their respective owners. E&OE

References

[Sin12] David Singleton. *How I built a neural network controlled self-driving (RC) car!* http://blog.davidsingleton.org/nnrccar

[Moh12] Iqbal Mohomed. *Self-driving Lego Mindstorms Robot* http://slowping.com/2012/self-driving-lego-mindstorms-robot/

[Tur98] Matthew A. Turk, David G. Morgenthaler, Keith D. Gremban, and Martin Marra. *VITS-A Vision System for Autonomous Land Vehicle Navigation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 10(3):342-361, May 1988.

[Pom93] Dean A. Pomerleau. *Knowledge-based Training of Artificial Neural Networks for Autonomous Robot Driving*, Robot Learning, 1993.

[Mon08] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. *Junior: The Stanford entry in the Urban Challenge*, Journal of Field Robotics, 25(9):569-597, September 2008.

The Reference Model for Disease Progression

Jacob Barhak[‡*]

Abstract—The Reference Model for disease progression is based on a modeling framework written in Python. It is a prototype that demonstrates the use of computing power to aid in chronic disease forecast. The model uses references to publicly available data as a source of information, hence the name for the model. The Reference Model also holds this name since it is designed to be used by other models as a reference. The model uses parallel processing and computing power to create a competition among hypothesis of disease progression. The system runs on a multi core machine and is scalable to a SLURM cluster.

Index Terms—Disease Models, High Performance Computing, Simulation

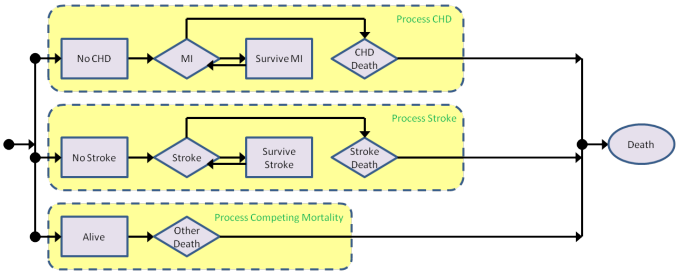


Fig. 1: *The Reference Model structure.*

Introduction

Disease modeling is a field where disease progression and its impact are studied. The field combines clinical knowledge, bio statistics, health economics, and computer science to create models. Such models are potential candidates for disease forecast tasks.

Within chronic disease models, most models target high risk diseases such as Coronary Heart Disease (CHD) and Stroke, especially with diabetes [McE10] , [Wil98], [Ste01], [Kot02], [Cla04], [Ste04], [Hip08], [Zet11], [CDC02], [EBMI], [Mich], [Bar10], [Edd03], [Par09]. Yet there are other models such as cancer models [Urb97], models to asses antidepressant cost-effectiveness [Ram12], infectious disease models [Gin09], and more complex models for policy makers [Che11].

Although models differ from each other in structure and implementation, most models can be defined as a function that maps initial population to clinical outcomes after a certain number of years. Once clinical outcomes are established, it is possible to derive quality of life and costs [Cof02], [Bra03]. Hence chronic disease models can predict economic impact or calculate cost effectiveness and therefore be valuable tools for decision makers.

Never the less, past behavior does not ensure future behavior, especially under different circumstances. Therefore the function the model represents is hard to calculate. Not only it depends on many factors such as biomarkers, health state, and treatment, it may also change with time and with unknowns. Moreover, models that work well on one population may work poorly on another population. Therefore it is recommended to validate a model against many populations [Her03]. It is interesting to see how different models behave on the same population. The latter procedure is accomplished at the Mount Hood conference where diabetes models are compared [MH4]. Mount Hood 6 held in 2012 and included 8 models from around the world. The Reference Model [Bar12] was a new model based on Python that joined the challenge and it is the main topic of this paper.

* *Corresponding author: jacob.barhak@gmail.com*
‡ *freelancer*

The Reference Model

The prototype version of the model consists of three main processes: CHD, Stroke, and Competing Mortality. The model structure is shown in figure 1.

The Reference Model is composed from references to publicly available literature. This is one reason for the name. Another reason is since the model is designed to act as a reference for model developers to test new populations or equations.

Equations are typically extracted from clinical trials and represent phenomena encountered in the trial populations. One equation may perform differently on other populations where another equation may have better performance. Moreover, different equations rely on different parameters as shown in Figure 2.

The Reference Model is a platform where different equations can be tested on different populations to deduce fitness. Moreover, the system combines equations from different sources and tests the fitness of these equation combinations. Such a combination of equations can include hypothesis, so the user can test "what if" scenarios in case of uncertainty.

An example of uncertainty is biomarker progression during the study. When a study is published it typically contains the initial population statistics in it, see table 1 in [Cla10], [ACC10], [Kno06]. These statistics are sufficient to recreate a population without need to access private and restricted data. Yet this information does not always include information on biomarker progression, so the modeler can create several hypothesis to see which fits best with the equations and population information.

This approach of using secondary published data expands the reach of the model. Instead of building a model on top of information extracted from a single population such as UKPDS [UKP98],

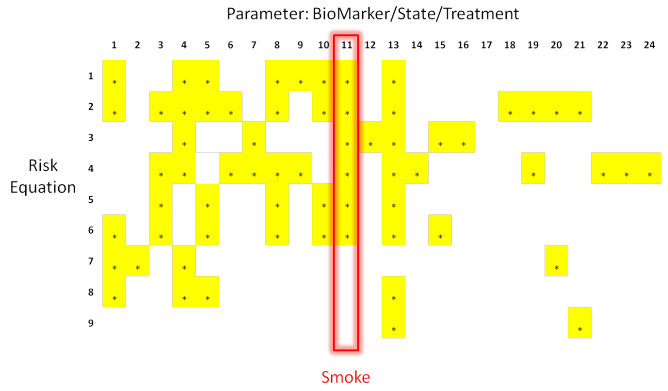


Fig. 2: *Different risk equations found in the literature and parameters they use. When an equation uses a specific parameter, it is highlighted. Smoking status is highlighted to demonstrate the fact that the 6 first equations test for smoking. The Reference Model uses these risk equations.*

the system uses models from multiple sources and compares them to multiple populations to find fitness between the sources of information. This enhances modeling capabilities significantly beyond a single data set and allows access to more populations overall. This is done while avoiding access to restricted individual data.

The Reference Model accomplishes these tasks by using computing power and a Python based disease modeling framework [Bar10], [Bar], [Mich]. There are several main aspects of the software that are important enablers of this work:

1) Simulation language
2) Population generator
3) Simulation engine
4) Simulation launcher

These components rely on the power of the Python language and will be discussed in the following sections.

Simulation language

The Python language exposes itself to the degree that it can be reused. The modeling framework takes advantage of this to create a language that is a based on Python.

The modeling framework allows the user to define variables and assemble expressions with a simple syntax. The syntax consists of numerical operations and a limited set of system functions. Users can also define their own functions.

Once an expression is entered to the system it is parsed to check if it will compile well. Expression validness is established by: 1) Check if tokens used are valid operators, system variables, or reserved words and are not recursively cyclically used, 2) Check if syntax is correct by parsing and evaluation.

The system uses the `parser` library to figure out grammar in conjunction with the `tokenize` library to handle variable names. The `re` library is used to figure out regular expression patterns within the expression. The `eval` command is used by the system as a validation tool by using an evaluation dictionary that consists only of `__builltins__`, `NaN`, `Inf` to handle expressions that support non finite values.

Errors reported during these checks are considered compilation errors that the user receives as feedback. Since the Python interpreter provides good and meaningful error messages, the system wraps Python error messages and returns these to the user, while adding its own language specific error messages.

After passing validity checks the expression is stored in the system database. This expression will be later reused once the system compiles files for runtime Python script.

For example, consider the following definition of a user defined function:

`CappedGaussian3` is defined as:

```
Max(-3,Min(3,Gaussian(0,1)))
```

Once entered into the system the expression undergoes the following stages:

1) The tokens `Max`, `Min`, and `Gaussian` are recognized as system functions and therefore valid.
2) The parser library successfully parses the expression, meaning there is no parenthesis mismatch or other syntax error.
3) The expression is evaluated to make sure evaluation is possible.
4) The expression is stored and whenever `CappedGaussian3` is used in the future, it will be replaced with that expression.

Here is an example of another expression that reuses the user defined function:

```
50+10*CappedGaussian3
```

This expression undergoes similar validity checks, yet `CappedGaussian3` is now recognized as a valid token representing a function. Also, it is verified that there is no recursive cyclic reuse of this token.

When this expression is compiled by the system into a Python script that will run, the expression will become Python code where the system will first calculate `CappedGaussian3` and then substitute the result in the expression that uses this token. The system will also recognize `Min`, `Max`, and `Gaussian` as system functions with Python implementation and will call those functions.

The runtime language is Python that implements the modeling language that is derived from Python. Keeping the languages close allowed shortening development time considerably.

The compiled files are executed by the population generator and by the simulation engine that will be discussed in the following sections.

Population Generator

Constructing a population from published data is a key element. The Reference Model cross validates against as many populations as possible. Since populations typically hold restricted healthcare data, full information on many populations is not available. Yet summary statistics are available through the publication. The modeling framework is designed to allow reconstruction of a population from these statistics.

Typically a clinical trial will describe the initial population by a table showing distributions. Table 1 will be used as a simplified example:

To implement this simplified example, the system will use the internal language previously discussed. Table 2 describes the implementation.

Age is assumed to be distributed with normal distribution. The user defined function we introduced previously is used to avoid

Biomarker	Distribution
Age	Mean 65, SD 7
Male	48%
Smoke	30% for Age<50, 15% for Age>=50

TABLE 1: *Simple example of population distributions.*

Biomarker	Implementation
Age	`65+7*CappedGaussian3`
Male	`Bernoulli(0.48)`
Smoke	`Bernoulli(0.15+0.15*Ls(Age,50))`

TABLE 2: *Implementation of the distributions in table 1.*

extreme outliers. Gender and Smoke use the Bernoulli distribution. However, Smoke has a dependency on the Age parameter. The system supports such dependencies in population creation by allowing the user to reference other parameters in the population and include these in expressions. The system raises an error in case of recursive cyclic references. This is important if the population is defined out of order, e.g. the Smoke formula uses Age before the Age distribution is defined. Actually, the system resolves the order by which calculations are performed when a population generation script is compiled.

The compiled Python script has the expressions in correct order and repeats the generation for each individual in the population. The output of running this script is a list of individuals, each with characteristics drawn from the given distributions. This mock population represents the starting conditions for the simulation engine to apply the model to.

Simulation Engine

The simulation engine has been described previously in [Bar10] and in the system documentation [Bar], [Mich]. Therefore this paper will only briefly relate to python related issues and many aspects are simplified.

The simulation engine applies a function to a vector for each simulation step for each individual. The function is complex and composed of rules, and state transitions happening in parallel in random order. The vector the function is applied to consists of biomarkers, states indicators, treatment parameters, costs and quality of life parameters. After each simulation step some values in the vector change and this updated vector will become the input for the same function in the next simulation step and so on. This continues until the individual dies or a predefined number of simulation steps is reached.

The modeling framework uses two mechanisms to compile the simulation files. 1) rule expansion to code, 2) state processing queue.

Rules are simple conditional assignments of the type

```
if Conditional and InState:
        AffectedParameter = Expression
```

Both the `Conditional` and the `Expression` are general expressions using the simulation language. Each of these expressions may contain user defined functions. The system compiles the code so that value bound checks can be incorporated into every calculated expression to maintain a strict simulation. Even though calculations are expanded, the compiled code is still readable and can be debugged since the compiled code uses the user defined names instead of `AffectedParameter` as variable names.

State transitions are handled by a queue that processes events of transition between states. The queue is loaded in random order and changes within simulation. Random order of events is important to allow scenarios where event 1 can happen before event 2, or event 2 happens before event 1 at the same simulation step. The Python implementation of the queue is such that the queue consists of functions that define transitions to check. These functions are automatically generated as Python code from the model structure as defined by the user. The Python script pops the next transition from the queue and calls it's state transition function. Each state transition function can change the queue or state indicator values. If this results in more events, those are added to the queue. For specific details, please consult the developer guide that arrives with the software [Bar], [Mich].

Note that, state indicator names and variable names are readable to simplify debugging and controlled simulation reconstruction.

Actually the implementation initializes the vector upon which the function is applied as a Python sequence of variables with names such as:

```
[Age, Gender, Smoke, ...]  =
  _PopulationSetInfo.Data[IndividualID-1]
```

Where `IndividualID` is the loop counter and `_Population` holds the population data created by the population generator after merging it with the model.

Actually, the population sequence is pickled and embedded within the code as well as all other data structures that created the simulation file. This is done to allow reconstructing the simulation conditions from the simulation file. This is also important for clerical back tracing purposes and for debugging.

Another such back tracing feature is saving the random state to file at the beginning of simulation. This is on top of allowing the user to select a random seed. This method facilitates recreation of a simulation even in the case of a random seed set by the system timer rather than set by the user.

The simulation engine supports simulation control through overrides. These overrides are essential to support multiple scenarios to run in parallel through the simulation launcher.

Simulation Launcher

Simulations can be launched through the WxPython [WxP] based GUI. This is appropriate for testing and model development purposes. Yet this is not sufficient for running many scenarios or many repetitions to reduce the Monte Carlo error. Moreover, running the same simulation with variations on populations or equations cannot be accomplished in an automated way through the GUI. Therefore the modeling framework offers an external launcher for simulations that arrives with it as a Python script.

The script `SlurmRun.py` allows running multiple simulations in a programmatic manner. The script controls the launch of the simulations and also responsible for collecting the results and creating csv reports and plots through `matplotlib`.

The `SlurmRun.py` script arrives as an example that runs a self test scenario on a SLURM cluster [SLU]. It is setup to send summary results by email to the developer once the simulation ends.

The script holds placeholders for modifications so that it can be adapted to new simulations. The basic idea behind the

launch script is that the user provides the file name that holds the simulation data definitions pickled an zipped. This file includes the default simulation instructions. The system then sends this file for actual simulation using `sbatch` SLURM commands. These defaults are then overridden according to user instructions.

User instructions include definitions of variations around the default simulation. Each such variation is described by a tuple consisting of an override value and a unique variation name string. The override value can be considered as an override for a parameter the simulation function relies on. These overrides are passed to the simulation engine in the command line as a vector. Each component in this vector represents a different override and taken from the value part of the tuple. Exchanging the override value with the unique variation name string creates a unique key sentence that can later be used to describe each simulation variation.

The number of simulation variations is combinatorial depending on amount of options for each override in the vector. Many combinations of variations may not be meaningful or desirable. So the system contains 3 variables to restrict the number of simulation variations: 1) `Inclusions`, 2) `Exclusions`, 3) `MaxDimensionsAllowedForVariations`.

`Inclusions` is a sequence of tuples. Each tuple is composed of a set of variation sub strings. If `Inclusions` in not empty the system will include only variations that their variation key sentence includes all the strings in any tuple.

`Exclusions` is also a sequence of tuples of strings. Yet the system excludes any variation that includes all sub strings in a tuple.

`MaxDimensionsAllowedForVariations` is the maximal Hamming distance from default allowed for simulation variations. In other words, it is an integer that holds the maximal number of override vector components allowed to change from the default.

These override mechanisms allow controlling the large number of combinations generated. The following example demonstrates the large number of variations.

The Reference Model calibration for the Mount Hood 6 Challenge used 16 populations and 48 equation/hypothesis variations. Each such simulation was repeated 20 times to reduce Monte Carlo error. This resulted in 15360 simulations that the system launched. The launcher was modified to run these simulations on a single 8 core desktop machine with Ubuntu using `batch` command rather than using the SLURM `sbatch` command. These computations took 4.5 days on this machine.

In the future more computing power will be needed to process information since more populations and equation variations will exist.

Conclusions

Previous work was focused on merging information that is available in the literature using statistical methods [Isa06], [Isa10], [Ye12]. The Reference Model continues in the same spirit while relying on the availability of computing power.

The Reference Model for disease progression relies on a Python based framework that provides the computational support needed for comparing a myriad of scenarios.

The state of the art in the field of chronic disease models is such that different groups offer different models. Each such model is built from equations that depend on different parameters. Therefore equation performance differs on different populations.

So far only a few groups have addressed the issues of comparing equation performance over populations [Sim09], [Par09]. Validation of the same model with multiple populations is more common [Edd03]. Comparisons of multiple Models against multiple populations traditionally happens at the Mount Hood conference [MH4]. Yet this comparison involves manual labor from multiple groups and much of the modeling remains closed. The Reference Model on the other hand performs this comparison automatically under controlled conditions. The Reference Model depends on availability of published information. It relies on existing equations and human guidance. Even with the automation it offers, modelers will still need to work on extracting new equations. Yet it's availability provides advantages such as: 1) a testing facility for new equations/hypothesis. 2) similarity identifier in data sets through fitness. 3) common framework for modeling information that can be reused in other ways.

From an implementation point of view, relying on parallelization and on the regular increase in computing speed [Moo65] may be enhanced by using compiled languages. Such needs have been identified in the disease modeling field [McE10] and by the Python community [Sel09], [Fri09]. So future implementations may include a python front end, while simulations will run in a compiled language to improve speed. Never the less, the use of the Python languange was a good selection for this project since it allowed rapid progress and many suitable tools.

Software Availability

The latest version of the GPL modeling framework is available for download from the author's personal website at: [Bar]. Previous versions are available at [Mich].

The Reference Model is not released at the time this paper is written.

Acknowledgment

The author wished to thank Deanna J.M. Isaman for her tutoring, inspiration, and introduction to this field. Special thanks to Chris Scheller the very capable cluster administrator.

The GPL modeling framework described above was supported by the Biostatistics and Economic Modeling Core of the MDRTC (P60DK020572) and by the Methods and Measurement Core of the MCDTR (P30DK092926), both funded by the National Institute of Diabetes and Digestive and Kidney Diseases. The modeling framework was initially defined as GPL and was funded by Chronic Disease Modeling for Clinical Research Innovations grant (R21DK075077) from the same institute.

The Reference Model was developed independently without financial support.

References

[Bar12] J. Barhak. *The Reference Model for Disease Progression You Tube*: Online: http://www.youtube.com/watch?v=7qxPSgINaD8&feature=youtube_gdata_player

[McE10] P. McEwan, K. Bergenheim, Y. Yuan, A. Tetlow, J. P. Gordon *Assessing the Relationship between Computational Speed and Precision: A Case Study Comparing an Interpreted versus Compiled Programming Language using a Stochastic Simulation Model in Diabetes Care*, Pharmacoeconomics: 28(8):665-674, August 2010, doi: 10.2165/11535350-000000000-00000

[Wil98] P.W.F. Wilson, R.B. D'Agostino, D. Levy, A.M. Belanger, H. Silbershatz, W.B. Kannel, *Prediction of Coronary Heart Disease Using Risk Factor Categories*, Circulation 97:1837-1847, 1998.

[Ste01] R. Stevens, V. Kothari, A. Adler, I. Stratton, *The UKPDS risk engine: a model for the risk of coronary heart disease in type II diabetes UKPDS 56*, Clinical Science 101:671-679, 2001.

[Kot02] V. Kothari, R.J. Stevens, A.I. Adler, I.M. Stratton, S.E. Manley, H.A. Neil, R.R. Holman, *Risk of stroke in type 2 diabetes estimated by the UK Prospective Diabetes Study risk engine (UKPDS 60)*, Stroke 33:1776-1781, 2002, doi: 10.1161/01.STR.0000020091.07144.C7

[Cla04] P.M. Clarke, A.M. Gray, A. Briggs, A.J. Farmer, P. Fenn, R.J. Stevens, D.R. Matthews, I.M. Stratton, R.R. Holman, *A model to estimate the lifetime health outcomes of patients with type 2 diabetes, UK Prospective Diabetes Study (UKPDS): the United Kingdom Prospective Diabetes Study (UKPDS) Outcomes Model (UKPDS no. 68)*, Diabetologia 47(10):1747-1759, 2004.

[Ste04] R.J. Stevens, R.L. Coleman, A.I. Adler, I.M. Stratton, D.R. Matthews, R.R. Holman, *Risk Factors for Myocardial Infarction Case Fatality and Stroke Case Fatality in Type 2 Diabetes: UKPDS 66*, Diabetes Care, 27:201-207, January 2004, doi:10.2337/diacare.27.1.201

[Hip08] J. Hippisley-Cox, C. Coupland, Y. Vinogradova, J. Robson, R. Minhas, A. Sheikh, P. Brindle, *Predicting cardiovascular risk in England and Wales: prospective derivation and validation of QRISK2*, BMJ 336:1475-1482, June 2008, doi: 10.1136/bmj.39609.449676.25x

[Zet11] B. Zethelius, B. Eliasson, K. Eeg-Olofsson, A.M. Svensson, S. Gudbjornsdottir, J. Cederholm, *A new model for 5-year risk of cardiovascular disease in type 2 diabetes, from the Swedish National Diabetes Register (NDR)*, Diabetes Res Clin Pract. 93(2):276-84. August 2011, doi:10.1016/j.diabres.2011.05.037

[CDC02] CDC Diabetes Cost-effectiveness Group. *Cost-effectiveness of intensive glycemic control, intensified hypertension control, and serum cholesterol level reduction for type 2 diabetes*, JAMA 287(19): 2542-2551, 2002, doi: 10.1001/jama.287.19.2542

[EBMI] *EBMI. Evidence-Based Medicine Integrator* Online: http://code.google.com/p/ebmi/ (Accessed: 8 July 2012).

[Mich] Michigan Diabetes Research and Training Center. *Disease Modeling Software for Clinical Research*, Online: http://www.med.umich.edu/mdrtc/cores/DiseaseModel/ (Accessed: 8 July 2012).

[Bar10] J. Barhak, D.J.M. Isaman, W. Ye, D. Lee, *Chronic disease modeling and simulation software*, Journal of Biomedical Informatics, 43(5): 791-799, 2010, doi:10.1016/j.jbi.2010.06.003

[Edd03] D.M. Eddy, L. Schlessinger, *Validation of the Archimedes Diabetes Model*. Diabetes Care 26(11):3102-3110, 2003, doi: 10.2337/diacare.26.11.3102

[Par09] S. Paredes, T. Rocha, P. Carvalho, J. Henriques, M. Harris, M. Antunes, *Long Term Cardiovascular Risk Models' Combination - A new approach*, 2009. Online: https://www.cisuc.uc.pt/publication/show/2175 (Accessed: 9 July 2012).

[Urb97] N. Urban, C. Drescher, R. Etzioni, C. Colby. *Use of a stochastic simulation model to identify an efficient protocol for ovarian cancer screening*, Control Clin Trials.18(3):251-70, June 1997, doi:10.1016/S0197-2456(96)00233-4

[Ram12] J. Ramsberg, C. Asseburg, M. Henriksson, *Effectiveness and cost-effectiveness of antidepressants in primary care - a multiple treatment comparison meta-analysis and cost-effectiveness model*, PLoS One 2012 (Accepted).

[Gin09] J. Ginsberg, M.H. Mohebbi1, R.S. Patel1, L.Brammer, M.S. Smolinski, L. Brilliant, *Detecting influenza epidemics using search engine query data*, Nature (457)19, February 2009, doi:10.1038/nature07634

[Che11] C.H. Chen-Ritzo, *Simulation for Understanding Incentives in Healthcare Delivery*, IBM, T.J. Watson Research Center, 7 December 2011. Online: http://www-304.ibm.com/industries/publicsector/fileserve?contentid=228827 (Accessed: 13 July 2012)

[Cof02] J.T. Coffey, M. Brandle, H. Zhou, D. Marriott, R. Burke, B.P. Tabaei, M.M. Engelgau, R.M. Kaplan, W.H. Herman, *Valuing health-related quality of life in diabetes*, Diabetes Care 25:2238-2243, 2002, doi: 10.2337/diacare.25.12.2238

[Bra03] M. Brandle, H. Zhou, B.R.K. Smith, D. Marriott, R. Burke, B.P. Tabaei, M.B. Brown, W.H. Herman, *The direct medical cost of type 2 diabetes*, Diabetes Care 26:2300-2304, 2003, doi: 10.2337/diacare.26.8.2300

[Her03] W.H. Herman. *Diabetes modeling*, Diabetes Care 26:3182-3183, November 2003. doi: 10.2337/diacare.26.11.3182

[MH4] The Mount Hood 4 Modeling Group, *Computer Modeling of Diabetes and Its Complications, A report on the Fourth Mount Hood Challenge Meeting*, Diabetes Care 30:1638-1646, 2007, doi: 10.2337/dc07-9919

[Cla10] P.M. Clarke, P.G. A. Patel, J. Chalmers, M. Woodward, S.B. Harrap, J.A. Salomon, on behalf of the ADVANCE Collaborative Group. *Event Rates, Hospital Utilization, and Costs Associated with Major Complications of Diabetes: A Multicountry Comparative Analysis*, PLoS Med 7(2):e1000236, 2010, doi:10.1371/journal.pmed.1000236

[ACC10] The ACCORD Study Group, *Effects of Intensive Blood-Pressure Control in Type 2 Diabetes Mellitus*, N Engl J Med 362:1575-85, March 2010, doi:10.1056/NEJMoa1001286

[Kno06] R.H. Knopp, M. d'Emden, J.G. Smilde, S.J. Pocock, *Efficacy and Safety of Atorvastatin in the Prevention of Cardiovascular End Points in Subjects With Type 2 Diabetes: The Atorvastatin Study for Prevention of Coronary Heart Disease Endpoints in Non-Insulin-Dependent Diabetes Mellitus (ASPEN)*, Diabetes Care 29:1478-1485, July 2006, doi:10.2337/dc05-2415

[UKP98] UKPDS: *UK Prospective Diabetes Study UKPDS Group, Intensive blood-glucose control with sulphonylureas or insulin compared with conventional treatment and risk of complications in patients with type 2 diabetes UKPDS 33*, Lancet 352:837-853, 1998, doi:10.1016/S0140-6736(98)07019-6

[Bar] *Jacob Barhak*, Online: http://sites.google.com/site/jacobbarhak/ (Accessed: 14 July 2012)

[WxP] *WxPython*, Online: http://wxpython.org/ (Accessed: 14 July 2012)

[SLU] *SLURM: A Highly Scalable Resource Manager*, Online: https://computing.llnl.gov/linux/slurm/slurm.html (Accessed: 9 July 2012).

[Isa06] D.J.M. Isaman, W.H. Herman, M.B. Brown, *A discrete-state and discrete-time model using indirect estimates*, Statistics in Medicine 25:1035-1049, 2006, doi:10.1002/sim.2241.

[Isa10] D.J.M. Isaman, J. Barhak, W. Ye, *Indirect estimation of a discrete-state discrete-time model using secondary data analysis of regression data*, Statistics in Medicine 28(16):2095-2115, 2009, doi:10.1002/sim.3599. Erratum available: Statistics in Medicine 29(10):1158, 2010, doi: 10.1002/sim.3855

[Ye12] W. Ye, J. Barhak, D.J.M. Isaman, *Use of Secondary Data to Estimate Instantaneous Model Parameters of Diabetic Heart Disease: Lemonade Method*, Information Fusion 13(2):137-145, 2012, doi:10.1016/j.inffus.2010.08.003

[Sim09] R.K. Simmons, R.L. Coleman, H.C. Price, R.R. Holman, K. Khaw, N.J. Wareham, S.J. Griffin, *Performance of the UKPDS Risk Engine and the Framingham risk equations in estimating cardiovascular disease in the EPIC-Norfolk cohort*, Diabetes Care 32:708-713 December 2009, doi: 10.2337/dc08-1918

[Moo65] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics, 38(8), 19 April 1965

[Sel09] D. Seljebotn, *Fast numerical computations with Cython*, Proceedings of the 8th Python in Science conference (SciPy 2009), G Varoquaux, S van der Walt, J Millman (Eds.), pp. 15-22. Online: http://conference.scipy.org/proceedings/scipy2009/paper_2/ (Accessed: 12 July 2012)

[Fri09] A. Friedley, C. Mueller, A. Lumsdaine, *High-Performance Code Generation Using CorePy*, Proceedings of the 8th Python in Science conference (SciPy 2009), G Varoquaux, S van der Walt, J Millman (Eds.), pp. 23-28. Online: http://conference.scipy.org/proceedings/scipy2009/paper_3/ (Accessed: 12 July 2012)

Fcm - A python library for flow cytometry

Jacob Frelinger‡*, Adam Richards‡, Cliburn Chan‡

Abstract—Flow cytometry has the ability to measure multiple parameters of a heterogeneous mix of cells at single cell resolution. This has lead flow cytometry to become an integral tool in immunology and biology. Most flow cytometry analysis is performed in expensive proprietary software packages, and few opensource tool exist for working with flow cytometry data. In this paper we present *fcm*, an BSD licensed python library for traditional gating based analysis in addition to newer model based analysis methods.

Index Terms—Flow Cytometry, Model-based Analysis, Automation, Biology, Immunology

Introduction

Flow cytometry (FCM) has become an integral tool in immunology and biology due to the ability of FCM to measure cell properties at the single cell level for thousands to millions of cells in a high throughput manner. In FCM, cells are typically labeled with monoclonal antibodies to cell surface or intracellular proteins. The monoclonal antibodies are conjugated to different fluorochromes that emit specific wavelengths of light when excited by lasers. These cells are then streamed single file via a capillary tube where they may be excited by multiple lasers. Cells scatter the laser light in different ways depending on their size and granularity, and excited fluorochromes emit light of characteristic wavelengths. Scattered light is recorded in forward and side scatter detectors, and specific fluorescent emission light is recorded into separate channels. Since each fluorescent dye is attached to specific cell markers by monoclonal antibodies, the intensity of emitted light is a measure of the number of bound antibodies of that specificity [Herzenberg2006] . The data recorded for each cell is known as an event, although events may sometimes also represent cell debris or clumps. Modern instruments can resolve about a dozen fluorescent emissions simultaneously and hence measure the levels of a dozen different markers per cell - further increase in resolution is limited by the spectral overlap (spillover) between fluorescent dyes.

Analysis of FCM data has traditionally relied on expert interpretation of scatter plots known as dot plots that show the scattered light or fluorescence intensity for each cell depicted as a point. Expert operators examine these two dimensional dot plots in sequence and manually define boundaries around cell subsets of interest in each projection. The regions demarcated by these boundaries are known as gates, and the cell subsets of interest may require multiple levels of gates to identify. Much work is needed

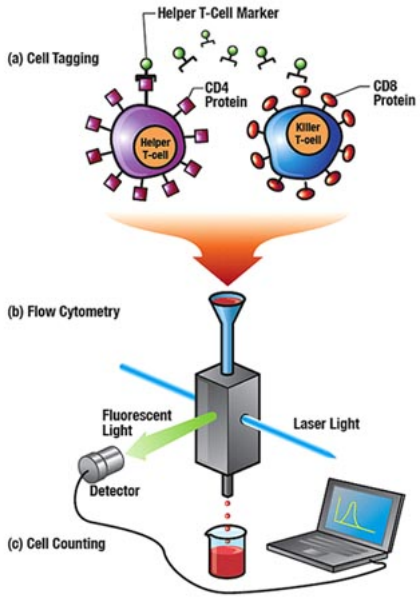


Fig. 1: *Diagram of how events are recorded in a flow cytometer provided by lanl.gov*

train expert operators to standardize gate placement and minimize variance. Maecker et al [Maecker2005] found a significant source of variability in a multi-center study was due to variability in gating. New technologies have the potential to greatly increase the number of simultaneous markers that can be resolved with FCM. Inductively coupled plasma mass spectrometry [Ornatsky2006] replaces the fluorescent dyes with stable heavy metal isotopes and fluorescent detection with mass spectrometry. This eliminates the spectral overlap (spillover) from fluorescent dyes allowing a significantly increased number of markers to be resolved simultaneously.

With the increasing number of markers that can be resolved simultaneously, there has been an increasing interest in automated methods of cell subset identification. While there is need for such tools, with the exception of the R BioConductor package, few open source packages exist for doing both traditional analysis and automated analysis. The majority of open source packages simply extract flow events into tabular/csv formats, losing all metadata and providing no additional tools for analysis. *fcm* attempts to resolve this by providing methods for working with flow data in both gating-based and model-based methods.

The goals in writing *fcm* [fcm] are to provide a general-purpose python library for working with flow cytometry data.

* *Corresponding author: jacob.frelinger@duke.edu*
‡ *Duke University*

Targeted uses include interactive data exploration with [ipython], building pipelines for batch data analysis, and development of GUI and web based applications. In this paper we will explore the basics of working with flow cytometry data using *fcm* and how to use fcm to perform analysis using both gating and model based methods.

Loading, compensating and transforming data

Flow cytometry samples that have been prepared and run through a flow cytometer generate flow cytometry standard (FCS) files, consisting of metadata about the sample, the reagents and instrument used, together with the scatter and fluorescent values for each event captured in the sample acquisition. These binary FCS files are then used to perform quality control and analysis of the data, typically with specialized proprietary software.

In *fcm*, the *loadFCS()* function will read in version 2 or 3 FCS files and return a *FCMdata* object. *FCMdata* objects contain the recorded scatter and fluorescent marker values for each event in an underlying numpy array, along with the associated metadata stored in the FCS file. In the FCS specification, metdata is stored in separate text, header and analysis sections in the original FCS file, and these can be accessed within a *FCMdata* instance from *FCMdata.notes.text*, *FCMdata.notes.header*, and *FCMdata.notes.analysis* respectively using either attribute or dictionary lookup conventions. The *FCMdata* object provides a few methods to directly manipulate the event data extracted from the FCS file, but mostly simply delegates to the underlying numpy array storing the event data matrix. Conveniently, this allows *FCMdata* objects to perform numpy array methods, such as *mean()* or *std()*, and also allows *FCMdata* objects to be passed to functions expecting numpy arrays. In addition to traditional numpy array indexing, the text names of channels can be used to access channels too.

```
In [1]: import numpy as np

In [2]: import fcm

In [3]: x = fcm.loadFCS('62851.fcs')

In [4]: x.channels[7]
Out[4]: 'AViD'

In [5]: np.all(x[:,7] == x[:,'AViD'])
Out[5]: True
```

When processing cells and acquiring data, often the emission spectra of fluorescent dyes overlap with neighboring channels. This spillover of light needs to be corrected in a process called compensation that attempts to remove the additional signal from neighboring channels. Using a compensation matrix that describes the amount of spillover from each channel into others, *fcm* will by default apply compensation at the time of loading data, but this default behavior can be suppressed and compensation performed at a later time if necessary. The spillover or compensation matrix is typically found in the *FCMdata.notes.text* metadata, and *loadFCS()* will default to compensating using that matrix if another is not specified.

Since FCM fluorescent data typically approximately follows a lognormal distribution, data is often transformed into log or log-like scales. *fcm* supports both log transforms and logicle [Parks2005] transforms as methods of *FCMdata* objects. *loadFCS()* will default to the logicle transform if the data is on the correct scale, that is if P#R value in the text segment is

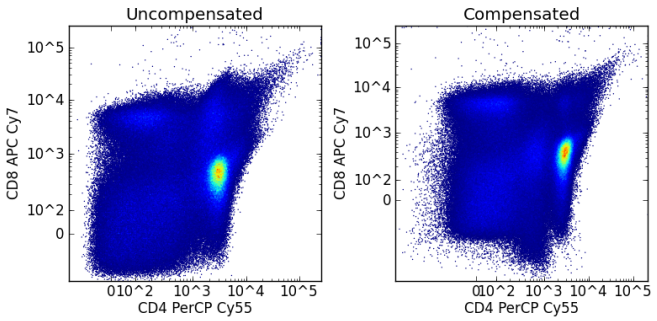


Fig. 2: *Compensation changes the data via matrix multiplication operation to reduce the spillover from other markers into each channel and can improve the resolution of individual cell populations.*

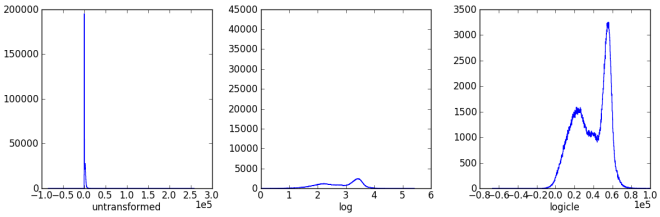


Fig. 3: *Illustration of the effects of logicle and log transform on CD3 AmCyan fluorescent from a FCS file from the EQAPOL data set.*

262144. Figure 3 illustrates the effects that transforming has on the distribution of events in each fluorescent channel.

Gating Analysis

In gating based analysis, the objective is to identify specific cellular subsets by sequentially drawing boundary regions, called gates, in a succession of one dimensional and two dimensional plots to select the cellular subsets of interest. Each successive gate captures increasingly specific cellular subsets. Once the required populations have been identified, summary statistics, typically mean or frequency, can easily be computed to compare with other populations.

fcm provides several gating objects to assist in traditional gating analysis of FCS files. Gate objects provided by *fcm* include *PolygonGate*, defining a region of interest by a set of vertices of the boundary of the region, *QuadrantGate*, dividing a two-dimensional projection into four quadrants defined by the point of intersection of all four quadrants, *ThresholdGate*, a region defined by all points above or below a point in a single parameter, and an *IntervalGate*, the set of points between two points in a single parameter. In addition to traditional gates, *fcm* provides additional gate like filters, *DropChannel*, to remove unwanted columns from a view, and *Subsample*, that use a python slice objects to filter events. *FCMdata* objects *gate()* method can be used to apply gate objects in successive manner as it returns the updated *FCMdata* object allowing chaining of *gate()* calls, like so:

```
FCMdata.gate(g1).gate(g2).gate(g3)
```

which is equivalent to the following three lines of code:

```
FCMdata.gate(g1)
FCMdata.gate(g2)
FCMdata.gate(g3)
```

In *fcm*, gating *FCMdata* object does not produce new *FCMdata* objects, but rather each *FCMdata* object maintains a tree of

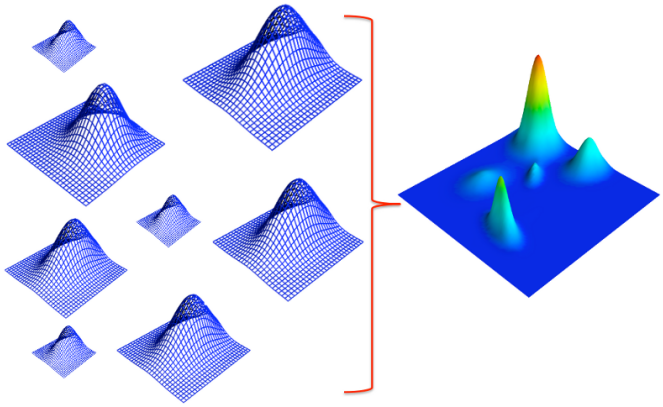


Fig. 4: *Mixture models are comprised of multiple simpler distributions. These simpler distributions are added together to describe more complex distributions. Using these simpler distributions, in this case multivariate normal distributions, it becomes possible to describe very complex distributions.*

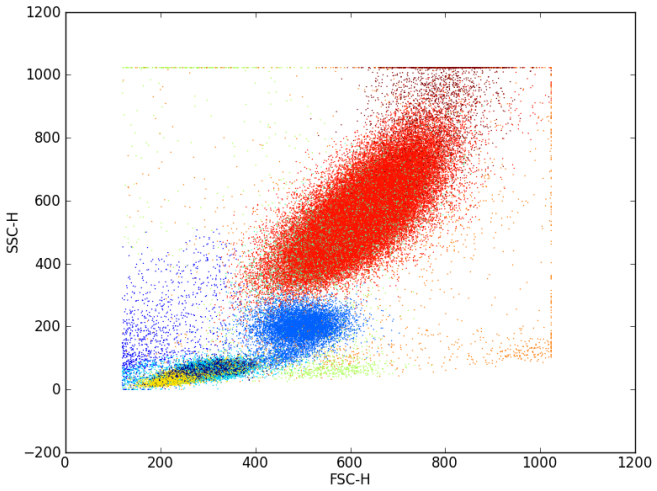


Fig. 5: *Events in a sample data set clustered by DPMixtureModel using Bayesian EM*

each gated populations. Moving between nodes of the tree, accomplished by the *FCMdata.visit()* method, selects which events are retured on array lookup, using *numpy*'s efficient indexing to generate views. This allows *FCMdata* objects to contain an entire analysis in a single object and reduces the need to keep multiple large high dimensional arrays in memory.

Model Based Analysis

As a result of the increasing dimensionality of FCM data resulting from technological advances, manual analysis is increasingly complex and time-consuming. Therefor there is much interest in finding automated methods of analyzing flow data. Model based analysis is an approach to automate and increase reproducibility in the analysis of flow data by the use of statistical models fitted to the data. With the appropriate multivariate statistical models, data fitting can be naturally performed on the full dimensionality, allowing analysis to scale well with the increasing number of parameters in flow cytometry. Mixture models are one such model based method. Mixture models are often chosen due to their ability to use multiple simpler distributions added together to describe a much more complex distribution as seen in figure 4.

fcm provides several model based methods for identifying cell subsets, the simplest method being k-means classification, and more advanced methods based on the use of mixtures of Gaussians for data fitting. The general procedure for fitting a data set to a statistical model consists of creating a *FCMmodel* object containing hyper-parameters, followed by calling its *fit* method on a collection of (or just one) *FCMdata* objects to generate *ModelResult* objects. Each *ModelResult* object holds the estimated parameters of the statistical model -- a *KMeans* object representing the centroid locations in a k-means model, or a *DPMixture* object representing the estimated weights, means and covariances for Gaussian mixture models. These objects can then be used to classify arbitrary datasets or to explore the estimated model parameters.

Gaussian mixture models describe events as coming from a mixture of multiple multivariate Gaussian distributions, where an event x comes from each Gaussian component with probability π_i, the weight. Hence the overall probability is

$$p(x|\pi,\mu,\sigma) = \sum_{i=1}^{k} \pi_i N(x|\mu_i,\sigma_i)$$

where N is a Gaussian, and x can be assigned to the Gaussian component with the highest probability. *fcm* provides two related mixture models to fit data from the [dpmix] package, which is capable of using [gpustats] to utilize GPU cards for efficient estimation of mixture parameters. The two models are *DPMixtureModel* and *HDPMixtureModel*, describing a truncated Dirichlet process mixture model, and a hierarchical truncated Dirichlet process mixture model.

DPMixtureModel has two methods of estimating parameters of the model for a given dataset, the first using Markov chain monte carlo (MCMC) and the second using Bayesian expectation maximization (BEM). Sensible defaults for hyperparameters have been chosen that in our experience perform satisfactorily on all FCS data samples we have analyzed.

```
import fcm
import fcm.statistics as stats
import pylab

#load FCS file
data = fcm.loadFCS('3FITC_4PE_004.fcs')

#ten component model fit using BEM for
# 100 iterations
dpmodel = stats.DPMixtureModel(10, niter=100,
    type='BEM')

# estimate parameters printing every 10 iterations
results = dpmodel.fit(data,verbose=10)

#assign data to components
c = results.classify(data)

# plot data coloring by label
pylab.scatter(data[:,0], data[:,1], c=c,
    s=1, edgecolor='none')

pylab.xlabel(data.channels[0])
pylab.ylabel(data.channels[1])
```

The above code labels each event by color to the cluster it belongs to as seen in figure 5

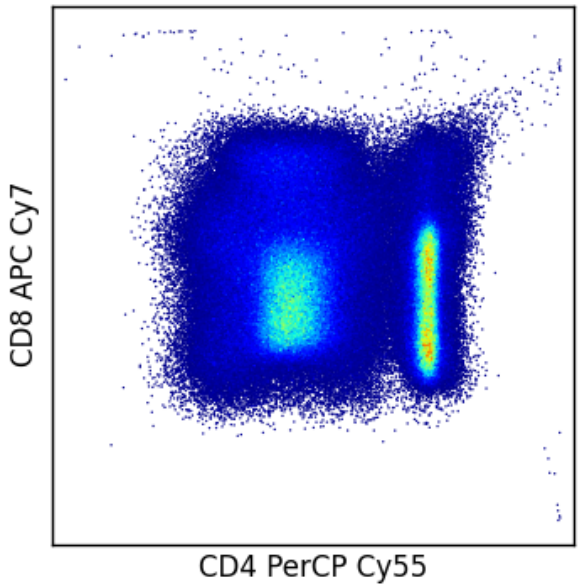


Fig. 6: *Pseudo-color heatmap produced by fcm.graphics.heatmap function showing CD4 versus CD8.*

HDPMixtureModel fits multiple data sets simultaneously so as to identify a hierarchical model that fits all datasets such that component means and covariance are common to all fitted samples but the weights of components are specific for each sample. Since *HDPMixtureModel* estimates multiple datasets simultaneously, a list of *DPMixture* objects is returned corresponding to each of the *FCMdata* objects passed to *HDPMixureMode.fit()*.

Visualization

By using packages like [matplotlib] it becomes easy to recreate the typical plots flow cytometry analysts are used to seeing. Convenience functions for several common plot types have been included in the *fcm.graphics* sub-package. The common pseudocolor dotplot is handled by the function *fcm.graphics.pseudocolor()*

```
1 import fcm
2 import fcm.graphics as graph
3 x = fcm.loadFCS('B6901GFJ-08_CMV pp65.fcs')
4 graph.pseudocolor(x, [('CD4 PerCP Cy55','CD8 APC Cy7')])
```

The above code produces the plot like that seen in figure 6

Another common plot is overlay histograms, which is provided by *fcm.graphics.hist()*

```
1 import fcm
2 import fcm.graphics as graph
3 from glob import glob
4 xs =[fcm.loadFCS(x) for x in glob('B6901GFJ-08_*.fcs')]
5 graph.hist(xs,3, display=True)
```

The code above will produce the histogram seen in figure 7

More examples of flow cytometry graphics can be seen in the gallery at http://packages.python.org/fcm/gallery.

Conclusion and future work

Currently *fcm* is approaching its 1.0 release, providing a stable API for development and we feel *fcm* is ready for wider usage

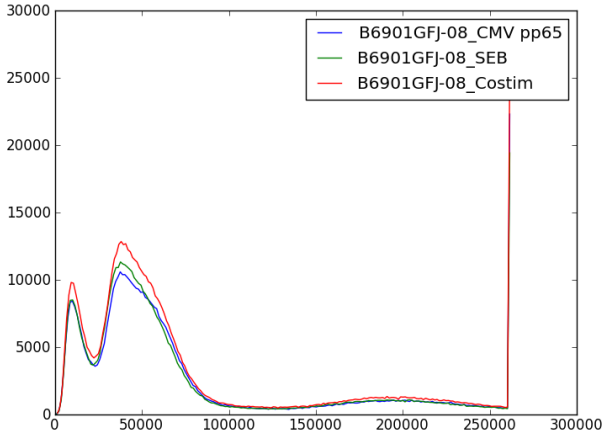


Fig. 7: *Overlay histogram of three samples from the EQAPOL data set.*

in the scientific community. Internally we use *fcm* for EDA for data sets from HIV/AIDS, caner, and solid-organ transplantation studies. In addition we have developed pipelines for batch analysis of large numbers of FCS files from the Duke Center for AIDS Research, External Quality Assurance Program Oversight Laboratory (EQAPOL), and the Association for Cancer Immunotherapy (CIMT). We have also developed a graphical tool to assist immunologist to perform model based analysis [cytostream]. Our hope is that *fcm* can fill a need in the biomedical community and facilitate the growth of python as a tool suited for scientific programming.

With the growing complexity of flow cytometry data, we foresee an increased need for computational tools. Current mass-spec based flow cytometers are capable of resolving many more parameters than current fluorescent based cytometers, necessitating improved tools for analysis. Imaging cytometers, which take digital images of events as they pass through the detection apparatus, will also produce a wealth of additional information about each event based on analyzing the images generated. These technologies will necessitate improved tools to analyze data generated by these newer cytometers. Our hope is that *fcm* can meet these needs and continue to grow to address these needs, with specific goals of developing tools to facilitate cross sample comparison and time series of flow data.

The next generation of the FCS file standard, Analytical Cytometry Standard, has been proposed, using NetCDF as the format for event storage. The ACS file will be a container allowing storage of much more than the current FCS limitations of event and textual metadata. Thanks to the availability of several good libraries for dealing with NetCDF, and the associated xml and image files proposed to be included in the ACS container, adding support for the finalized version of ACS standard should not be difficult. Gating-ML, an XML format proposed with ACS for describing gates and thier placement, has been gaining popularity. We are exploring how best to implement readers and writers for Gating-ML

Acknowledgements

We are thankful to Kent Weinhold and the Duke SORF flow core, and the statistics group led by Mike West at Duke University

for many helpful discussions. Research supported by National Institutes of Health (RC1AI086032-01, UL1RR024128 Cliburn Chan).

REFERENCES

[fcm] Frelinger J, Richards A, Chan C, http://code.google.com/p/py-fcm/

[Herzenberg2006] Herzenberg LA, Tung J et al (2006), *Interpreting flow cytometry data: a guide for the perplexed*, Nat Immunol 7(7):681-685

[Maecker2005] Maecker HT, Frey T et al (2007), *Standardization of cytokine flow cytometry assays*, BMC Immunol 6:13

[Ornatsky2006] Ornatsky O, Baranov VI et al (2006), *Multiple cellular antigent detection by ICP-MS*, J Immunol Methods 308(1-2):68-76

[ipython] Pérez F, Granger BE, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

[Parks2005] Parks, D. R., Roederer, M. and Moore, W. A. (2006), *A new "Logicle" display method avoids deceptive effects of logarithmic scaling for low signals and compensated data*, Cytometry, 69A: 541–551. doi: 10.1002/cyto.a.20258

[dpmix] Cron A, https://github.com/andrewcron/dpmix

[gpustats] Cron A and McKinney W, https://github.com/dukestats/gpustats

[matplotlib] Hunter JD, (2007), *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering 9, 90 (2007)

[cytostream] Richards A, http://code.google.com/p/cytostream/

Uncertainty Modeling with SymPy Stats

Matthew Rocklin‡*

Abstract—We add a random variable type to a mathematical modeling language. We demonstrate through examples how this is a highly separable way to introduce uncertainty and produce and query stochastic models. We motivate the use of symbolics and thin compilers in scientific computing.

Index Terms—Symbolics, mathematical modeling, uncertainty, SymPy

Introduction

Scientific computing is becoming more challenging. On the computational machinery side heterogeneity and increased parallelism are increasing the required effort to produce high performance codes. On the scientific side, computation is used for problems of increasing complexity by an increasingly broad and untrained audience. The scientific community is attempting to fill this widening need-to-ability gulf with various solutions. This paper discusses symbolic mathematical modeling.

Symbolic mathematical modeling provides an important interface layer between the *description of a problem* by domain scientists and *description of methods of solution* by computational scientists. This allows each community to develop asynchronously and facilitates code reuse.

In this paper we will discuss how a particular problem domain, uncertainty propagation, can be expressed symbolically. We do this by adding a random variable type to a popular mathematical modeling language, SymPy [Sym, Joy11]. This allows us to describe stochastic systems in a highly separable and minimally complex way.

Mathematical models are often flawed. The model itself may be overly simplified or the inputs may not be completely known. It is important to understand the extent to which the results of a model can be believed. Uncertainty propagation is the act of determining the effects of uncertain inputs on outputs. To address these concerns it is important that we characterize the uncertainty in our inputs and understand how this causes uncertainty in our results.

Motivating Example - Mathematical Modeling

We motivate this discussion with a familiar example from kinematics.

Consider an artilleryman firing a cannon down into a valley. He knows the initial position (x_0, y_0) and orientation, θ, of the cannon as well as the muzzle velocity, v, and the altitude of the target, y_f.

```
# Inputs
>>> x0 = 0
>>> y0 = 0
>>> yf = -30 # target is 30 meters below
>>> g = -10 # gravitational constant
>>> v = 30 # m/s
>>> theta = pi/4
```

If this artilleryman has a computer nearby he may write some code to evolve forward the state of the cannonball to see where it hits lands.

```
>>> while y > yf: # evolve time forward until y hits the ground
...     t += dt
...     y = y0 + v*sin(theta)*t
...            + g*t**2 / 2
>>> x = x0 + v*cos(theta)*t
```

Notice that in this solution the mathematical description of the problem $y = y_0 + v\sin(\theta)t + \frac{gt^2}{2}$ lies within the while loop. The problem and method are woven together. This makes it difficult both to reason about the problem and to easily swap out new methods of solution.

If the artilleryman also has a computer algebra system he may choose to model this problem and solve it separately.

```
>>> t = Symbol('t') # SymPy variable for time
>>> x = x0 + v * cos(theta) * t
>>> y = y0 + v * sin(theta) * t + g*t**2
>>> impact_time = solve(y - yf, t)
>>> xf = x0 + v * cos(theta) * impact_time
>>> xf.evalf() # evaluate xf numerically
65.5842

# Plot x vs. y for t in (0, impact_time)
>>> plot(x, y, (t, 0, impact_time))
```

In this case the *solve* operation is nicely separated. SymPy defaults to an analytic solver but this can be easily swapped out if analytic solutions do not exist. For example we can easily drop in a numerical binary search method if we prefer.

If he wishes to use the full power of SymPy the artilleryman may choose to solve this problem generally. He can do this simply by changing the numeric inputs to sympy symbolic variables

```
>>> x0 = Symbol('x_0')
>>> y0 = Symbol('y_0')
>>> yf = Symbol('y_f')
>>> g = Symbol('g')
>>> v = Symbol('v')
>>> theta = Symbol('theta')
```

He can then run the same modeling code found in (missing code block label) to obtain full solutions for impact_time and the final x position.

* Corresponding author: mrocklin@cs.uchicago.edu
‡ University of Chicago, Computer Science

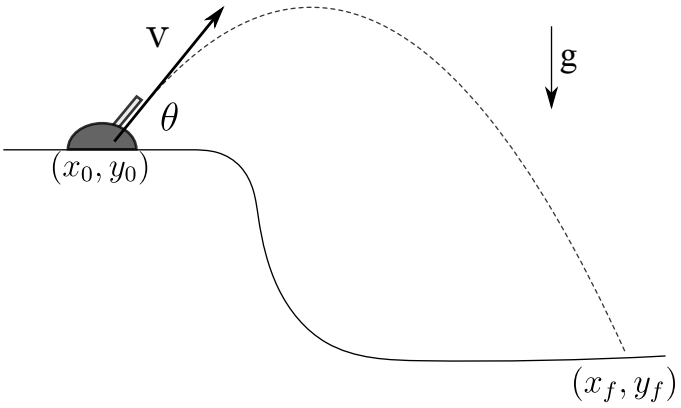


Fig. 1: *The trajectory of a cannon shot*

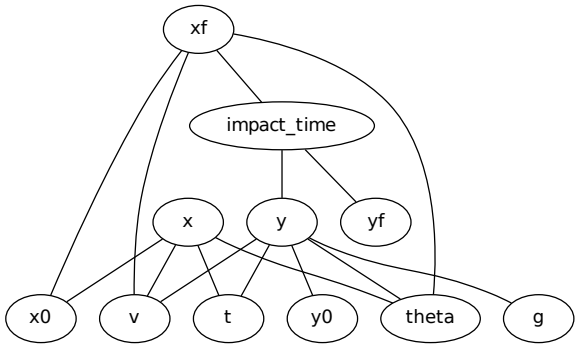


Fig. 2: *A graph of all the varibles in our system. Variables on top depend on variables connected below them. The leaves are inputs to our system.*

```
>>> impact_time
```

$$\frac{-v\sin(\theta)+\sqrt{-4gy_0+4gy_f+v^2\sin^2(\theta)}}{2g}$$

```
>>> xf
```

$$x_0+\frac{v\left(-v\sin(\theta)+\sqrt{-4gy_0+4gy_f+v^2\sin^2(\theta)}\right)\cos(\theta)}{2g}$$

Rather than produce a numeric result, SymPy produces an abstract syntax tree. This form of result is easy to reason about for both humans and computers. This allows for the manipulations which provide the above expresssions and others. For example if the artilleryman later decides he needs derivatives he can very easily perform this operation on his graph.

Motivating Example - Uncertainty Modeling

To control the velocity of the cannon ball the artilleryman introduces a certain quantity of gunpowder to the cannon. He is unable to pour exactly the desired quantity of gunpowder however and so his estimate of the velocity will be uncertain.

He models this uncertain quantity as a *random variable* that can take on a range of values, each with a certain probability. In this case he believes that the velocity is normally distributed with mean 30 and standard deviation 1.

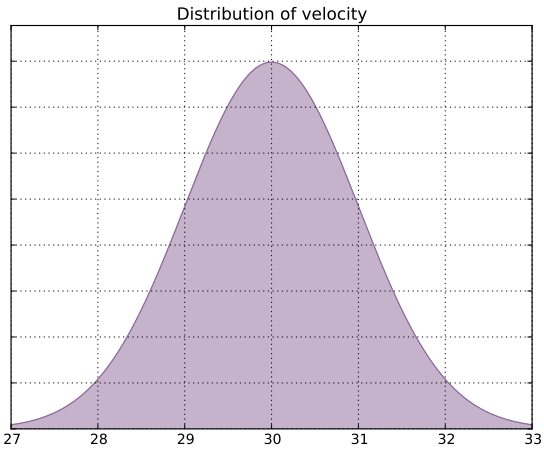


Fig. 3: *The distribution of possible velocity values*

```
>>> from sympy.stats import *
>>> v = Normal('v', 30, 1)
>>> pdf = density(v)
>>> z = Symbol('z')
>>> plot(pdf(z), (z, 27, 33))
```

$$\frac{\sqrt{2}e^{-\frac{1}{2}(z-30)^2}}{2\sqrt{\pi}}$$

`v` is now a random variable. We can query it with the following operators

```
P           -- # Probability
E           -- # Expectation
variance    -- # Variance
density     -- # Probability density function
sample      -- # A random sample
```

These convert stochasitc expressions into computational ones. For example we can ask the probability that the muzzle velocity is greater than 31.

```
>>> P(v > 31)
```

$$-\frac{1}{2}\operatorname{erf}\left(\frac{1}{2}\sqrt{2}\right)+\frac{1}{2}$$

This converts a random/stochastic expression `v > 31` into a deterministic computation. The expression `P(v > 31)` actually produces an intermediate integral expression which is solved with SymPy's integration routines.

```
>>> P(v > 31, evaluate=False)
```

$$\int_{31}^{\infty}\frac{\sqrt{2}e^{-\frac{1}{2}(z-30)^2}}{2\sqrt{\pi}}\,dz$$

Every expression in our graph that depends on `v` is now a random expression

We can ask similar questions about the these expressions. For example we can compute the probability density of the position of the ball as a function of time.

```
>>> a,b = symbols('a,b')
>>> density(x)(a) * density(y)(b)
```

$$\frac{e^{-\frac{a^2}{t^2}}e^{-\frac{\left(b+5t^2\right)^2}{t^2}}e^{30\frac{\sqrt{2}a}{t}}e^{30\frac{\sqrt{2}\left(b+5t^2\right)}{t}}}{\pi t^2 e^{900}}$$

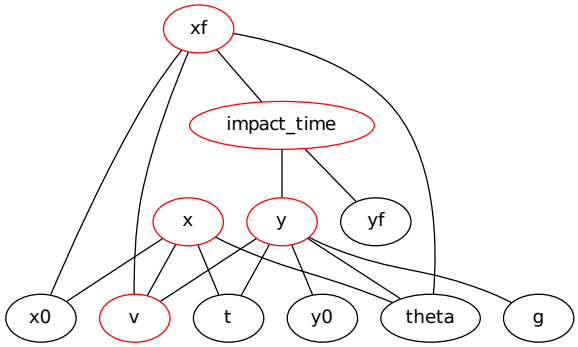
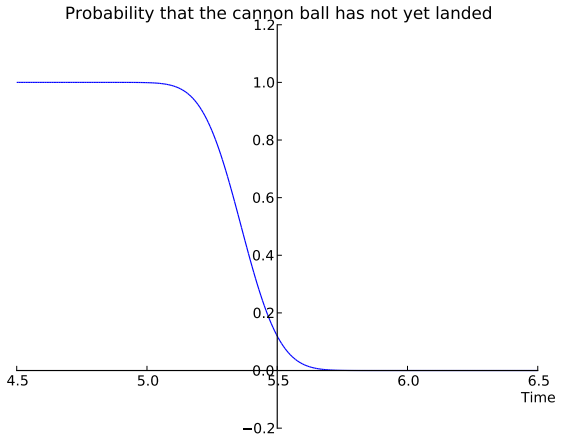


Fig. 4: *A graph of all the varibles in our system. Red variables are stochastic. Every variable that depends on the uncertain input, v, is red due to its dependence.*



Or we can plot the probability that the ball is still in the air at time t

```
>>> plot( P(y>yf), (t, 4.5, 6.5))
```

Note that to obtain these expressions the only novel work the modeler needed to do was to describe the uncertainty of the inputs. The modeling code was not touched.

We can attempt to compute more complex quantities such as the expectation and variance of `impact_time` the total time of flight.

```
>>> E(impact_time)
```

$$\int_{-\infty}^{\infty} \frac{\left(v + \sqrt{v^2 + 2400}\right) e^{-\frac{1}{2}(v-30)^2}}{40\sqrt{\pi}}\, dv$$

In this case the necessary integral proved too challenging for the SymPy integration algorithms and we are left with a correct though unresolved result.

This is an unfortunate though very common result. Mathematical models are usually far too complex to yield simple analytic solutions. I.e. this unresolved result is the common case. Fortunately computing integral expressions is a problem of very broad interest with many mature techniques. SymPy stats has successfully transformed a specialized and novel problem (uncertainty propagation) into a general and well studied one (computing integrals) to which we can apply general techniques.

RV Type	Computational Type
Continuous	SymPy Integral
Discrete - Finite (dice)	Python iterators / generators
Discrete - Infinite (Poisson)	SymPy Summation
Multivariate Normal	SymPy Matrix Expression

TABLE 1: *Different types of random expressions reduce to different computational expressions (Note: Infinite discrete and multivariate normal are in development and not yet in the main SymPy distribution)*

Sampling

One method to approximate difficult integrals is through sampling.

SymPy.stats contains a basic Monte Carlo backend which can be easily accessed with an additional keyword argument.

```
>>> E(impact_time, numsamples=10000)
5.36178452172906
```

Implementation

A `RandomSymbol` class/type and the functions `P`, `E`, `density`, `sample` are the outward-facing core of sympy.stats and the `PSpace` class in the internal core representing the mathematical concept of a probability space.

A `RandomSymbol` object behaves in every way like a standard sympy `Symbol` object. Because of this one can replace standard sympy variable declarations like

```
x = Symbol('x')
```

with code like

```
x = Normal('x', 0, 1)
```

and continue to use standard SymPy without modification.

After final expressions are formed the user can query them using the functions `P`, `E`, `density`, `sample`. These functions inspect the expression tree, draw out the `RandomSymbols` and ask these random symbols to construct a probabaility space or `PSpace` object.

The `PSpace` object contains all of the logic to turn random expressions into computational ones. There are several types of probability spaces for discrete, continuous, and multivariate distributions. Each of these generate different computational expressions.

Implementation - Bayesian Conditional Probability

SymPy.stats can also handle conditioned variables. In this section we describe how the continuous implementation of sympy.stats forms integrals using an example from data assimilation.

We measure the temperature and guess that it is about 30C with a standard deviation of 3C.

```
>>> from sympy.stats import *
>>> T = Normal('T', 30, 3) # Prior distribution
```

We then make an observation of the temperature with a thermometer. This thermometer states that it has an uncertainty of 1.5C

```
>>> noise = Normal('eta', 0, 1.5)
>>> observation = T + noise
```

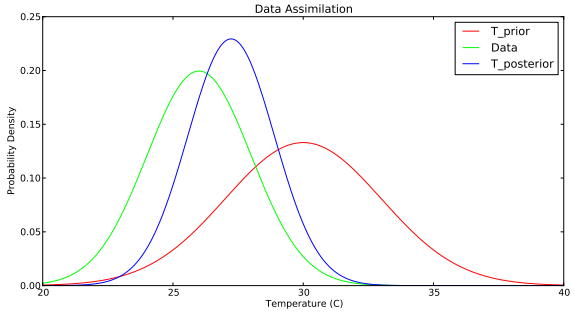


Fig. 5: *The prior, data, and posterior distributions of the temperature.*

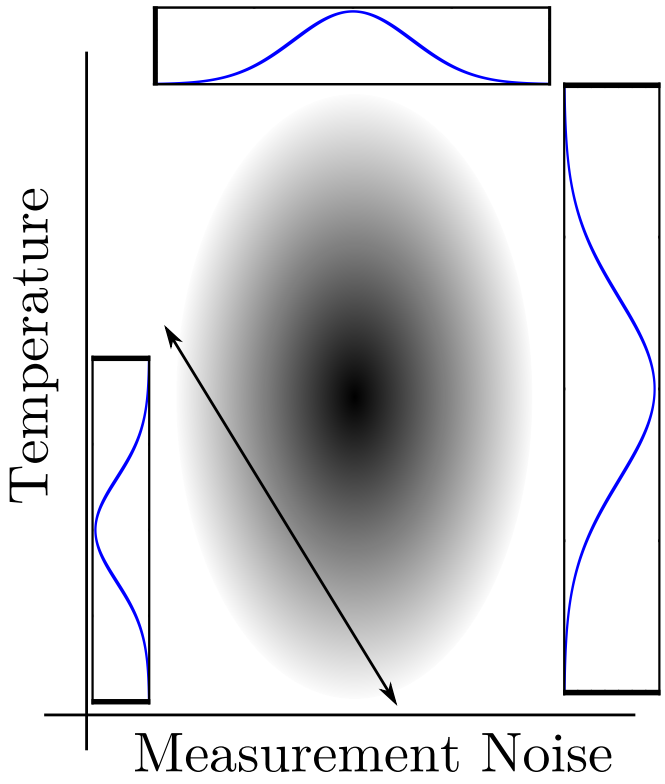


Fig. 6: *The joint prior distribution of the temperature and measurement noise. The constraint* `T + noise == 26` *(diagonal line) and the resultant posterior distribution of temperature on the left.*

With this thermometer we observe a temperature of 26C. We compute the posterior distribution that cleanly assimilates this new data into our prior understanding. And plot the three together.

```
>>> data = 26 + noise
>>> T_posterior = Given(T, Eq(observation, 26))
```

We now describe how SymPy.stats obtained this result. The expression `T_posterior` contains two random variables, `T` and `noise` each of which can independently take on different values. We plot the joint distribution below in figure 6. We represent the observation that `T + noise == 26` as a diagonal line over the domain for which this statement is true. We project the probability density on this line to the left to obtain the posterior density of the temperature.

These gemoetric operations correspond exactly to Bayesian probability. All of the operations such as restricting to the condition, projecting to the temperature axis, etc... are managed using

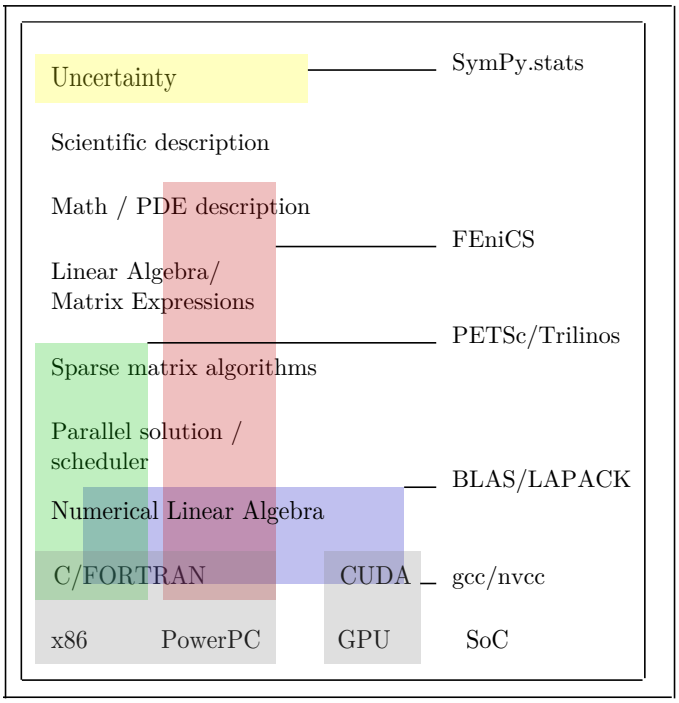


Fig. 7: *The scientific computing software stack. Various projects are displayed showing the range that they abstract. We pose that scientific computing needs more horizontal and thin layers in this image.*

core SymPy functionality.

Multi-Compilation

Scientific computing is a demanding field. Solutions frequently encompass concepts in a domain discipline (such as fluid dynamics), mathematics (such as PDEs), linear algebra, sparse matrix algorithms, parallelization/scheduling, and local low level code (C/FORTRAN/CUDA). Recently uncertainty layers are being added to this stack.

Often these solutions are implemented as single monolithic codes. This approach is challenging to accomplish, difficult to reason about after-the-fact and rarely allows for code reuse. As hardware becomes more demanding and scientific computing expands into new and less well trained fields this challenging approach fails to scale. This approach is not accessible to the average scientist.

Various solutions exist for this problem.

Low-level Languages like C provide a standard interface for a range of conventional CPUs effectively abstracting low-level architecture details away from the common programmer.

Libraries such as BLAS and LAPACK provide an interface between linear algebra and optimized low-level code. These libraries provide an interface layer for a broad range of architecture (i.e. CPU-BLAS or GPU-cuBLAS both exist).

High quality implementations of vertical slices of the stack are available through higher level libraries such as PETSc and Trilinos or through code generation solutions such as FENICS. These projects provide end to end solutions but do not provide intermediate interface layers. They also struggle to generalize well to novel hardware.

Symbolic mathematical modeling attempts to serve as a thin horizontal interface layer near the top of this stack, a relatiely empty space at present.

SymPy stats is designed to be as vertically thin as possible. For example it transforms continuous random expressions into integral expressions and then stops. It does not attempt to generate an end-to-end code. Because its backend interface layer (SymPy integrals) is simple and well defined it can be used in a plug-and-play manner with a variety of other back-end solutions.

Multivariate Normals produce Matrix Expressions

Other sympy.stats implementations generate similarly structured outputs. For example multivariate normal random variables found in `sympy.stats.mvnrv` generate matrix expressions. In the following example we describe a standard data assimilation task and view the resulting matrix expression.

```
mu = MatrixSymbol('mu', n, 1) # n by 1 mean vector
Sigma = MatrixSymbol('Sigma', n, n) # covariance matrix
X = MVNormal('X', mu, Sigma)

H = MatrixSymbol('H', k, n) # An observation operator
data = MatrixSymbol('data', k, 1)

R = MatrixSymbol('R', k, k) # covariance matrix for noise
noise = MVNormal('eta', ZeroMatrix(k, 1), R)

# Conditional density of X given  HX+noise==data
density(X , Eq(H*X+noise, data)  )
```

$$\mu = \begin{bmatrix}\mathbb{I} & \mathbf{0}\end{bmatrix}\left(\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\right)^{-1}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\mu\\ \mathbf{0}\end{bmatrix} - data\right) + \begin{bmatrix}\mu\\ \mathbf{0}\end{bmatrix}\right)$$

$$\Sigma = \begin{bmatrix}\mathbb{I} & \mathbf{0}\end{bmatrix}\left(\mathbb{I} - \begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\right)^{-1}\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\right)\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}\mathbb{I}\\ \mathbf{0}\end{bmatrix}$$

$$\mu = \mu + \Sigma H^T\left(R + H\Sigma H^T\right)^{-1}(H\mu - data)$$

$$\Sigma = \left(\mathbb{I} - \Sigma H^T\left(R + H\Sigma H^T\right)^{-1}H\right)\Sigma$$

Those familiar with data assimilation will recognize the Kalman Filter. This expression can now be passed as an input to other symbolic/numeric projects. Symbolic/numerical linear algebra is a vibrant and rapidly changing field. Because `sympy.stats` offers a clean interface layer it is able to easily engage with these developments. Matrix expressions form a clean interface layer in which uncertainty problems can be expressed and transferred to computational systems.

We generally support the idea of approaching the scientific computing conceptual stack (Physics/PDEs/Linear-algebra/MPI/C-FORTRAN-CUDA) with a sequence of simple and atomic compilers. The idea of using interface layers to break up a complex problem is not new but is oddly infrequent in scientific computing and thus warrants mention. It should be noted that for heroic computations this approach falls short - maximal speedup often requires optimizing the whole problem at once.

Conclusion

We have foremost demonstrated the use of `sympy.stats` a module that enhances `sympy` with a random variable type. We have shown how this module allows mathematical modellers to describe the undertainty of their inputs and compute the uncertainty of their outputs with simple and non-intrusive changes to their symbolic code.

Secondarily we have motivated the use of symbolics in computation and argued for a more separable computational stack within the scientific computing domain.

REFERENCES

[Sym] SymPy Development Team (2012). SymPy: Python library for symbolic mathematics URL http://www.sympy.org.

[Roc12] M. Rocklin, A. Terrel, *Symbolic Statistics with SymPy* Computing in Science & Engineering, June 2012

[Joy11] D. Joyner, O. Čertík, A. Meurer, B. Granger, *Open source computer algebra systems: SymPy* ACM Communications in Computer Algebra, Vol 45 December 2011

QuTiP: A framework for the dynamics of open quantum systems using SciPy and Cython

Robert J. Johansson‡*, Paul D. Nation‡

Abstract—We present QuTiP (http://www.qutip.org), an object-oriented, open-source framework for solving the dynamics of open quantum systems. Written in Python, and using a combination of Cython, NumPy, SciPy, and matplotlib, QuTiP provides an environment for computational quantum mechanics that is both easy and efficient to use. Arbitrary quantum systems, including time-dependent systems, may be built up from operators and states defined by a quantum object class, and then passed on to a choice of unitary or dissipative evolution solvers. Here we give an overview of the basic structure for the framework, and the techniques used in its implementation. We also present a few selected examples from current research on quantum mechanics that illustrate the strengths of the framework, as well as the types of calculations that can be performed. This framework is particularly well suited to the fields of quantum optics, superconducting circuit devices, nanomechanics, and trapped ions, while also being ideal as an educational tool.

Index Terms—quantum mechanics, master equation, monte-carlo

Introduction

One of the main goals of contemporary physics is to control the dynamics of *individual* quantum systems. Starting with trapped-ion experiments in the 1970s [Hor97], the ability to manipulate single realizations, as opposed to ensembles, of quantum systems allows for fundamental tests of quantum mechanics [Har06] and quantum computation [Lad10]. Traditionally, the realm of quantum mechanics has been confined to systems with atomic and molecular characteristic length scales with system properties fixed by nature. However, during the last two decades, advances in experimental and manufacturing techniques have opened up the possibility of producing man-made micro and nanometer scale devices with controllable parameters that are governed by the laws of quantum mechanics. These engineered systems can now be realized with a wide range of different technologies such as quantum optics [Obr09], superconducting circuits [You11], semiconducting quantum dots [Han08], nanomechanical devices [Oco10], and ion traps [Bla12], and have received considerable experimental and theoretical attention.

With the increasing interest in engineered quantum devices, the demand for efficient numerical simulations of open quantum dynamics has never been greater. By definition, an open quantum system is coupled to an environment, also called a reservoir or bath, where the complexity of the environmental dynamics renders the combined evolution of system plus reservoir analytically intractable and must therefore be simulated numerically. With a quantum computer out of reach for the foreseeable future, these simulations must be performed using classical computing techniques, where the exponentially increasing dimensionality of the underlying Hilbert space severely limits the size of system that can be efficiently simulated [Fey82]. Fortunately, recent experimental advances have lead to quantum systems fabricated from a small number of oscillator and spin components, containing only a few quanta, that lend themselves to simulation in a truncated state space.

Here we introduce QuTiP [Joh12], a framework designed for simulating open quantum systems using SciPy and Cython. Although other software solutions exist [Sch97], [Vuk07], [Tan99], QuTiP goes beyond these earlier tools by providing a completely open-source solution with an easy to read syntax and extended

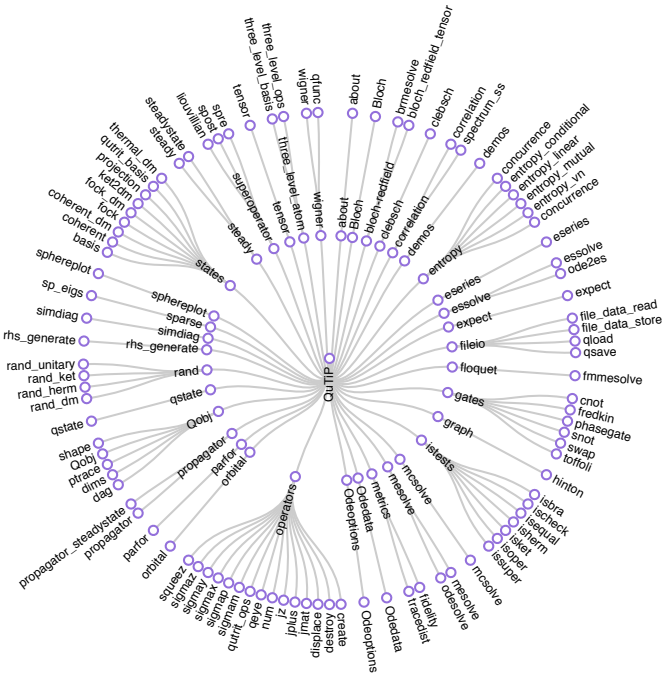


Fig. 1: The organization of user available functions in the QuTiP framework. The inner circle represents submodules, while the outer circle gives the public functions and classes contained in each submodule.

* *Corresponding author: robert@riken.jp*
‡ *Advanced Science Institute, RIKEN, Wako-shi, 351-0198 Japan*

functionality, such as built-in multiprocessing. Our objective with QuTiP is to provide a thoroughly tested and well documented generic framework that can be used for a diverse set of quantum mechanical problems, that encourages openness, verifiability, and reproducibility of published results in the computational quantum mechanics community.

Numerical quantum mechanics

In quantum mechanics, the state of a system is represented by the wavefunction Ψ, a probability amplitude that describes, for example, the position and momentum of a particle. The wavefunction is in general a function of space and time, and its evolution is ideally governed by the Schrödinger equation, $-i\partial_t\Psi = \hat{H}\Psi$, where $\hat{H}$ is the Hamiltonian that describes the energies of the possible states of the system (total energy function). In general, the Schrödinger equation is a linear partial differential equation. For computational purposes, however, it is useful to expand the wavefunction, Hamiltonian, and thus the equation of motion, in terms of basis functions that span the state space (Hilbert space), and thereby obtain a matrix and vector representation of the system. Such a representation is not always feasible, but for many physically relevant systems it can be an effective approach when used together with a suitable truncation of the basis states that often are infinite. In particular, systems that lend themselves to this approach includes resonator modes and systems that are well characterized by a few quantum states (e.g., the two quantum levels of an electron spin). These components also represent the fundamental building blocks of engineered quantum devices.

In the matrix representation the Schrödinger equation can be written as

$$-i\frac{d}{dt}|\psi\rangle = H(t)|\psi\rangle, \tag{1}$$

where $|\psi\rangle$ is a state vector and H is the Hamiltonian matrix. Note that the introduction of complex values in (1) is a fundamental property of evolution in quantum mechanics. In this representation, the equations of motion are a system of ordinary differential equations (ODEs) in matrix form with, in general, time-dependent coefficients. Therefore, to simulate the dynamics of a quantum system we need to obtain the matrix representation of the Hamiltonian and the initial state vector in the chosen basis. Once this is achieved, we have a numerically tractable problem, that involves solving systems of coupled ODEs defined by complex-valued matrices and state vectors.

The main challenge in numerical simulation of quantum systems is that the required number basis states, and thus the size of the matrices and vectors involved in the numerical calculations, quickly become excessively large as the size of the system under consideration is increased. In a composite system, the state space increases exponentially with the number of components. Therefore, in practice only relatively small quantum systems can be simulated on classical computers with reasonable efficiency. Fortunately, in essentially all experimentally realizable systems, the local nature of the physical interactions gives rise to system Hamiltonians containing only a few nonzero elements that are thus highly sparse. This sparsity plays a fundamental role in the efficiency of simulations on quantum computers as well [Aha03]. The exact number of states that can be managed depends on the detailed nature of the problem at hand, but the upper limit is typically on the order of a few thousand quantum states. Many experimentally relevant systems fall within this limit, and numerical simulations of quantum systems on classical computers is therefore an important subject.

Although the state of an ideal quantum systems is completely defined by the wavefunction, or the corresponding state vector, for realistic systems we also need to describe situations where the true quantum state of a system is not fully known. In such cases, the state is represented as a statistical mixture of state vectors $|\psi_n\rangle$, that can conveniently be expressed as a state (density) matrix $\rho = \sum_n p_n |\psi_n\rangle\langle\psi_n|$, where p_n is the classical probability that the system is in the state $|\psi_n\rangle$. The need for density matrices, instead of wavefunctions, arises in particular when modeling open quantum system, where the system's interaction with its surrounding is included. In contrast to the Schrödinger equation for closed quantum systems, the equation of motion for open systems is not unique, and there exists a large number of different equations of motion (e.g., Master equations) that are suitable for different situations and conditions. In QuTiP, we have implemented many of the most common equations of motion for open quantum systems, and provide a framework that can be extended easily when necessary.

The QuTiP framework

As a complete framework for computational quantum mechanics, QuTiP facilitates automated matrix representations of states and operators (i.e. to construct Hamiltonians), state evolution for closed and open quantum systems, and a large library of common utility functions and operations. For example, some of the core functions that QuTiP provides are: `tensor` for constructing composite states and operators from its fundamental components, `ptrace` for decomposing states into their components, `expect` for calculating expectation values of measurement outcomes for an operator and a given state, an extensive collection of functions for generating frequently used states and operators, as well as additional functions for entanglement measures, entropy measures, correlations and much more. A visual map of the user-accessible functions in QuTiP is shown in Fig. 1. For a complete list of functions and their usage, see the QuTiP user guide [Nat12].

The framework is designed so that its syntax and procedures mirror, as closely as possible, the standard mathematical formulation of a quantum mechanical problem. This is achieved thanks to the Python language syntax, and an object-oriented design that is centered around the class `Qobj`, used for representing quantum objects such as states and operators.

In order to simulate the quantum evolution of an arbitrary system, we need an object that not only incorporates both states and operators, but that also keeps track of important properties for these objects, such as the composite structure (if any) and the Hermicity. This later property is especially important as all physical observables are Hermitian, and this dictates when real values should be returned by functions corresponding to measurable quantities. In QuTiP, the complete information for any quantum object is included in the `Qobj` class. This class is the fundamental data structure in QuTiP. As shown in Fig. 2, the `Qobj` object can be thought of as a container for the necessary properties need to completely characterize a given quantum object, along with a collection of methods that act on this operator alone.

A typical simulation in QuTiP takes the following steps:

- Specify system parameters and construct Hamiltonian, initial state, and any dissipative quantum (`Qobj`) objects.

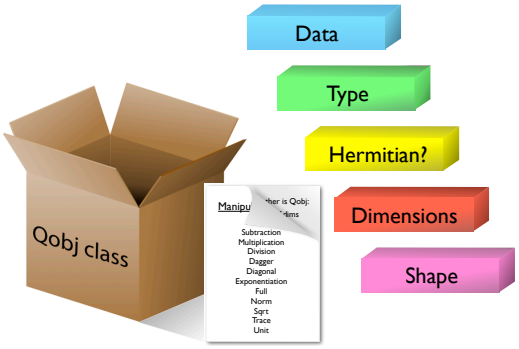


Fig. 2: *`Qobj` class used for defining quantum objects. The class properties include the sparse matrix representation of the object (data), the type of object represented, a nested list describing the composite structure (dimensions), whether the object is Hermitian, and the shape of the underlying data matrix. Also included is a lengthy set of operations acting on the `Qobj`, a list of which can be found at [Nat12].*

- Calculate the evolution of the state vector, or density matrix, using the system Hamiltonian in the appropriate solver.
- Post-process output `Qobj` and/or arrays of return values, including visualization.

Given the generality of this process, we highlight each of these steps below by demonstrating the setup and simulation of select real-world examples.

Constructing Hamiltonians and states

The first step in any QuTiP simulation is the creation of the Hamiltonian that describes the system of interest, initial state, and any possible operators that characterize the interaction between the system and its environment. Although it is possible to directly input a system Hamiltonian into a `Qobj` class object, QuTiP includes a number of predefined operators for oscillator and spin systems out of which a large collection of Hamiltonians can be composed. The simplest, and most common, example is the so-called Jaynes-Cummings model for a two-level atom (qubit) interacting with a single harmonic oscillator [Jay63]

$$\hat{H} = \hbar\omega_c \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{\sigma}_z/2 + \hbar g/2\left(\hat{a}\hat{\sigma}_+ + \hat{a}^\dagger \hat{\sigma}_-\right) \qquad (2)$$

where the first term in (2) describes the oscillator in terms of creation operators, the second gives the bare qubit Hamiltonian, and the final term characterizes the interaction between oscillator and qubit. Here, ω_c is the oscillator frequency, ω_q is the qubit energy splitting frequency, and g gives the strength of the oscillator-qubit coupling. Typically one is interested in the exchange of a single excitation between the qubit and oscillator. Although the oscillator has an infinite number of states, in this case, we can truncate the Hilbert space. For the initial state with the excitation in the qubit, this state may be written in QuTiP as (we omit the `from qutip import *` statement):

```
N = 4 # number of oscillator levels to consider
psi_osc = basis(N)
psi_qubit = basis(2,1)
psi_sys = tensor(psi_osc,psi_qubit)
```

where `basis(N,m)` creates a basis function of size `N` with a single excitation in the `m` level, and the `tensor` function creates the composite initial state from the individual state vectors for the oscillator and qubit subsystems. The total Hamiltonian (2) can be created in a similar manner using built-in operators and user defined system parameters:

```
wc = wq = 1.0
g = 0.1
a = tensor(destroy(N),qeye(2))
sz = tensor(qeye(N),sigmaz())
sp = tensor(qeye(N),sigmap())
sm = tensor(qeye(N),sigmam())
H = wc*a.dag()*a + wq/2.*sz + g/2.*(a*sp+a.dag()*sm)
```

This final Hamiltonian is a `Qobj` class object representing the Jaynes-Cummings model and is created with a syntax that closely resembles the mathematical formulation given in Eq. (2). Using the `print` function, we can list all of the properties of `H` (omitting the underlying data matrix):

```
Quantum object: dims = [[4, 2], [4, 2]],
shape = [8, 8], type = oper, isHerm = True
```

showing the composite (4×2) structure, the type of object, and verifying that indeed the Hamiltonian is Hermitian as required. Having created collapse operators, if any, we are now in a position to pass the Hamiltonian and initial state into the QuTiP evolution solvers.

Time-evolution of quantum systems

The time-evolution of an initial state of a closed quantum system is completely determined by its Hamiltonian. The evolution of an open quantum system, however, additionally depends on the environment surrounding the system. In general, the influence of such an environment cannot be accounted for in detail, and one need to resort to approximations to arrive at a useful equation of motion. Various approaches to this procedure exist, which results in different equations of motion, each suitable for certain situations. However, most equations of motion for open quantum systems can be characterized with the concept of collapse operators, which describe the effect of the environment on the system and the rate of those processes. A complete discussion of dissipative quantum systems, which is outside the scope of this paper, can be found in [Joh12] and references therein.

QuTiP provides implementations of the most common equations of motion for open quantum systems, including the Lindblad master equation (`mesolve`), the Monte-Carlo quantum trajectory method (`mcsolve`), and certain forms of the Bloch-Redfield (`brmesolve`) and Floquet-Markov (`fmmesolve`) master equations. In QuTiP, the basic type signature and the return value are the same for all evolution solvers. The solvers take following parameters: a Hamiltonian `H`, an initial state `psi_sys`, a list of times `tlist`, an optional list of collapse operators `c_ops` and an optional list of operators for which to evaluate expectation values. For example,

```
c_ops = [sqrt(0.05) * a]
expt_ops = [sz, a.dag() * a]
tlist = linspace(0, 10, 100)
out = mesolve(H, psi_sys, tlist, c_ops, expt_ops)
```

Each solver returns (`out`) an instance of the class `Odedata` that contains all of the information about the solution to the problem, including the requested expectation values, in `out.expect`. The evolution of a closed quantum system can also be computed using the `mesolve` or `mcsolve` solvers, by passing an empty list in place of the collapse operators in the fourth argument. On top of this shared interface, each solver has a set of optional function parameters and class members in `Odedata`, allowing

for modification of the underlying ODE solver parameters when necessary.

Visualization

In addition to providing a computational framework, QuTiP also implements of a number of visualization methods often employed in quantum mechanics. It is of particular interest to visualize the state of a quantum system. Quantum states are often complex superpositions of various basis states, and there is an important distinction between pure quantum coherent superpositions and statistical mixtures of quantum states. Furthermore, the set of all quantum states also includes the classical states, and it is therefore of great interest to visualize states in ways that emphasize the differences between classical and quantum states. Such properties are not usually apparent by inspecting the numerical values of the state vector or density matrix, thus making quantum state visualization techniques an important tool.

Bloch sphere

A quantum two-level system (qubit), can not only occupy the two classical basis states, e.g., "0" and "1", but an arbitrary complex-valued superposition of those two basis states. Such states can conveniently be mapped to, and visualized as, points on a unit sphere, commonly referred to as the Bloch sphere. QuTiP provides a class `Bloch` for visualizing individual quantum states, or lists of data points, on the Bloch sphere. Internally it uses matplotlib to render a 3D view of the sphere and the data points. The following code illustrates how the `Bloch` class can be used

```
bs = Bloch()
bs.add_points([x, y, z])
bs.show()
```

where `x`, `y`, and `z` are the expectation values for the operators σ_x, σ_y, and σ_z, respectively, for the given states. The expectation values can be obtained from the `Odedata` instance returned by a time-evolution solver, or calculated explicitly for a particular state, for example

```
psi = (basis(2,0) + basis(2,1)).unit()
op_axes = sigmax(), sigmay(), sigmaz()
x, y, z = [expect(op, psi) for op in op_axes]
```

In Fig. 5, the time-evolution of a two-level system is visualized on a Bloch sphere using the `Bloch` class.

Quasi-probability distributions

One of goals in engineered quantum systems is to manipulate the system of interest into a given quantum state. Generating quantum states is a non-trivial task as classical driving fields typically lead to classical system states, and the environment gives rise to noise sources that destroy the delicate quantum superpositions and cause unwanted dissipation. Therefore, it is of interest to determine whether the state of the system at a certain time is in a non-classical state. One way to verify that the state of a system is indeed quantum mechanical is to visualize the state of the system as a Wigner quasi-probability distribution. This Wigner function is one of several quasi-probability distributions that are linear transformations of the density matrix, and thus give a complete characterization of the state of the system [Leh10]. The Wigner function is of particular interest since any negative Wigner values indicate an inherently quantum state. Here we demonstrate the ease of calculating Wigner functions in QuTiP by visualizing

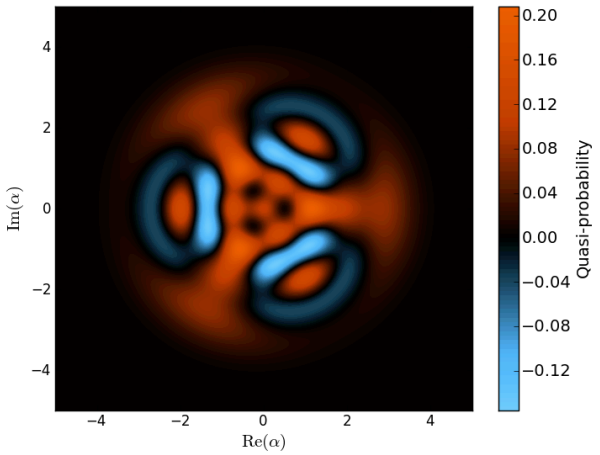


Fig. 3: *Wigner function for the state* $|\Psi\rangle = \frac{1}{\sqrt{3}}[|0\rangle + |3\rangle + |6\rangle]$ *as reconstructed experimentally in [Hof09] . Negative (blue) values indicate that this state is inherently quantum mechanical. The x- and y-axes represent the oscillator position and momentum, respectively.*

the quantum oscillator state $|\Psi\rangle = \frac{1}{\sqrt{3}}[|0\rangle + |3\rangle + |6\rangle]$ recently generated in a superconducting circuit device [Hof09]:

```
psi = (basis(10)+basis(10,3)+basis(10,6)).unit()
xvec = linspace(-5,5,250)
X,Y = meshgrid(xvec, xvec)
W = wigner(psi,xvec,xvec)
```

Again, the quantum state is written in much the same manner as the corresponding mathematical expression with the `basis` functions representing the Fock states $|0\rangle$, $|3\rangle$, and $|6\rangle$ in a truncated Hilbert space with $N = 10$ levels. Here, the `unit` method of the `Qobj` class automatically normalizes the state vector. The `wigner` then takes this state vector (or a density matrix) and generates the Wigner function over the requested interval. The result in shown in Fig. 3.

Example: Multiple Landau-Zener transitions

To demonstrate additional features in QuTiP, we now consider a quantum two-level system, with static tunneling rate Δ and energy-splitting ε, that is subject to a strong driving field of amplitude A coupled to the σ_z operator. In recent years, this kind of system has been actively studied experimentally [Oli05], [Sil06], [Ste12] for its applications in amplitude spectroscopy and Mach-Zehnder interferometry. The system is described by the Hamiltonian

$$\hat{H} = -\frac{\Delta}{2}\hat{\sigma}_x - \frac{\varepsilon}{2}\hat{\sigma}_z - \frac{A}{2}\cos(\omega t)\hat{\sigma}_z, \tag{3}$$

and the initial state $|\psi(t=0)\rangle = |0\rangle$. This is a time-dependent problem, and we cannot represent the Hamiltonian with a single `Qobj` instance. Instead, we can use a nested list of `Qobj` instances and their time-dependent coefficients. In this notation (referred to as list-string notation in QuTiP), the Hamiltonian in Eq. 3 can be defined as

```
H0 = -delta/2 * sigmax() - epsilon/2 * sigmaz()
H1 = sigmaz()
H_td = [H0, [H1, 'A/2 * cos(omega * t)']]
args = {'omega': omega, 'A': A}
```

The QuTiP time-evolution solvers, as well as other functions that use time-dependent operators, then know how to evaluate

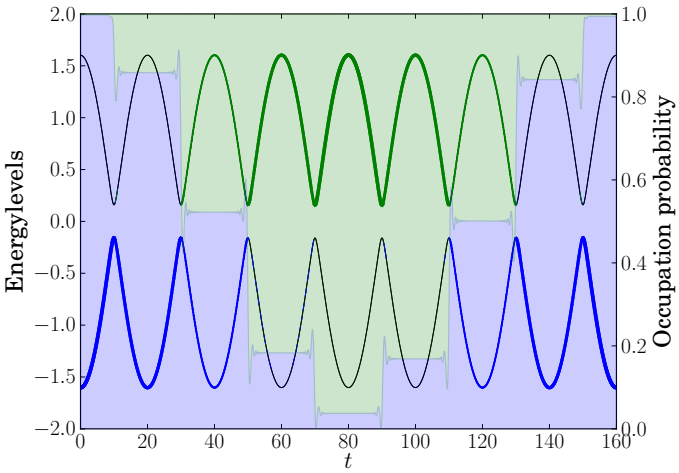


Fig. 4: *Repeated Landau-Zener-like transitions in a quantum two-level system. In each successive sweep through the avoided-level crossing, a small additative change in the occupation probability occurs, and after many crossings a nearly complete state transfer has been achieved. This is an example of constructive interference.*

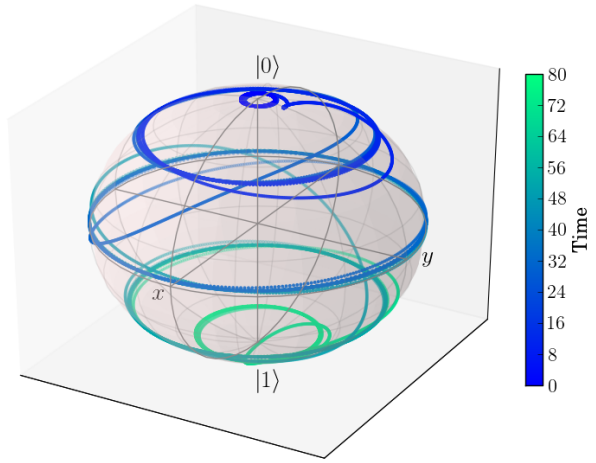


Fig. 5: *Bloch-sphere visualization of the dynamics of a quantum two-level system subject to repeated Landau-Zener-like avoided-level crossings. All the points lay on the surface of the Bloch sphere, so we can immediately conclude that the dynamics is the unitary evolution of a closed quantum system (we did not include any collapse operators in this example).*

the nested list `H_td` to the appropriate operator expression. In this list-string format, this nested list is converted into a Cython source file and compiled. Here, the dictionary `args` is used for passing values of variables that occur in the expression for the time-dependent coefficients. Given this QuTiP representation of the Hamiltonian 3, we can evolve an initial state, using for example the Lindblad master equation solver, with the following lines of code:

```
psi0 = basis(2,0)
tlist = linspace(0, 160, 500)
output = mesolve(H_td, psi0, tlist, [], [], args)
```

Note that here we passed empty lists as fourth and and fifth arguments to the solver `mesolve`, that indicates that we do not have any collapse operators (that is, a closed quantum system) and we do not request any expectation values to be calculated directly by the solver. Instead, we will obtain a list `output.states` that contains the state vectors for each time specified in `tlist`.

These states vectors can be used in further calculations, or for example to visualizing the occupation probabilities of the two states, as show in Figs. 4 and 5. In Fig. 5 we used the previously discussed `Bloch` class to visualize the trajectory of the two-level system.

Implementation and optimization techniques

In implementing the QuTiP framework, we have relied heavily on the excellent Scipy and Numpy packages for Python. Internally, in the class for representing quantum objects, `Qobj`, and in the various time-evolution solvers, we use the sparse matrix from Scipy (in particular the compressed-row format), and in some special cases dense Numpy arrays, for the matrix and vector representation quantum operators and states. Most common quantum mechanics operations can be mapped to the linear algebra operations that are implemented in Scipy for sparse matrices, including matrix-matrix and matrix-vector multiplication, outer and inner products of matrices and vectors, and eigenvalue/eigenvector decomposition. Additional operations that do not have a direct correspondence in matrix algebra, such as the `ptrace` function for decomposing composite states, have been implemented mostly in Python and NumPy. Note that in quantum mechanics it is essential that all matrix and vector elements are complex numbers, and Scipy's thorough support for complex-valued sparse matrices has been a fundamental prerequisite for using Scipy in QuTiP. Overall, Scipy's sparse matrices, and the corresponding functions, have delivered excellent performance. However, we have found that by replacing the built-in matrix-vector multiplication in selected locations with a less general Cython implementation (without, for example type and out-of-bounds checks) we can obtain additional speed-ups.

The ordinary differential equation solver is another feature in Scipy that is used extensively in QuTiP, as most time-evolution solvers use the `scipy.integrate.ode` interface at some level. The configurability and flexibility of Scipy's ODE solver has significantly simplified the implementation of many time-evolution solvers in QuTiP. The Monte-Carlo solver in particular, which is a hybrid method that mixes evolution according to an ODE with stochastic processes, uses some of the more advanced modes of operating Scipy's ODE solver including the high level of control of step size, selectively stopping and restarting the solver, etc.

In a typical simulation using QuTiP, the vast majority of the elapsed time is devoted to evolving ODEs. Fine-tuning Scipy's ODE solver and ensuring that we obtain optimal performance from it has therefore been a priority. Among the optimization measures we have used, the largest impact has been gained by implementing the callback function for the right-hand side (RHS) of the ODE in standard form using Cython. By doing so, a significant amount of overhead related to Python function calls can be avoided, and with the additional speed-up that is gained by evaluating the callback using Cython, this technique has given speed-up factors of up to an order of magnitude or greater [Joh12]. Given this level of speed-up, for any computational problem using Scipy's ODE solver, we would recommend investigating if the callback function

can be implemented in Cython as one of the first performance optimization measures.

One complicating factor that prevents using static Cython implementations for the RHS function with Scipy's ODE, is that in QuTiP the ODEs are generated dynamically by the QuTiP framework. For time-independent problems the RHS function for the ODEs reduce to matrix-vector multiplication, and can be delegated to a pre-compiled Cython function, but in a general time-dependent problem this is not possible. To circumvent this problem, we have employed a method of dynamically generating, compiling and loading Cython code for the RHS callback function. This approach allows us to benefit from the speed-ups gained with a Cython implementation with nontrivial time-dependent RHS functions.

Finally, in implementing QuTiP we have used the Python `multiprocessing` package to parallellize of many time-consuming tasks using the QuTiP `parfor` function, ensuring efficient use of the resources commonly available on modern multicore systems. The Monte-Carlo solver, which requires the evolution of many hundreds of independent ODE systems, is particularly easy to parallelize and has benefited greatly from the `multiprocessing` package, and its good scaling properties as a function of the number of CPU cores.

Conclusions

The Python, Numpy/Scipy and matplotlib environment provides and encourages a unique combination of intuitive syntax and good coding practices, rapid code development, good performance, tight integration between the code and its documentation and testing. This has been invaluable for the QuTiP project. With the additional selective optimization using Cython, QuTiP delivers performance that matches and in many cases exceeds those of natively compiled alternatives [Tan99], accessible through an easy to use environment with a low learning curve for quantum physicists. As a result, sophisticated quantum systems and models can be programmed easily and simulated efficiently using QuTiP.

Acknowledgements

We would like to thank all of the contributors who helped test and debug QuTiP. RJJ and PDN were supported by Japanese Society for the Promotion of Science (JSPS) Fellowships P11505 and P11202, respectively. Additional support comes from Kakenhi grant Nos. 2302505 (RJJ) and 2301202 (PDN).

References

[Aha03] D. Aharonov and A. Ta-Shma, *Adiabatic quantum state generation and statistical zero knowledge*, ACM Symposium on Theory of Computing 20, 2003, available at quant-ph/0301023.

[Bla12] R. Blatt and C. F. Roos, *Quantum simulations with trapped ions*, Nat. Physics, 8:277, 2012.

[Fey82] R. Feynman, *Simulating Physics with Computers*, Int. J. Theor. Phys., 21(6):467, 1982.

[Han08] R. Hanson and D. D. Awschalom, *Coherent manipulation of single spins in semiconductors*, Nature, 453:1043, 2008.

[Har06] S. Haroche and J-M. Raimond, *Exploring the Quantum: Atoms, Cavities, and Photons*, Oxford University Press, 2006.

[Hof09] M. Hofheinz et al., *Synthesizing arbitrary quantum states in a superconducting resonator*, Nature, 459:546, 2009.

[Hor97] G. Z. K. Horvath et al., *Fundamental physics with trapped ions*, Contemp. Phys., 38:25, 1997.

[Jay63] E. T. Jaynes and F. W. Cummings, *Comparison of quantum and semiclassical radiation theories with application to the beam maser*, Proc. IEEE 51(1):89 (1963).

[Joh12] J. R. Johansson et al., *QuTiP: An open-source Python framework for the dynamics of open quantum systems*, Comp. Phys. Commun., 183:1760, 2012, available at arXiv:1110.0573.

[Lad10] T. D. Ladd et al., *Quantum computers*, Nature, 464:45, 2010.

[Leh10] U. Leonhardt, *Essential Quantum Optics*, Cambridge, 2010.

[Nat12] P. D. Nation and J. R. Johansson, *QuTiP: Quantum Toolbox in Python*, Release 2.0, 2012, available at www.qutip.org.

[Obr09] J. L. O'Brien et al., *Photonic quantum technologies*, Nat. Photonics, 3:687, 2009.

[Oco10] A. D. O'Connell et al., *Quantum ground state and single-phonon control of a mechanical resonator*, Nature, 464:697, 2010.

[Sch97] R. Schack and T. A. Brun, *A C++ library using quantum trajectories to solve quantum master equations*, Comp. Phys. Commun., 102:210, 1997.

[Tan99] S. M. Tan, *A computational toolbox for quantum and atomic optics*, J. Opt. B: Quantum Semiclass. Opt., 1(4):424, 1999.

[Vuk07] A. Vukics and H. Ritsch, *C++QED: an object-oriented framework for wave-function simulations of cavity QED systems*, Eur. Phys. J. D, 44:585, 2007.

[You11] J. Q. You and F. Nori, *Atomic Physics and Quantum Optics Using Superconducting Circuits*, Nature, 474:589, 2011.

[Oli05] W. D. Oliver et al., *Mach-Zehnder Interferometry in a Strongly Driven Superconducting Qubit*, Science. 310:1653, 2005.

[Sil06] M. Sillanpää et al., *Continuous-Time Monitoring of Landau-Zener Interference in a Cooper-Pair Box*, Phys. Rev. Lett., 96:187002, 2006.

[Ste12] J. Stehlik et al., *Landau-Zener-Stuckelberg Interferometry of a Single Electron Charge Qubit*, ArXiv:1205.6173, 2012.

cphVB: A System for Automated Runtime Optimization and Parallelization of Vectorized Applications

Mads Ruben Burgdorff Kristensen‡*, Simon Andreas Frimann Lund‡, Troels Blum‡, Brian Vinter‡

Abstract—Modern processor architectures, in addition to having still more cores, also require still more consideration to memory-layout in order to run at full capacity. The usefulness of most languages is deprecating as their abstractions, structures or objects are hard to map onto modern processor architectures efficiently.

The work in this paper introduces a new abstract machine framework, cphVB, that enables vector oriented high-level programming languages to map onto a broad range of architectures efficiently. The idea is to close the gap between high-level languages and hardware optimized low-level implementations. By translating high-level vector operations into an intermediate vector bytecode, cphVB enables specialized vector engines to efficiently execute the vector operations.

The primary success parameters are to maintain a complete abstraction from low-level details and to provide efficient code execution across different, modern, processors. We evaluate the presented design through a setup that targets multi-core CPU architectures. We evaluate the performance of the implementation using Python implementations of well-known algorithms: a jacobi solver, a kNN search, a shallow water simulation and a synthetic stencil simulation. All demonstrate good performance.

Index Terms—runtime optimization, high-performance, high-productivity

Introduction

Obtaining high performance from today's computing environments requires both a deep and broad working knowledge on computer architecture, communication paradigms and programming interfaces. Today's computing environments are highly heterogeneous consisting of a mixture of CPUs, GPUs, FPGAs and DSPs orchestrated in a wealth of architectures and lastly connected in numerous ways.

Utilizing this broad range of architectures manually requires programming specialists and is a very time-consuming task – time and specialization a scientific researcher typically does not have. A high-productivity language that allows rapid prototyping and still enables efficient utilization of a broad range of architectures is clearly preferable. There exist high-productivity language and libraries that automatically utilize parallel architectures [Kri10], [Dav04], [New11]. They are however still few in numbers and have one problem in common. They are closely coupled to both the front-end, i.e. programming language and IDE, and the back-end, i.e. computing device, which makes them interesting only to the few using the exact combination of front and back-end.

A tight coupling between front-end technology and back-end presents another problem; the usefulness of the developed program expires as soon as the back-end does. With the rapid development of hardware architectures the time spend on implementing optimized programs for specific hardware, is lost as soon as the hardware product expires.

In this paper, we present a novel approach to the problem of closing the gap between high-productivity languages and parallel architectures, which allows a high degree of modularity and reusability. The approach involves creating a framework, cphVB[*] (Copenhagen Vector Bytecode). cphVB defines a clear and easy to understand intermediate bytecode language and provides a runtime environment for executing the bytecode. cphVB also contains a protocol to govern the safe, and efficient exchange, creation, and destruction of model data.

cphVB provides a retargetable framework in which the user can write programs utilizing whichever cphVB supported programming interface they prefer and run the program on their own workstation while doing prototyping, such as testing correctness and functionality of their programs. Users can then deploy exactly the same program in a more powerful execution environment without changing a single line of code and thus effectively solve greater problem sets.

The rest of the paper is organized as follows. In Section *Programming Model.* we describe the programming model supported by cphVB. The section following gives a brief description of *Numerical Python*, which is the first programming interface that fully supports cphVB. Sections *Design* and *Implementation* cover the overall cphVB design and an implementation of it. In Section *Performance Study*, we conduct an evaluation of the implementation. Finally, in Section *Future Work* and *Conclusion* we discuss future work and conclude.

Related Work

The key motivation for cphVB is to provide a framework for the utilization of heterogeneous computing systems with the

* *Corresponding author: madsbk@nbi.dk*
‡ *University of Copenhagen*

*. Open Source Project - Website: http://cphvb.bitbucket.org.

goal of obtaining high-performance, high-productivity and high-portability (HP^3). Systems such as pyOpenCL/pyCUDA [Klo09] provides a direct mapping from front-end language to the optimization target. In this case, providing the user with direct access to the low-level systems OpenCL [Khr10] and CUDA [Nvi10] from the high-level language Python [Ros10]. The work in [Klo09] enables the user to write a low-level implementation in a high-productivity language. The goal is similar to cphVB – the approach however is entirely different. cphVB provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Intel Math Kernel Library [Int08] is in this regard more comparable to cphVB. Intel MKL is a programming library providing utilization of multiple targets ranging from a single-core CPU to a multi-core shared memory CPU and even to a cluster of computers all through the same programming API. However, cphVB is not only a programming library it is a runtime system providing support for a vector oriented programming model. The programming model is well-known from high-productivity languages such as MATLAB [Mat10], [Rrr11], [Idl00], GNU Octave [Oct97] and Numerical Python (NumPy) [Oli07] to name a few.

cphVB is more closely related to the work described in [Gar10], here a compilation framework is provided for execution in a hybrid environment consisting of both CPUs and GPUs. Their framework uses a Python/NumPy based front-end that uses Python decorators as hints to do selective optimizations. cphVB similarly provides a NumPy based front-end and equivalently does selective optimizations. However, cphVB uses a slightly less obtrusive approach; program selection hints are sent from the front-end via the NumPy-bridge. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of cphVB without changing a single line of code. Whereas unPython requires the user to manually modify the source code by applying hints in a manner similar to that of OpenMP [Pas05]. This non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator [Dav04] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in cphVB but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. cphVB instead allows indexed operations and additionally supports **array-views**, which are array-aliases that provide multiple ways to access the same chunk of allocated memory. Thus, the data structure in cphVB is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [New11] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in cphVB as a plain and simple configuration file that define the cphVB runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a vector oriented programming model similar to cphVB. However, ArBB only provides access to the programming model via C++ whereas cphVB is not biased towards any one specific front-end language.

On multiple points cphVB is closely related in functionality and goals to the SEJITS [Cat09] project. SEJITS takes a different approach towards the front-end and programming model.

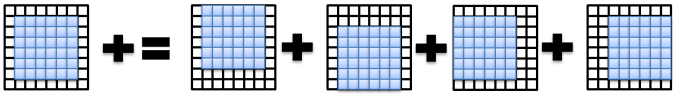


Fig. 1: *Matrix expression of a simple 5-point stencil computation example. See line eight in the code example, for the Python expression.*

SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards an optimality criteria. This approach has shown to provide performance that at times out-performs even hand-written specialized code towards a given architecture. Being able to construct computational kernels is a core issue in data-parallel programming.

The programming model in cphVB does not provide this kernel methodology. cphVB has a strong NumPy heritage which also shows in the programming model. The advantage is easy adaptability of the cphVB programming model for users of NumPy, Matlab, Octave and R. The cphVB programming model is not a stranger to computational kernels – cphVB deduce computational kernels at runtime by inspecting the vector bytecode generated by the Bridge.

cphVB provides in this sense a virtual machine optimized for execution of vector operations, previous work [And08] was based on a complete virtual machine for generic execution whereas cphVB provides an optimized subset.

Numerical Python

Before describing the design of cphVB, we will briefly go through Numerical Python (NumPy) [Oli07]. Numerical Python heavily influenced many design decisions in cphVB – it also uses a vector oriented programming model as cphVB.

NumPy is a library for numerical operations in Python, which is implemented in the C programming language. NumPy provides the programmer with a multidimensional array object and a whole range of supported array operations. By using the array operations, NumPy takes advantage of efficient C-implementations while retaining the high abstraction level of Python.

NumPy uses an array syntax that is based on the Python list syntax. The arrays are indexed positionally, 0 through length – 1, where negative indexes is used for indexing in the reversed order. Like the list syntax in Python, it is possible to index multiple elements. All indexing that represents more than one element returns a view of the elements rather than a new copy of the elements. It is this view semantic that makes it possible to implement a stencil operation as illustrated in Figure 1 and demonstrated in the code example below. In order to force a real array copy rather than a new array reference NumPy provides the "copy" method.

In the rest of this paper, we define the **array-base** as the originally allocated array that lies contiguously in memory. In addition, we will define the **array-view** as a view of the elements in an **array-base**. An **array-view** is usually a subset of the elements in the **array-base** or a re-ordering such as the reverse order of the elements or a combination.

```
1 center = full[1:-1, 1:-1]
2 up     = full[0:-2, 1:-1]
3 down   = full[2:  , 1:-1]
4 left   = full[1:-1, 0:-2]
5 right  = full[1:-1, 2:  ]
```

```
while epsilon < delta:
    work[:] = center
    work += 0.2 * (up+down+left+right)
    center[:] = work
```

Target Programming Model

To hide the complexities of obtaining high-performance from a heterogeneous environment any given system must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: 1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. 2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

cphVB is not biased towards any specific choice of abstraction or front-end technology as long as it is compatible with a vector oriented programming model. This provides means to use cphVB in functional programming languages, provide a front-end with a strict mathematic notation such as APL [Apl00] or a more relaxed syntax such as MATLAB.

The vector oriented programming model encourages expressing programs in the form of high-level array operations, e.g. by expressing the addition of two arrays using one high-level function instead of computing each element individually. The NumPy application in the code example above figure 1 is a good example of using the vector oriented programming model.

Design of cphVB

The key contribution in this paper is a framework, cphVB, that support a vector oriented programming model. The idea of cphVB is to provide the mechanics to seamlessly couple a programming language or library with an architecture-specific implementation of vectorized operations.

cphVB consists of a number of components that communicate using a simple protocol. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that perfectly match a specific execution environment. cphVB consist of the following components:

Programming Interface
: The programming language or library exposed to the user. cphVB was initially meant as a computational back-end for the Python library NumPy, but we have generalized cphVB to potential support all kinds of languages and libraries. Still, cphVB has design decisions that are influenced by NumPy and its representation of vectors/matrices.

Bridge
: The role of the Bridge is to integrate cphVB into existing languages and libraries. The Bridge generates the cphVB bytecode that corresponds to the user-code.

Vector Engine
: The Vector Engine is the architecture-specific implementation that executes cphVB bytecode.

Vector Engine Manager
: The Vector Engine Manager manages data location and ownership of vectors. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

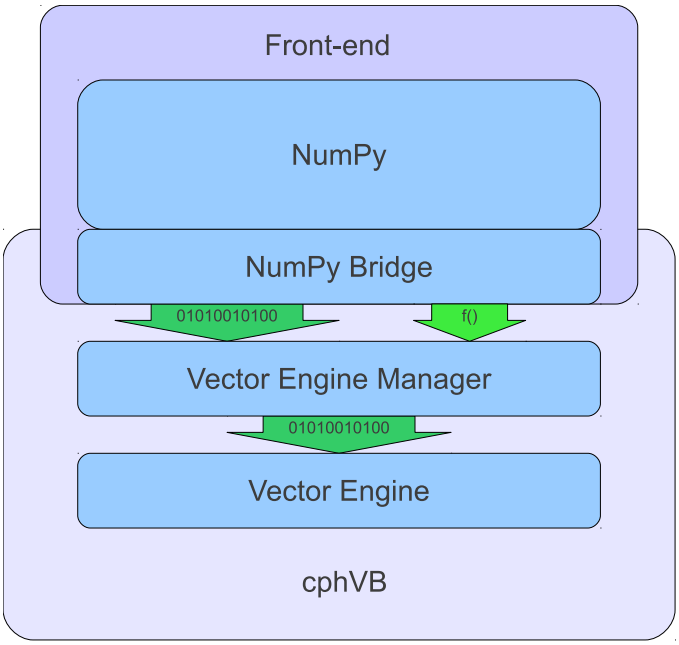


Fig. 2: *cphVB design idea.*

An overview of the design can be seen in Figure 2.

Configuration

To make cphVB as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user can change the setup of components simply by editing the configuration file before executing the user application. Additionally, the user only has to change the configuration file in order to run the application on different systems with different computational resources. The configuration file uses the ini syntax, an example is provided below.

```
# Root of the setup
[setup]
bridge = numpy
debug = true

# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libcphvb_vem_node.so
children = mcore

# Vector Engine using TLP on shared memory
[mcore]
type = ve
impl = libcphvb_ve_mcore.so
```

This example configuration provides a setup for utilizing a shared memory machine with thread-level-parallelism (TLP) on one machine by instructing the vector engine manager to use a single multi-core TLP engine.

Bytecode

The central part of the communication between all the components in cphVB is vector bytecode. The goal with the bytecode language is to be able to express operations on multidimensional vectors.

Taking inspiration from single instruction, multiple data (SIMD) instructions but adding structure to the data. This, of course, fits very well with the array operations in NumPy but is not bound nor limited to these.

We would like the bytecode to be a concept that is easy to explain and understand. It should have a simple design that is easy to implement. It should be easy and inexpensive to generate and decode. To fulfill these goals we chose a design that conceptually is an assembly language where the operands are multidimensional vectors. Furthermore, to simplify the design the assembly language should have a one-to-one mapping between instruction mnemonics and opcodes.

In the basic form, the bytecode instructions are primitive operations on data, e.g. addition, subtraction, multiplication, division, square root etc. As an example, let us look at addition. Conceptually it has the form:

```
add $d, $a, $b
```

Where `add` is the opcode for addition. After execution `$d` will contain the sum of `$a` and `$b`.

The requirement is straightforward: we need an opcode. The opcode will explicitly identify the operation to perform. Additionally the opcode will implicitly define the number of operands. Finally, we need some sort of symbolic identifiers for the operands. Keep in mind that the operands will be multidimensional arrays.

Interface

The Vector Engine and the Vector Engine Manager exposes simple API that consists of the following functions: initialization, finalization, registration of a user-defined operation and execution of a list of bytecodes. Furthermore, the Vector Engine Manager exposes a function to define new arrays.

Bridge

The Bridge is the **bridge** between the programming interface, e.g. Python/NumPy, and the Vector Engine Manager. The Bridge is the only component that is specifically implemented for the programming interface. In order to add cphVB support to a new language or library, one only has to implement the bridge component. It generates bytecode based on programming interface and sends them to the Vector Engine Manager.

Vector Engine Manager

Instead of allowing the front-end to communicate directly with the Vector Engine, we introduce a Vector Engine Manager (VEM) into the design. It is the responsibility of the VEM to manage data ownership and distribute bytecode instructions to several Vector Engines. It is also the ideal place to implement code optimization, which will benefit all Vector Engines.

To facilitate late allocation, and early release of resources, the VEM handles instantiation and destruction of arrays. At array creation only the meta data is actually created. Often arrays are created with structured data (e.g. random, constants), with no data at all (e.g. empty), or as a result of calculation. In any case it saves, potentially several, memory copies to delay the actual memory allocation. Typically, array data will exist on the computing device exclusively.

In order to minimize data copying we introduce a data ownership scheme. It keeps track of which components in cphVB that needs to access a given array. The goal is to allow the system to have several copies of the same data while ensuring that they are in synchronization. We base the data ownership scheme on two instructions, **sync** and **discard**:

Sync
: is issued by the bridge to request read access to a data object. This means that when acknowledging a **sync** request, the copy existing in shared memory needs to be the most resent copy.

Discard
: is used to signal that the copy in shared memory has been updated and all other copies are now invalid. Normally used by the bridge to upgrading a read access to a write access.

The cphVB components follow the following four rules when implementing the data ownership scheme:

1) The Bridge will always ask the Vector Engine Manager for access to a given data object. It will send a **sync** request for read access, followed by a **release** request for write access. The Bridge will not keep track of ownership itself.
2) A Vector Engine can assume that it has write access to all of the output parameters that are referenced in the instructions it receives. Likewise, it can assume read access on all input parameters.
3) A Vector Engine is free to manage its own copies of arrays and implement its own scheme to minimize data copying. It just needs to copy modified data back to share memory when receiving a **sync** instruction and delete all local copies when receiving a **discard** instruction.
4) The Vector Engine Manager keeps track of array ownership for all its children. The owner of an array has full (i.e. write) access. When the parent component of the Vector Engine Manager, normally the Bridge, request access to an array, the Vector Engine Manager will forward the request to the relevant child component. The Vector Engine Manager never accesses the array itself.

Additionally, the Vector Engine Manager needs the capability to handle multiple children components. In order to maximize parallelism the Vector Engine Manager can distribute workload and array data between its children components.

Vector Engine

Though the Vector Engine is the most complex component of cphVB, it has a very simple and a clearly defined role. It has to execute all instructions it receives in a manner that obey the serialization dependencies between instructions. Finally, it has to ensure that the rest of the system has access to the results as governed by the rules of the **sync**, **release**, and **discard** instructions.

Implementation of cphVB

In order to demonstrate our cphVB design we have implemented a basic cphVB setup. This concretization of cphVB is by no means exhaustive. The setup is targeting the NumPy library executing on a single machine with multiple CPU-cores. In this section, we will describe the implementation of each component in the cphVB setup – the Bridge, the Vector Engine Manager, and the Vector Engine. The cphVB design rules (Sec. Design) govern the interplay between the components.

Bridge

The role of the Bridge is to introduce cphVB into an already existing project. In this specific case NumPy, but could just as well be `R` or any other language/tool that works primarily on vectorizable operations on large data objects.

It is the responsibility of the Bridge to generate cphVB instructions on basis of the Python program that is being run. The NumPy Bridge is an extension of NumPy version 1.6. It uses hooks to divert function call where the program access cphVB enabled NumPy arrays. The hooks will translate a given function into its corresponding cphVB bytecode when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forcing NumPy to handle the function call itself.

The Bridge operates with two address spaces for arrays: the cphVB space and the NumPy space. All arrays starts in the NumPy space as a default. The original NumPy implementation handles these arrays and all operations using them. It is possible to assign an array to the cphVB space explicitly by using an optional cphVB parameter in array creation functions such as `empty` and `random`. The cphVB bridge implementation handles these arrays and all operations using them.

In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1) When an operation accesses an array in the cphVB address space but it is not possible for the bridge to translate the operation into cphVB code. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency no data is actually copied instead the bridge uses the `mremap`[†] function to remap the relevant memory pages.
2) When an operations access arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the cphVB space. Afterwards, the bridge will translate the operation into bytecode that cphVB can execute.

In order to detect direct access to arrays in the cphVB address space by the user, the original NumPy implementation, a Python library or any other external source, the bridge protects the memory of arrays that are in the cphVB address space using `mprotect`[‡]. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application since it can always fallback to the original NumPy implementation.

In order to gather greatest possible information at runtime, the Bridge will collect a batch of instructions rather than executing one instruction at a time. The Bridge will keep recording instruction until either the application reaches the end of the program or untranslatable NumPy operations forces the Bridge to move an array to the NumPy address space. When this happens, the Bridge will call the Vector Engine Manager to execute all instructions recorded in the batch.

Vector Engine Manager

The Vector Engine Manager (VEM) in our setup is very simple because it only has to handle one Vector Engine thus all operations go to the same Vector Engine. Still, the VEM creates and deletes arrays based on specification from the Bridge and handles all meta-data associated with arrays.

Processor	Intel Core i5-2510M
Clock	2.3 GHz
Private L1 Data Cache	128 KB
Private L2 Data Cache	512 KB
Shared L3 Cache	3072 KB
Memory Bandwidth	21.3 GB/s
Memory	4GB DDR3-1333
Compiler	GCC 4.6.3

TABLE 1: *ASUS P31SD.*

Vector Engine

In order to maximize the CPU cache utilization and enables parallel execution the first stage in the VE is to form a set of instructions that enables data blocking. That is, a set of instructions where all instructions can be applied on one data block completely at a time without violating data dependencies. This set of instructions will be referred to as a kernel.

The VE will form the kernel based on the batch of instructions it receives from the VEM. The VE examines each instruction sequentially and keep adding instruction to the kernel until it reaches an instruction that is not **blockable** with the rest of the kernel. In order to be blockable with the rest of the kernel an instruction must satisfy the following two properties where A is all instructions in the kernel and N is the new instruction.

1) The input arrays of N and the output array of A do not share any data or represents precisely the same data.
2) The output array of N and the input and output arrays of A do not share any data or represents precisely the same data.

When the VE has formed a kernel, it is ready for execution. Since all instruction in a kernel supports data blocking the VE can simply assign one block of data to each CPU-core in the system and thus utilizing multiple CPU-cores. In order to maximize the CPU cache utilization the VE may divide the instructions into even more data blocks. The idea is to access data in chunks that fits in the CPU cache. The user, through an environment variable, manually configures the number of data blocks the VE will use.

Performance Study

In order to demonstrate the performance of our initial cphVB implementation and thereby the potential of the cphVB design, we will conduct some performance benchmarks using NumPy[§]. We execute the benchmark applications on ASUS P31SD with an Intel Core i5-2410M processor (Table 1).

The experiments used the three vector engines: *simple*, *score* and *mcore* and for each execution we calculate the relative speedup of cphVB compared to NumPy. We perform strong scaling experiments, in which the problem size is constant though all the executions. For each experiment, we find the block size that results in best performance and we calculate the result of each experiment using the average of three executions.

†. The function mremap() in GNU C library 2.4 and greater.
‡. The function mprotect() in the POSIX.1-2001 standard.

The benchmark consists of the following Python/NumPy applications. All are pure Python applications that make use of NumPy and none uses any external libraries.

- **Jacobi Solver** An implementation of an iterative jacobi solver with fixed iterations instead of numerical convergence. (Fig. 3).
- **kNN** A naive implementation of a k Nearest Neighbor search (Fig. 4).
- **Shallow Water** A simulation that simulates a system governed by the shallow water equations. It is a translation of a MATLAB application by Burkardt [Bur10] (Fig. 5).
- **Synthetic Stencil** A synthetic stencil simulation the code relies heavily on the slicing operations of NumPy. (Fig. 6).

Discussion

The jacobi solver shows an efficient utilization of data-blocking to an extent competing with using multiple processors. The *score* engine achieves a 1.42x speedup in comparison to NumPy (3.98*sec* to 2.8*sec*).

On the other hand, our naive implementation of the k Nearest Neighbor search is not an embarrassingly parallel problem. However, it has a time complexity of $O(n^2)$ when the number of elements and the size of the query set is n, thus the problem should be scalable. The result of our experiment is also promising – with a performance speedup of of 3.57x (5.40*sec* to 1.51*sec*) even with the two single-core engines and a speed-up of nearly 6.8x (5.40*sec* to 0.79) with the multi-core engine.

The Shallow Water simulation only has a time complexity of $O(n)$ thus it is the most memory intensive application in our benchmark. Still, cphVB manages to achieve a performance speedup of 1.52x (7.86*sec* to 5.17*sec*) due to memory-allocation optimization and 2.98x (7.86*sec* to 2.63*sec*) using the multi-core engine.

Finally, the synthetic stencil has an almost identical performance pattern as the shallow water benchmark the *score* engine does however give slightly better results than the *simple* engine. Score achieves a speedup of 1.6x (6.60*sec* to 4.09*sec*) and the *mcore* engine achieves a speedup of 3.04x (6.60*sec* to 2.17*sec*).

It is promising to observe that even most basic vector engine (*simple*) shows a speedup and in none of our benchmarks a slowdown. This leads to the promising conclusion that the memory optimizations implemented outweigh the cost of using cphVB. Adding the potential of speedup due to data-blocking motivates studying further optimizations in addition to thread-level-parallelization. The *mcore* engine does provide speedups, the speedup does however not scale with the number of cores. This result is however expected as the benchmarks are memory-intensive and the memory subsystem is therefore the bottleneck and not the number of computational cores available.

Future Work

The future goals of cphVB involves improvement in two major areas; expanding support and improving performance. Work has started on a CIL-bridge which will leverage the use of cphVB to every CIL based programming language which among others

§. NumPy version 1.6.1.

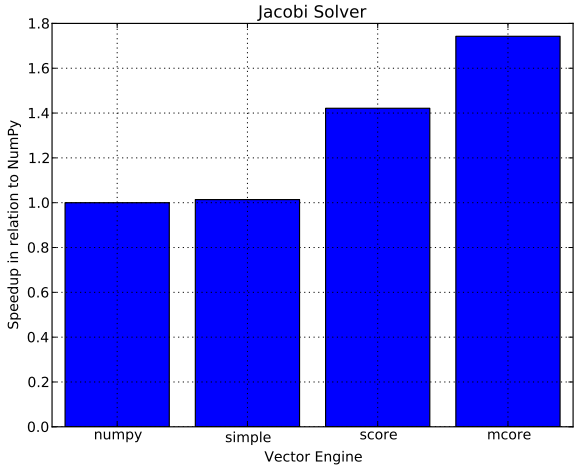


Fig. 3: *Relative speedup of the Jacobi Method. The job consists of a vector with* 7168*x*7168 *elements using four iterations.*

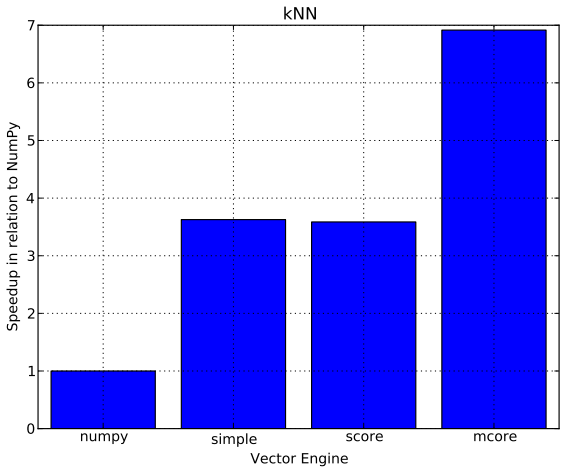


Fig. 4: *Relative speedup of the k Nearest Neighbor search. The job consists of 10.000 elements and the query set also consists of 1K elements.*

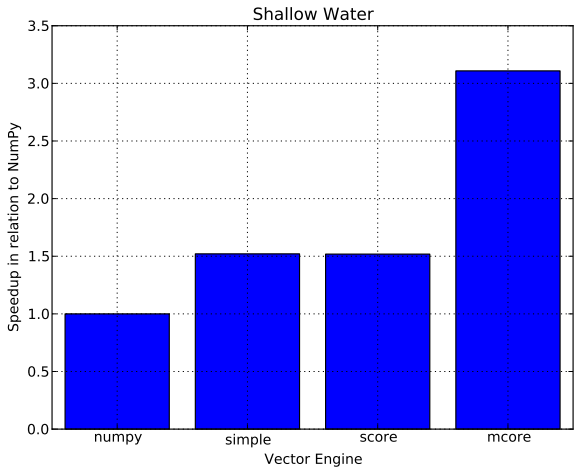


Fig. 5: *Relative speedup of the Shallow Water Equation. The job consists of 10.000 grid points that simulate 120 time steps.*

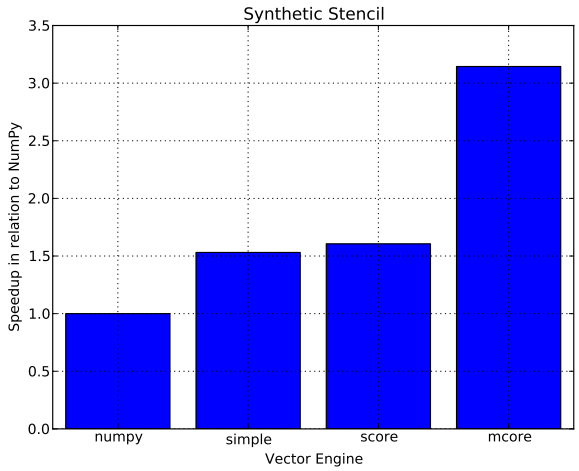


Fig. 6: *Relative speedup of the synthetic stencil code. The job consists of vector with* 10240*x*1024 *elements that simulate 10 time steps.*

include: C#, F#, Visual C++ and VB.NET. Another project in current progress within the area of support is a C++ bridge providing a library-like interface to cphVB using operator overloading and templates to provide a high-level interface in C++.

To improve both support and performance, work is in progress on a vector engine targeting OpenCL compatible hardware, mainly focusing on using GPU-resources to improve performance. Additionally the support for program execution using distributed memory is on progress. This functionality will be added to cphVB in the form a vector engine manager.

In terms of pure performance enhancement, cphVB will introduce JIT compilation in order to improve memory intensive applications. The current vector engine for multi-cores CPUs uses data blocking to improve cache utilization but as our experiments show then the memory intensive applications still suffer from the von Neumann bottleneck [Bac78]. By JIT compile the instruction kernels, it is possible to improve cache utilization drastically.

Conclusion

The vector oriented programming model used in cphVB provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the cphVB design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist. The authors in [Kri11] demonstrate that combining shared memory and distributed memory parallelism through hybrid programming is essential in order to utilize the Blue Gene/P architecture fully.

In a case study, we demonstrate the design of cphVB by implementing a front-end for Python/NumPy that targets multi-core CPUs in a shared memory environment. The implementation executes vectorized applications in parallel without any user intervention. Thus showing that it is possible to retain the high abstraction level of Python/NumPy while fully utilizing the underlying hardware. Furthermore, the implementation demonstrates scalable performance – a k-nearest neighbor search purely written in Python/NumPy obtains a speedup of more than five compared to a native execution.

Future work will further test the cphVB design model as new front-end technologies and heterogeneous architectures are supported.

References

[Kri10] M. R. B. Kristensen and B. Vinter, *Numerical Python for Scalable Architectures*, in Fourth Conference on Partitioned Global Address Space Programming Model, PGAS{'}10. ACM, 2010. [Online]. Available: http://distnumpy.googlecode.com/files/kristensen10.pdf

[Dav04] T. David, P. Sidd, and O. Jose, *Accelerator : Using Data Parallelism to Program GPUs for General-Purpose Uses*, October. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70250

[New11] C. J. Newburn, B. So, Z. Liu, M. Mccool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, *Intels Array Building Blocks : A Retargetable , Dynamic Compiler and Embedded Language*, Symposium A Quarterly Journal In Modern Foreign Literatures, pp. 1–12, 2011. [Online]. Available: http://software.intel.com/en-us/blogs/wordpress/wp-content/uploads/2011/03/ArBB-CGO2011-distr.pdf

[Klo09] A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, o and A. Fasih, *PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation*, Brain, vol. 911, no. 4, pp. 1–24, 2009. [Online]. Available: http://arxiv.org/abs/0911.3456

[Khr10] K. Opencl, W. Group, and A. Munshi, *OpenCL Specification*, ReVision, pp. 1–377, 2010. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OpenCL+Specification#2

[Nvi10] N. Nvidia, *NVIDIA CUDA Programming Guide 2.0*, pp. 1–111, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/32prod/toolkit/docs/CUDACProgrammingGuide.pdf

[Ros10] G. V. Rossum and F. L. Drake, *Python Tutorial*, History, vol. 42, no. 4, pp. 1–122, 2010. [Online]. Available: http://docs.python.org/tutorial/

[Int08] Intel, *Intel Math Kernel Library (MKL)*, pp. 2–4, 2008. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl/

[Mat10] MATLAB, version 7.10.0 (R2010a). Natick, Massachusetts: The MathWorks Inc., 2010.

[Rrr11] R Development Core Team, *R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing*, Vienna, Austria, 2011. [Online]. Available: http://www.r-project.org

[Idl00] B. A. Stern, *Interactive Data Language*, ASCE, 2000.

[Oct97] J. W. Eaton, *GNU Octave*, History, vol. 103, no. February, pp. 1–356, 1997. [Online]. Available: http://www.octave.org

[Oli07] T. E. Oliphant, *Python for Scientific Computing*, Computing in Science Engineering, vol. 9, no. 3, pp. 10–20, 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4160250

[Gar10] R. Garg and J. N. Amaral, *Compiling Python to a hybrid execution environment*, Computing, pp. 19–30, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1735688.1735695

[Pas05] R. V. D. Pas, *An Introduction Into OpenMP*, ACM SIGARCH Computer Architecture News, vol. 34, no. 5, pp. 1–82, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1168898

[Cat09] B. Catanzaro, S. Kamil, Y. Lee, K. Asanov'i, J. Demmel, c K. Keutzer, J. Shalf, K. Yelick, and O. Fox, *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization*, in Proc of 1st Workshop Programmable Models for Emerging Architecture PMEA, no. UCB/EECS-2010-23, EECS Department, University of California, Berkeley. Citeseer, 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html

[And08] R. Andersen and B. Vinter, *The Scientific Byte Code Virtual Machine*, in Proceedings of the 2008 International Conference on Grid Computing & Applications, GCA2008 : Las Vegas, Nevada, USA, July 14-17, 2008. CSREA Press., 2008, pp. 175–181. [Online]. Available: http://dk.migrid.org/public/doc/published_papers/sbc.pdf

[Apl00] "why apl?" [Online]. Available: http://www.sigapl.org/whyapl.htm

[Sci02] R. Pozo and B. Miller, *SciMark 2.0*, 2002. [Online]. Available: http://math.nist.gov/scimark2/

[Bur10] J. Burkardt, *Shallow Water Equations*, 2010. [Online]. Available: http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/

[Bac78] J. Backus, *Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs*, Communications of the ACM, vol. 16, no. 8, pp. 613–641, 1978.

[Kri11] M. Kristensen, H. Happe, and B. Vinter, *Hybrid Parallel Programming for Blue Gene/P*, Scalable Computing: Practice and Experience, vol. 12, no. 2, pp. 265–274, 2011.

OpenMG: A New Multigrid Implementation in Python

Tom S. Bertalan‡*, Akand W. Islam‡, Roger B. Sidje§, Eric Carlson‡

Abstract—In many large-scale computations, systems of equations arise in the form $Au = b$, where A is a linear operation to be performed on the unknown data u, producing the known right-hand side, b, which represents some constraint of known or assumed behavior of the system being modeled. Since such systems can be very large, solving them directly can be too slow. In contrast, a multigrid solver solves partially at full resolution, and then solves directly only at low resolution. This creates a correction vector, which is then interpolated to full resolution, where it corrects the partial solution. This project aims to create an open-source multigrid solver called OpenMG, written only in Python. The existing PyAMG multigrid implementation is a highly versatile, configurable, black-box solver, but is difficult to read and modify due to its C core. Our proposed OpenMG is a pure Python experimentation environment for testing multigrid concepts, not a production solver. By making the code simple and modular, we make the algorithmic details clear. We thereby create an opportunity for education and experimentation with the partial solver (Jacobi, Gauss Seidel, SOR, etc.), the restriction mechanism, the prolongation mechanism, and the direct solver, or the use of GPGPUs, multiple CPUs, MPI, or grid computing. The resulting solver is tested on an implicit pressure reservoir simulation problem with satisfactory results.

Index Terms—python, multigrid, numpy, partial differential equations

Introduction to Multigrid

Multigrid algorithms aim to accelerate the solution of large linear systems that typically arise from the discretization of partial differential equations. While small systems (hundreds of unknowns) can efficiently be solved with direct methods such as Gaussian elimination or iterative methods such as Gauss-Seidel, these methods do not scale well. In contrast, multigrid methods can theoretically solve a system in $O(N)$ CPU steps and memory usage [Brandt2].

The entire multigrid algorithm can be summarized in a few steps. The process below assumes that the user has first discretized the partial differential equation ("PDE") of interest, or otherwise expressed the problem as a matrix system of equations.

1) Setup hierarchies of operators and restriction matrices.
2) Find an approximation to the solution (pre-smooth the high-frequency error).
3) Find the fine residual.
4) Coarsen the residual, and produce the coarse right-hand side.
5) Solve at the coarse level (via a direct solver or a recursive call to the multigrid algorithm).
6) Prolong the coarse solution, and produce the fine correction vector.
7) Return the corrected solution.

Because of the possibility for a recursive call, this is often called a multigrid "cycle".

The basic premise of multigrid is that a quick but sloppy solution can be corrected using information calculated at a coarser resolution. That is, an approximation is first made at the fine resolution, and the residual from this approximation is used as the right-hand side for a correction equation to be solved at a much coarser resolution, where computational costs are also much lower. This basic two-grid scheme can be extended by using a recursive call at the coarse level instead of a direct solver.

History

Multigrid techniques were first introduced in 1964 in the USSR by R. P. Fedorenko [Fedorenko1], [Fedorenko2], who recognized the significance of the interaction between the mesh resolution and the components of the error of an iterative solution (see section *Pre-Smoothing*), but who initially conceived of the multigrid algorithm simply as an occasional correction to a basic iterative solver. Rigorous analysis of the technique was furthered in the seventies by Achi Brandt, Wolfgang Hackbusch, and R. A. Nicolaides. Brandt [Brandt2] placed more emphasis on the coarse-grid representations, describing multigrid as a method by which to "intermix discretization and solution processes, thereby making both of them orders of magnitude more effective." He further recast the process in terms of local and global mode analysis (Fourier analysis) in 1994 [Brandt1]. In 1979 Nicolaides wrote a useful synthesis of the work of Fedorenko and Brandt up to that point, and also contrasted the older two-level coarse-grid correction strategy with true, l-level multigrid [Nicolaides]. Hackbrush wrote one of the foundational texts on multigrid [Hackbusch].

More information on the history of multigrid techniques can be found in several books [Trottenberg], [Hackbusch], [Wesseling] or lecture notes [Heckbert] on the topic.

Examples of simulation problem domains that have benefited from multigrid techniques include porous media transport [Douglas2], [Kameswaran], molecular dynamics [Dzwinel], [Zapata], [Boschitsch], fluid dynamics [Denev], [Douglas2], [Kameswaran], and neural network simulations (and neurotransmitter diffusion) [Bakshi].

Multigrid concepts are not limited to applications in simulation. Mipmapped textures for computer graphics and ultra-high-resolution image viewing applications such as satellite imaging

* *Corresponding author: tom@tombertalan.com*
‡ *The University of Alabama, Department of Chemical and Biological Engineering*
§ *The University of Alabama, Department of Mathematics*

both rely on the concept of a hierarchy of grid resolutions. Here, intergrid transfer operators are used for the purpose of creating images at different resolutions than the original.

Existing Python Implementations

The current open-source Python multigrid implementation *PyAMG* (due to Nathan Bell [Bell]) is a very capable and speedy multigrid solver, with a core written in C. However, because of the extent of optimizations (and the inclusion of C code), it is not particularly readable.

Another interesting implementation is Yvan Notay's AGMG, which is available for Matlab and Fortran and includes parallel versions [Notay], [AGMG]. AGMG is available for free for academic use and by site-license for commercial use.

Our project, OpenMG, is not intended to be a production solver but instead a tool for education and experimentation. In this, it is largely inspired by the intentions behind Douglas, Deng, Haase, Liebmann, and Mckenzie's AMGlab [Douglas1], which is written for MATLAB. (AMGlab is freely available, although a license does not seem to be specified.) OpenMG is constructed in a modular fashion so each part can be understood by itself. Optimizations that might decrease readability have been avoided. Because of the modularity of the system, simplified components of the algorithm can be overridden with more optimized components in the future.

Theoretical Algorithm

Discretization

The need for any sort of linear algebraic solver arises when a system of partial differential equations is discretized on a finite grid of points. While this is not the work of the OpenMG solver itself (the arguments to the solver are already in discretized form), it is a necessary preliminary step.

A good illustration of discretization is that of the Poisson equation, $\nabla u = 0$. Here, ∇ is the Laplace operator, which signifies the sum of unmixed second partial derivatives.

$$\nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

One possible discretization of this equation uses a central difference of both forward- and backwards-difference discretizations of the first partial derivatives.

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{\frac{u_{i,j+1}-u_{i,j}}{h} - \frac{u_{i,j}-u_{i,j-1}}{h}}{h}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{\frac{u_{i+1,j}-u_{i,j}}{h} - \frac{u_{i,j}-u_{i-1,j}}{h}}{h}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \approx \left(\frac{1}{h^2}\right)(1u_{i-1,j} + 1u_{i,j-1} - 4u_{i,j} + 1u_{i,j+1} + 1u_{i+1,j}) \quad (1)$$

When applied to every point in the domain, the coefficient pattern $1,1,-4,1,1$ produces a five-banded square coefficent matrix A in the equation

$$Au = b \quad (2)$$

where u is the vector of unknowns, for which we must solve, and the right-hand side b includes boundary information.

1. Setup R and A Hierarchies

The basic requirement of multigrid is, unsurprisingly, a multiplicy of grids, each discretizing the problem domain at a different resolution. In the simplest ("two-grid") scheme, there are two grid levels, h and H, where grid h has N_h unknowns, grid H has N_H unknowns, $N_h > N_H$, and (for regular Cartesian grids) the values of h and H represent the fine and coarse grid spacings, respectively.

In *geometric multigrid*, the operator at the fine level A_h is replaced by the operator at the coarse level A_H by re-discretizing the underlying PDE. However, this method, while potentially faster, enforces a tighter coupling between the solver and the simulation problem at hand.

The alternative to geometric multigrid is *algebraic multigrid*, in which the coarse operator is derived not from the PDE but only from the fine operator. Ruge-Steuben coarsening bases this transformation on the pattern of coefficients in A_h, but our current implementation (see *Implementation*) instead uses a stencil-based average.

Before the cycling portion of the algorithm, a setup phase is executed in which we generate a hierarchy of restriction matrices and coefficient matrices. The restriction array at position h in the hierarchy, where the number of unknowns is N_h, and where the number of unknowns for the next coarsest level is N_H, is R_h^H, or simply R_h. It functions as an intergrid transfer operator from grid H to grid h, and has shape (N_H, N_h). That is, it can reduce the size of a vector from N_h to N_H elements:

$$u_H = R_h u_h \quad (3)$$

These restriction matrices are used to produce a similar hierarchy of coefficient matrices, via the Galerkin coarse-grid approximation [Zeng].

$$A_H = R_h A_h R_h^T$$

This is significant because the multigrid algorithm thereby requires no knowledge of the underlying PDE to generate the coarse-grid operator. Instead, the coarse-grid operator is created solely through algebraic manipulation, giving rise to the term "algebraic multigrid".

It should be noted that the labels h and H are used because, in cartesian structured grids, the characteristic that distinguishes between grid levels is the spacing between points. It is geometrically intuitive to call the distance between points h in the fine grid and H in the coarse grid.

2. Pre-Smoothing: $u_{apx,h}$

An iterative solver is used to produce an initial estimate of the solution. This solver can be a Jacobi, Gauss-Seidel, or conjugate gradient implementation, or any other solver that can use a number-of-iterations parameter to make a tradeoff between overall accuracy and speed.

These iterative solvers begin with some initial guess of the solution, which could either be the work of previous solvers or simply a zero-vector. Because the iterative solvers reduce the high-frequency components of the error in this guess more quickly than they reduce the low-frequency ones, they are often referred to as "smoothers" in the context of multigrid methods. The purpose of a multigrid scheme is to use these iterative smoothers only at high resolution to reduce the high-frequency error, relying on corrections at lower resolution to reduce the low-frequency components of the error. [Harimi] See Figure 5 c, and accompanying explanations in *Test Definitions and Results*.

So,

$$u_{apx,h} = iterative\_solve(A_h, b_h, iterations) \tag{4}$$

where *iterations* is a small integer, often simply 1.

3. Residual: r_h

After the iterative solution, an error r_h in the approximation $u_{apx,h}$ can be defined as

$$A_h u_{apx,h} + r_h = b_h \tag{5}$$

where b_h is the given right-hand side.

4. Coarse Right-hand-side: b_H

let $r_h = A_h v_h$

$$A_h u_{apx,h} + A_h v_h = b_h \tag{6}$$

$$A_h(u_{apx,h} + v_h) = b_h \tag{7}$$

So, v_h functions as a correction vector for the iterative approximation. Equation 6 can be rearranged to produce another matrix equation in the same form as Equation 2:

$$A_h v_h = b_h - A_h u_{apx,h} \tag{8}$$

Here, every element on the right-hand side is known, so it can be used to form a new right-hand side with which we can solve for the correction v_h. However, because this correction only serves the purpose of reducing the low-frequency components of the error, we can safely solve Equation 8 at a coarser resolution without losing information [Borzi]. So, we make use of our hierarchy of restriction and coefficient matrices to make Equation 8 an easier problem to solve (fewer unknowns):

$$A_H v_H = R_h(b_h - A_h u_{apx,h}) \tag{9}$$

where A_H and R_h are taken from the hierarchy generated earlier.

5. Coarse Solution

The unknown vector and right-hand side of Equation 9 can now be replaced with new variables, revealing a new problem with only N_H unknowns, down from the N_h unknowns in Equation 8.

$$A_H u_H = b_H \tag{10}$$

Because this is simply another matrix equation similar in form to Equation 2, it can be solved either with a recursive call to the multigrid solver, or with a direct solver, such Numpy's `np.linalg.solve` or SciPy's `scipy.base.np.linalg.solve`.

6. Interpolate Correction

In order to correct the iterative approximation u_{apx}, the solution from the coarse problem must be interpolated from N_H unknowns up to N_h unknowns. Because the restriction matrices are defined algebraically in Equation 3, it is possible to define an interpolation (or "prolongation") algebraically:

$$v_h = R_h^T u_H \tag{11}$$

This is used to prolongate the solution u_H from the coarse level for use as a correction v_h at the fine level. Note that, at the coarse level, the symbol u is used, since this is a solution to the coarse problem, but, at the fine level, the symbol v is used, since this is not the solution, but a correction to the iterative approximation.

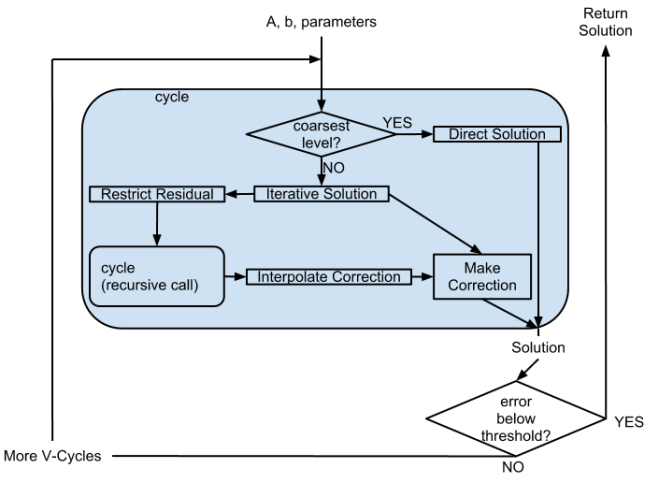


Fig. 1: *Recursive multigrid cycle, with V-cycle iteration until convergence.*

7. Return Corrected Solution

With the correction vector in hand, it is now possible to return a solution whose error has been reduced in both high- and low-frequency components:

$$u_h = u_{apx} + v_h \tag{12}$$

It is also possible to insert a second "post-smoothing" step between the interpolation and the return steps, similar to Equation 4.

As described in this section, this algorithm is a 2-grid V-cycle, because the high-resolution → low-resolution → high-resolution pattern can be visualized as a V shape. In our small sample problem, using more grid levels than two actually wasted enough time on grid setup to make the solver converge less quickly. However, repeated V-cycles were usually necessary for visually compelling convergence. That is, the solution from one V-cycle was used as the initial guess for the fine-grid pre-smoother of the next V-cycle. More complicated cycling patterns are also possible, such as W-cycles, or the full-multigrid ("FMG") pattern, which actually starts at the coarse level. However, these patterns are not yet addressed by OpenMG.

Implementation

The process shown in Figure 1 is a multigrid solver with nearly black-box applicability–the only problem-specific piece of information required (one of the "parameters" in the figure) is the shape of the domain, as a 3-tuple, and it is possible that future versions of `restriction()` will obviate this requirement. Note that, in code listings given below, `import numpy as np` is assumed.

Setup R and A Hierarchies

Any restriction can be described by a restriction matrix. Our current implementation, which is replacable in modular fashion, uses 2-point averages in one dimension, 4-point averages in two dimensions, and 8-point averages in three dimensions, as depicted in Figure 2. Alternate versions of these two functions have been developed that use sparse matrices, but the dense versions are shown here for simplicity.

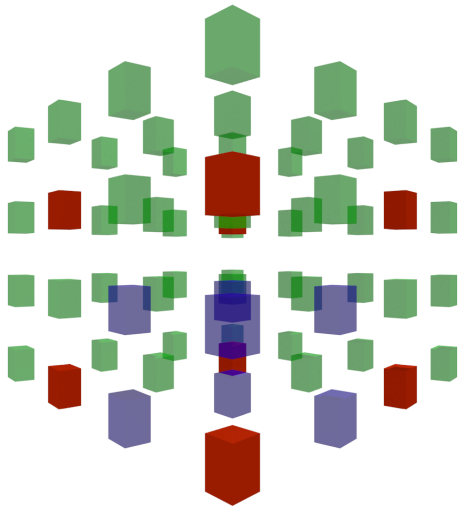


Fig. 2: *Eight-point average restriction method. All points are included in the fine set, but red points included in both the fine set and the coarse set. Blue points are used in the calculation of eight-point average for the coarse point nearest to the camera in the bottom plane.*

Other simplifications have also been made–for example, automatic V-cycling has been removed, although, in the actual code, this is contained with in the wrapper function `openmg.mg_solve()`. Forced line breaks have also reduced the readability of this sample code. We recommend downloading the most up-to-date OpenMG code from https://github.com/tsbertalan/openmg for working examples.

The following code generates a particular restriction matrix, given a number of unknowns `N`, and a problem domain shape tuple, `shape`. It fails (or works very inefficiently) for domains that have odd numbers of points along one or more dimensions. Operator-based coarsening would remove this restriction.

```
from sys import exit
def restriction(N, shape):
    alpha = len(shape)  # number of dimensions
    R = np.zeros((N / (2 ** alpha), N))
    r = 0  # rows
    NX = shape[0]
    if alpha >= 2:
        NY = shape[1]
    each = 1.0 / (2 ** alpha)
    if alpha == 1:
        coarse_columns = np.array(range(N)).\
                         reshape(shape)\
                         [::2].ravel()
    elif alpha == 2:
        coarse_columns = np.array(range(N)).\
                         reshape(shape)\
                         [::2, ::2].ravel()
    elif alpha == 3:
        coarse_columns = np.array(range(N)).\
                         reshape(shape)\
                         [::2, ::2, ::2].ravel()
    else:
        print "> 3 dimensions is not implemented."
        exit()
    for c in coarse_columns:
        R[r, c] = each
        R[r, c + 1] = each
        if alpha >= 2:
            R[r, c + NX] = each
            R[r, c + NX + 1] = each
            if alpha == 3:
                R[r, c + NX * NY] = each
                R[r, c + NX * NY + 1] = each
                R[r, c + NX * NY + NX] = each
                R[r, c + NX * NY + NX + 1] = each
        r += 1
    return R
```

The function `restriction()` is called several times by the following code to generate the complete hierarchy of restriction matrices.

```
def restrictions(N, problemshape, coarsest_level,\
                 dense=False, verbose=False):
    alpha = np.array(problemshape).size
    levels = coarsest_level + 1
    # We don't need R at the coarsest level:
    R = list(range(levels - 1))
    for level in range(levels - 1):
        newsize = N / (2 ** (alpha * level))
        R[level] = restriction(newsize,
                    tuple(np.array(problemshape)\
                        / (2 ** level)))
    return R
```

Using the hierarchy of restriction matrices produced by `restrictions()` and the user-supplied top-level coefficient matrix `A_in`, the following code generates a similar hierarchy of left-hand-side operators using the Galerkin coarse-grid approximation, $A_H = RA_hR^T$.

```
def coarsen_A(A_in, coarsest_level, R, dense=False):
    levels = coarsest_level + 1
    A = list(range(levels))
    A[0] = A_in
    for level in range(1, levels):
        A[level] = np.dot(np.dot(
                              R[level-1],
                              A[level-1]),
                          R[level-1].T)
    return A
```

Both `restrictions()` and `coarsen_A()` return lists of arrays.

Smoother

Our iterative smoother is currently a simple implementation of Gauss-Seidel smoothing, but this portion of the code could be replaced with a Jacobi implementation to allow parallelization if larger domains prove to spend more execution time here.

```
def iterative_solve(A, b, x, iterations):
    N = b.size
    iteration = 0
    for iteration in range(iterations):
        for i in range(N):  # [ 0 1 2 3 4 ... n-1 ]
            x[i] = x[i] + (b[i] - np.dot(
                                  A[i, :],
                                  x.reshape((N, 1)))
                          ) / A[i, i]
    return x
```

Multigrid Cycle

The following function uses all the preceeding functions to perform a multigrid cycle, which encompasses the *Residual*, *Coarse Solution*, *Interpolate Correction*, and *Return Corrected Solution* steps from the theoretical discussion above. It calls itself recursively until the specified number of `gridlevels` is reached. It can be called directly, or through a wrapper function with a more simplified prototype, `mg_solve(A_in, b, parameters)` (not shown here).

```
def amg_cycle(A, b, level, \
            R, parameters, initial='None'):
    # Unpack parameters, such as pre_iterations
    exec ', '.join(parameters) +\
        ',  = parameters.values()'
    if initial == 'None':
        initial = np.zeros((b.size, ))
    coarsest_level = gridlevels - 1
    N = b.size
    if level < coarsest_level:
        u_apx = iterative_solve(\
                                A[level],\
                                b,\
                                initial,\
                                pre_iterations,)
        b_coarse = np.dot(R[level],\
                          b.reshape((N, 1)))
        NH = len(b_coarse)
        b_coarse.reshape((NH, ))
        residual = b - np.dot(A[level], u_apx)
        coarse_residual = np.dot(\
                              R[level],\
                              residual.reshape((N, 1))\
                              ).reshape((NH,))
        coarse_correction = amg_cycle(
                              A,
                              coarse_residual,
                              level + 1,
                              R,
                              parameters,
                              )
        correction = np.dot(\
                              R[level].transpose(),\
                              coarse_correction.\
                              reshape((NH, 1))\
                            ).reshape((N, ))
        u_out = u_apx + correction
        norm = np.linalg.norm(b - np.dot(
                                    A[level],
                                    u_out.\
                                    reshape((N,1))
                                    ))
    else:
        norm = 0
        u_out = np.linalg.solve(A[level],\
                              b.reshape((N, 1)))
    return u_out
```

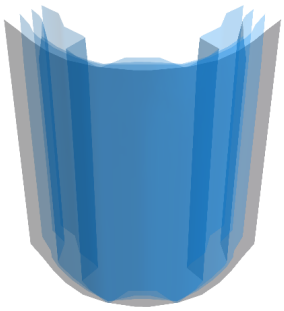


Fig. 3: *Pressure isosurfaces of several solutions to a 3D porous media problem with $12^3 = 1728$ unknowns. The grey outer surface is a direct solution, while the blue inner surfaces are the result of different numbers of multigrid V-cycles–with more V-cycles, the multigrid solution approaches the true solution. Plotted with MayaVi's* `mlab.contour3d`.

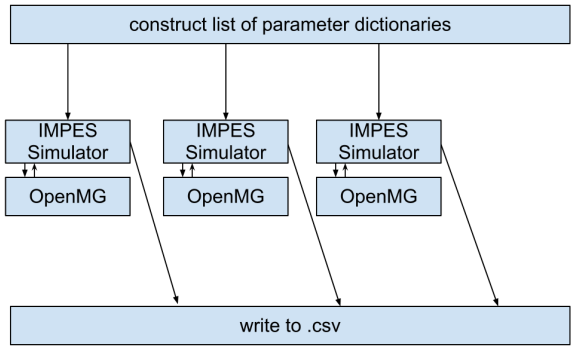


Fig. 4: *Parallel testing apparatus. The IMPES (implicit pressure, explicit saturation) simulation script calls the OpenMG script when solving its pressure equation, and then reports a dictionary of dependent variables of interest to be written to a comma-separated-value file.*

Results

Sample Application

In our test example we simulate the geologic sequestration of CO_2. The governing pressure-saturation equation is

$$v = -\mathbf{K}(\lambda_w + \lambda_{CO_2})\nabla p + \mathbf{K}(\lambda_w \rho_w + \lambda_{CO_2}\rho_{CO_2})G \tag{13}$$

and the saturation equation is

$$\phi \frac{\partial s_w}{\partial t} + \nabla\left(f_w(s_w)[v + d(s_w, \nabla s_w) + g(s_w)]\right) = \frac{q_w}{\rho_w} \tag{14}$$

where v is a velocity vector, the gravitational pull-down force G is $-g\nabla z$, subscript w represents water-saturated porous medium, g represents gravitational acceleration, $\mathbf{K}$ represents the permeability tensor, p represents fluid pressure, q models sources and sinks, (outflow or inflow), S represents saturation, z represents the vertical direction, ρ represents water density, ϕ represents porosity, and λ represents mobility (ratio of permeability to viscosity).

Equation 14, the saturation equation, is generally parabolic. However, the terms for the viscous force $f(s)v$ and the gravity force $f(s)g(s)$ usually dominate the capillary force $f(s)d(s,\nabla s)$. Therefore the equation will have a strong hyperbolic nature and can be solved by many schemes [Aarnes]. On the other hand, Equation 13, the pressure equation, is of elliptic form. After discretization, this equation will reduce to $Au = b$ and a multigrid scheme can be used for efficient computation especially if the problem size is big (for instance, millions of cells [Carlson]).

The unknown quantity, which the solver algorithm must find, is the fluid pressure p. In Figure 3, we show ~3033 psi isosurfaces of this solution (pressure across the entire domain varies by only about 5 psi). The actual solution (via `np.linalg.solve`) is rendered in grey, and the three blue surfaces (from narrowest to widest) are the result of applying one, two, and three two-grid cycles, respectively.

As shown, this two-grid solver is converging on the true solution in the vicinity of this isosurface. The multigrid isosurface and the direct solution isosurface become indistinguishable within about ten V-cycles.

Discussion

Testing Setup

In a wrapper script depicted in Figure 4, we used the Python 2.6 module `multiprocessing.Pool` to accelerate the execution of test sets. A dictionary of parameters is constructed for each

distinct possible parameter combination where several parameters of interest are being varied. A process in the pool is then assigned to test each parameter combination. Each pool process then returns a dictionary of dependent variables of interest. Our tests are run on a dual-socket Intel Xeon E5645 (2.40GHz) machine with 32 GB of memory. However, care still must be taken to ensure that the number of processes in the pool is not so high that individual processes run out of memory.

Test Definitions and Results

In Figure 5 a, we show the results of a V-cycle convergence test with our OpenMG solver. Here, we specify the number of repeated 2-grid cycles as an independent variable, and monitor the residual norm as the dependent variable. There were $8^3 = 512$ unknowns, one pre-smoothing iteration, and zero post-smoothing iterations. OpenMG was able to reduce the error at a steady logarithmic rate. The norm used everywhere was the 2-norm.

This contrasts with Figure 5 b, where we show the convergence behavior of the ordinary Gauss-Seidel on its own. Similarly to the method used for Fig. 5 a, we used the number of iterations as the independent variable, and examined the residual norm as the dependent variable. There were $12^3 = 1723$ unknowns, and the test took 43 hours to complete 200,000 iterations. However (for this sample problem), the Gauss-Seidel solver quickly exhausts the high-frequency portions of the solution error, and begins slower work on the low-frequency components.

This frequency-domain effect can be seen more clearly in Figure 5 c, where we show the Fourier transform of the error ($u - u_{apx}$) after different numbers of Gauss-Seidel iterations. A Hann-window smoother with a window width of 28 was applied after the Fourier transform to better distinguish the several curves. For this test, we used a 1D Poisson coefficent matrix and an expected solution vector generated using `np.random.random((N,)).reshape((N,1))`, where `N` was 18,000 unknowns. Because of this method of noise generation (a continuous uniform distribution, or equal probability of all permitted magnitudes at all points in the domain), the pre-generated solution sampled all frequencies unequally, unlike true white noise. This accounts for the initial bell-shaped error in the frequency domain. However, the unequal rate of error-reduction for different frequencies that was observed as iterations were completed is to be expected of iterative solvers, hence their description as "smoother" in the context of multigrid methods. This recalls the argument from a frequency-domain perspective for a multigrid solver [Brandt2].

In Figure 5 d, we examine the effect of this Gauss-Seidel pre-smoother by increasing the number of pre-smoothing iterations from our default value of only one. Dependent variables include the number of V-cycles required to obtain a residual norm of 0.00021, and the time taken by the whole OpenMG solver to arrive at that precision. There were $8^3 = 512$ unknowns and two grid levels, and all restriction and coefficient matrices used were stored in dense format. As expected, increasing the number of pre-smoothing iterations does decrease the number of required V-cycles for convergence, but this does not generally improve the solution time, except in the transition from 3 V-cycles to 2 V-cycles. However, this trend is useful to validate that the smoother is behaving as expected, and might be useful if, in the future, some coarsening method is employed that makes V-cycling more expensive.

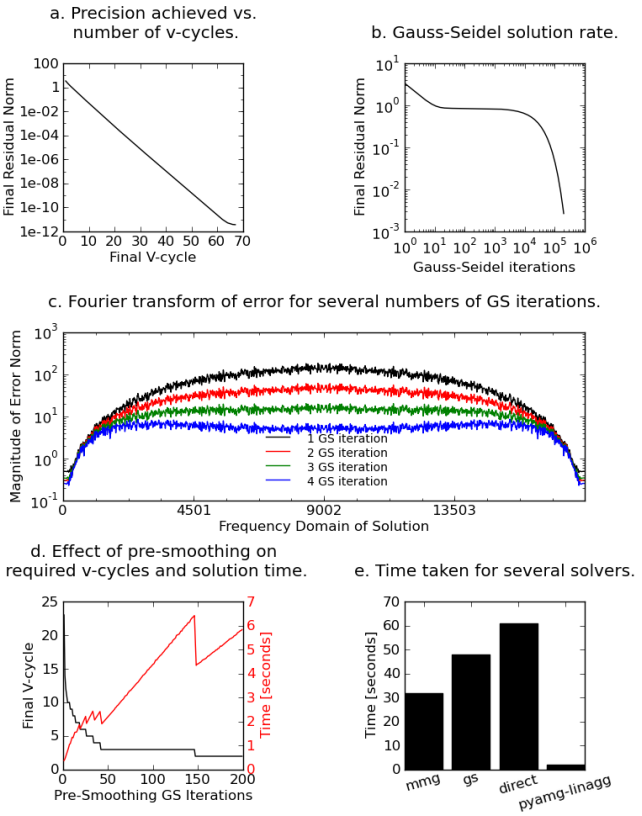


Fig. 5: *Results from explanatory tests. Tests described and interpreted in* Test Definitions and Results.

The Gauss-Seidel (GS) solver's very slow convergence in low-frequency error accounts for the difference in time between it and the OpenMG multigrid (mmg) solver, as shown in Figure 5 e. Here, we compare the running times of several solvers, including PyAMG's smoothed aggregation solver, our own pure-python Gauss-Seidel iterative solver, and the direct solver *np.linalg.solve*. There were $20^3 = 8000$ unknowns, and dense R and A matrices were used for OpenMG. In order to keep the GS bar similar in scale to the other bars in the chart, a relatively high residual norm tolerance of 0.73 was used for both the GS and mmg solvers. However, this tolerance parameter was not an option for the direct solver or PyAMG, both of which achieved very good precision without prompting. The PyAMG solver (pyamg-linagg) used linear aggregation coarsening, and so is not really comparable to our multigrid implementation in this example, but it is included in this plot to demonstrate the speed that can be achieved using optimized multigrid methods with efficient coarsening algorithms. Our own coarsener uses the simple geometric scheme shown in Figure 2, not the more efficient, general, and geometry-agnostic Ruge-Steuben method usually used in algebraic multigrid solvers.

Conclusion and Future Work

OpenMG is an environment for testing new implementations of algebraic multigrid components. While optimized implementations such as PyAMG are more suitable for use as production solvers, OpenMG serves as an easy-to-read and easy-to-modify implementation to foster understanding of multigrid methods. For example, future module improvements could include a parallel

Jacobi iterative solver, a method of generating restriction matrices that is tolerant of a wider range of problem sizes, or operator-based Ruge-Steuben coarsening in addition to the option of stencil-based coarsening. In order to find computational bottlenecks, it might be useful also to add a per-step time profiler.

As open-source software, the code for this project has been posted online under the New BSD license at https://github.com/tsbertalan/openmg. We invite the reader to download the code from this address to explore its unit tests and possible modifications, and to contribute new modules.

REFERENCES

[AGMG] Y Notay, *AGMG,* 2012. [Online]. Available: http://homepages.ulb.ac.be/~ynotay/AGMG.

[Aarnes] J E Aarnes, T Gimes, and K Lie. *An Introduction to the Numerics of Flow in Porous Media using Matlab*, Geometric Modeling, Numerical Simulation and Optimization. 2007, part II, 265-306.

[Bakshi] B R Bakshi and G Stephanopoulos, *Wave-net: a multiresolution, hierarchical neural network with localized learning,* AIChE Journal, vol. 39, no. 1, pp. 57-81, Jan. 1993.

[Bell] N Bell, L Olson, and J Schroder, *PyAMG: Algebraic Multigrid Solvers in Python,* 2011.

[Borzi] A Borzi, *Introduction to multigrid methods.* [Online]. Available: http://www.uni-graz.at/imawww/borzi/mgintro.pdf. [Accessed: 03-Jul-2012].

[Boschitsch] A H Boschitsch and M O Fenley, *A Fast and Robust Poisson-Boltzmann Solver Based on Adaptive Cartesian Grids.,* Journal of chemical theory and computation, vol. 7, no. 5, pp. 1524-1540, May 2011.

[Brandt1] A Brandt, *Rigorous Quantitative Analysis of Multigrid I. Constant Coefficients Two-Level Cycle with L2 Norm*, SIAM Journal on Applied Mathematics, vol. 31, no. 6, pp. 1695-1730, 1994.

[Brandt2] A Brandt, *Multi-Level Adaptive Solutions to Boundary-Value Problems*, Mathematics of Computation, vol. 31, no. 138, pp. 333-390, 1977.

[Brandt3] A Brandt, *Multilevel computations of integral transforms and particle interactions with oscillatory kernels*, Computer Physics Communications, vol. 65, no. 1–3, pp. 24-38, Apr. 1991.

[Brandt4] A Brandt, *AMG and Multigrid Time-Dependence*, Multigrid Methods: Lecture Notes In Mathematics, pp. 298-309, 1987.

[Carlson] E S Carlson, A W Islam, F Dumkwu, and T S Bertalan. *nSpyres, An OpenSource, Python Based Framework for Simulation of Flow through Porous Media*, 4th International Conference on Porous Media and Annual Meeting of the International Society for Porous Media, Purdue University, May 14-16, 2012.

[Denev] J A Denev, F Durst, and B Mohr, *Room Ventilation and Its Influence on the Performance of Fume Cupboards: A Parametric Numerical Study,* Industrial & Engineering Chemistry Research, vol. 36, no. 2, pp. 458-466, Feb. 1997.

[Douglas1] C C Douglas, L I Deng, G Haase, M Liebmann, and R Mckenzie, *Amglab: a community problem solving environment for algebraic multigrid methods.* [Online]. Available: http://www.mgnet.org/mgnet/Codes/amglab.

[Douglas2] C C Douglas, J Hu, M Iskandarani, M Kowarschik, U Rüde, and C Weiss, *Maximizing Cache Memory Usage for Multigrid Algorithms for Applications of Fluid Flow in Porous Media,* vol. 552. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[Dzwinel] W Dzwinel, D. A. Yuen, and K. Boryczko, *Bridging diverse physical scales with the discrete-particle paradigm in modeling colloidal dynamics with mesoscopic features,* Chemical Engineering Science, vol. 61, no. 7, pp. 2169-2185, Apr. 2006.

[Fedorenko1] R P Fedorenko, *The Speed of Convergence of One Iterative Process,* Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki, 1964.

[Fedorenko2] R P Fedorenko, *A relaxation method for solving elliptic difference equations,* Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki, pp. 922-927, 1961.

[Hackbusch] W Hackbusch, Multi-Grid Methods and Applications. Springer, 1985, p. 377.

[Harimi] I Harimi and M Saghafian, *Evaluation of the Capability of the Multigrid Method in Speeding Up the Convergence of Iterative Methods,* ISRN Computational Mathematics, vol. 2012, pp. 1-5, 2012.

[Heckbert] P Heckbert, *Survey of Multigrid Applications*, 1998. [Online]. Available: http://www.cs.cmu.edu/~ph/859E/www/notes/multigrid.pdf. [Accessed: 13-Jun-2012].

[Kameswaran] S Kameswaran, L T Biegler, and G H Staus, *Dynamic optimization for the core-flooding problem in reservoir engineering,* Computers & Chemical Engineering, vol. 29, no. 8, pp. 1787-1800, Jul. 2005.

[Nicolaides] R A Nicolaides, *On Some Theoretical and Practical Aspects of Multigrid Methods,* Mathematics of Computation, 1979. [Online]. Available: http://www.jstor.org/stable/10.2307/2006069. [Accessed: 07-Jul-2012].

[Notay] Y Notay, *An aggregation-based algebraic multigrid method,* Electronic Transactions on Numerical Analysis, vol. 37, pp. 123-146, 2010.

[Trottenberg] U Trottenberg, C W Oosterlee, and A Schüller, *Multigrid*, Academic Press, 2001, p. 631.

[Wesseling] P Wesseling, *An introduction to multigrid methods. 1992,* Wiley, New York, 1991.

[Zapata] G Zapata-Torres et al., *Influence of protonation on substrate and inhibitor interactions at the active site of human monoamine oxidase-a.,* Journal of chemical information and modeling, vol. 52, no. 5, pp. 1213-21, May 2012.

[Zeng] S Zeng and P Wesseling, *Galerkin Coarse Grid Approximation for the Incompressible Navier-Stokes Equations in General Coordinates*, Thesis, 2010.