Proceedings of the 10th

Python in Science Conference

July 11 - 16 • Austin, Texas

Stéfan van der Walt
Jarrod Millman

PROCEEDINGS OF THE 10TH PYTHON IN SCIENCE CONFERENCE

Edited by Stéfan van der Walt and Jarrod Millman.

SciPy 2011
Austin, Texas
July 11 - 16, 2011

ISSN:2575-9752
https://doi.org/10.25080/Majora-ebaa42b7-014

ORGANIZATION

Conference Chairs

JARROD MILLMAN, University of California Berkeley, USA
TRAVIS OLIPHANT, Enthought, Inc., USA

Program Chairs

STÉFAN VAN DER WALT, Applied Mathematics, Stellenbosch University, South Africa
WARREN WECKESSER, Enthought, Inc., USA

Program Committee

FRANCESC ALTED, PyTables.org, Spain
CHRISTOPHER BARKER, Oceanographer, NOAA Emergency Response Division, USA
C. TITUS BROWN, Computer Science and Biology, Michigan State, USA
ROBERT CIMRMAN, Mechanics and New Technologies Research, University of West Bohemia, Plzeň, Czech Republic
DARREN DALE, Cornell High Energy Synchrotron Source, Cornell University, USA
FERNANDO PÉREZ, Neuroscience Institute, UC Berkeley, USA
PRABHU RAMACHANDRAN, Aerospace Engineering, IIT Bombay, India
DHARHAS POTHIN, Texas Water Development Board, Austin, USA
DAN SCHULT, Department of Mathematics, Colgate University, USA
JAMES TURNER, Gemini Observatory, Chile

Tutorial Chair

BRIAN GRANGER, Physics, CalPoly, USA

Student Sponsorship Chair

CHRIS COLBERT, Enthought, Inc., USA

Python and Core Technologies Track Chair

ANTHONY SCOPATZ, Enthought, Inc., USA

Python in Data Science Track Chair

PETER WANG, Streamitive, LLC., USA

Proceedings Reviewers

JACOB VANDERPLAS, Department of Astronomy, University of Washington
JAMES TURNER, Gemini Observatory
YANNICK SCHWARTZ,
ANDY R. TERREL, Texas Advanced Computing Center, University of Texas at Austin
GERT-JAN VAN ROOYEN, Telecommunications, Stellenbosch University
TIZIANO ZITO, Bernstein Center for Computational Neuroscience, Humboldt University of Berlin
MATEUSZ PAPROCKI,
MICHAEL HANKE, Center for Behavioral Brain Sciences, University of Magdeburg
DAVID WARDE-FARLEY, Université de Montréal
JASON GROUT, Department of Mathematics and Computer Science, Drake University
ARIEL ROKEM, Department of Psychology, Stanford University
JACOB BARHAK, ??
TRAVIS VAUGHT, Ben Patterson Management Company, LLC
NICHOLAS PINTO, MIT
TONY YU, Brown University

SCHOLARSHIP RECIPIENTS

CHAD BAKER, University of Texas Austin
DREW CONWAY, New York University
BEN EDWARDS, University of New Mexico
CHRISTOPHER ELLISON, University of California Davis
PAUL IVANOV, University of California Berkeley
THINH LAM, University of Texas Austin
AARON MEURER, New Mexico Tech
MIN RAGAN-KELLEY, University of California Berkeley
JOON RO, University of Texas Austin
SKIPPER SEABOLD, American University
JORDI TORRENTS, University of Barcelona, IESE Business School
DAVID WARDE-FARLEY, Université de Montréal

JUMP TRADING AND NUMFOCUS DIVERSITY SCHOLARSHIP RECIPIENTS

,

CONTENTS

A Technical Anatomy of SPM.Python, a Scalable, Parallel Version of Python

Minesh B. Amin‡*

Abstract—SPM.Python is a scalable, parallel fault-tolerant version of the serial Python language, and can be deployed to create parallel capabilities to solve problems in domains spanning finance, life sciences, electronic design, IT, visualization, and research. Software developers may use SPM.Python to augment new or existing (Python) serial scripts for scalability across parallel hardware. Alternatively, SPM.Python may be used to better manage the execution of stand-alone (non-Python x86 and GPU) applications across compute resources in a fault-tolerant manner taking into account hard deadlines.

Index Terms—fault tolerance, parallel closures, parallel exceptions, parallel invariants, parallel programming, parallel sequence points, scalable vocabulary, parallel management patterns

Prologue

Consider the following acid test for general purpose parallel computing. A serial session is depicted on the left, whereas the session on the right describes its parallel equivalent:

```
                    >>> createVirtualCloud -async
>>> cmdA            >>> cmdA -parallel
>>> cmdB            >>> cmdB -parallel
>>> cmdC            >>> cmdC -parallel
>>> cmdD            >>> cmdD -parallel
```

For example, the command **cmdA** `-parallel` may be a parallel make-like capability, while the command **cmdB** `-parallel` may be a map-reduce capability. At the same time, the command **cmdC** `-parallel` may be a fine grain parallel SAT solver that limits itself to resources with specific incarnations of those utilized by the command **cmdA** `-parallel`. Finally, **cmdD** `-parallel` may be a parallel graph-based analytics capability.

Yet, notwithstanding the prosaic serial session, the equivalent parallel session is in fact predicated on solutions to what were several formally open problems, including (a) defining a scalable vocabulary rich enough to capture the essence of a wide range of parallel problems, (b) the ability to utilize a collection of hardware resources in completely different ways, depending on the nature of parallelism exploited by the respective commands within the same session, and (c) the ability to treat the conclusion of each parallel command as a sequence point, thus guaranteeing that there would be no pending side effects post conclusion.

* *Corresponding author: mamin@mbasciences.com*
‡ *MBA Sciences, Inc*

Introduction

In this paper, we shall review (patented) SPM technology, and the methodology behind it, both predicated on the supposition that parallelism entails nothing more than the *management* of a collection of *serial tasks*, where *management* refers to the policies by which:

- tasks are scheduled,
- premature terminations are handled,
- preemptive support is provided,
- communication primitives are enabled/disabled, and
- the manner in which resources are obtained and released

and *serial tasks* are classified in terms of either:

- Coarse grain – where tasks may not communicate prior to conclusion, or
- Fine grain – where tasks may communicate prior to conclusion.

We shall review how SPM.Python augments the serial Python language to include a suite of parallel primitives, henceforth referred to as parallel closures. These closures represent the sole means by which to express any parallelism when leveraging SPM.Python. Their APIs are designed to be as close to the developer's intent as possible, and therefore easy to relate to. Furthermore, the API of all closures represent the boundary that delineates the serial component (authored and maintained by the developer) from the parallel component (authored and embedded within SPM.Python).

Specifically, the context for and solutions to four formerly open technical problems will be reviewed:

a) decoupling tracking of resources from management of resources,
b) declaration and definition of parallel closures, the building blocks of all parallel constructs,
c) design and architecture of parallel closures in a way so that serial components are delineated from parallel components, and
d) extensions to the general exception handling infrastructure to account for exceptions across many compute resources.

We will illustrate key concepts by reviewing a simple, scalable, fault-tolerant, self-cleaning 60-line Python script that can be used to launch any stand-alone (x86 or GPU) applications in parallel. Appendix A will provide another self-contained Python script that calculates the total number of prime numbers within a given range;

thus, illustrating how any Python module may be parallelized using one of SPM.Python's several built-in parallel closures.

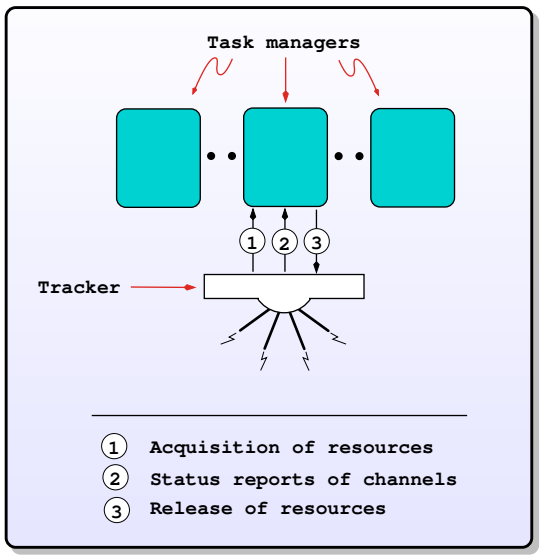

Fig. 1: *In order to facilitate the exploitation of multiple, potentially different, forms of parallelism within a single session of SPM.Python, tracking of resources is decoupled from the management of resources. Therefore, while the tracker is always online, at any moment in time, at most one task manager may be online.*

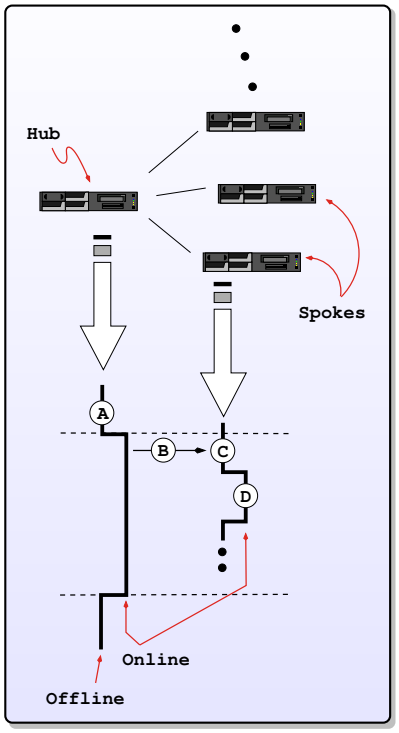

Fig. 2: *Parallel sequence points in terms of online and offline states of the Hub and Spokes. On the Hub, transition to online occurs when a task manager is invoked; transition back to offline occurs when the said manager concludes. On the Spoke, transition to online occurs when a task evaluator is invoked; transition back to offline occurs when the said evaluator concludes.*

Related Work

Traditionally, most parallel solutions in Python have taken the form of: (a) distributed task queues like Celery[1], Parallel Python[2], (b) distributed frameworks like Disco (MapReduce)[3], PaPy (parallel pipelines)[4], or (c) low-level wrappers around HPC libraries like MPI[5], PVM[6]. In sharp contrast, SPM.Python is a single runtime environment that provides access to multiple different, distinct forms of parallelism by way of parallel primitives called parallel closures.. Furthermore, these closures are architectured to be as close to the developer's intent as possible – in terms of, say, either coarse or fine-grain DAG/templates/hybrid flows, and lists – while de-emphasizing low-level error-prone concepts like locks, threads, pipes, mutexes and semaphores.

Tracking of Resources

In SPM.Python, compute resources are tracked independently of any task manager. In operation, any task manager may come online and request resources from the tracker. The task manager would then manage the execution of tasks using the acquired resources, and when done, go offline (i.e. release the resources back to the tracker). Another task manager may subsequently come online, obtain the same or different resources used by a previous task manager, and utilize those resources in a completely different way. In other words, the task managers can be implemented more simply because each manager would have a more narrowly focused discrete policy. Furthermore, a tight coupling can be established between a task manager and the communication closures, thus preventing a whole class of deadlocks from occurring. More details can be found at [7].

Declaration and Definition of Parallel Closures

In SPM.Python, parallel closures are the building blocks of all parallel constructs, and provide the sole means by which one may express how serial components interact with parallel components. The interactions may take place in one of two contexts (a) when creating, submitting, and evaluating tasks, and (b) when creating and processing messages. However, any usage of a parallel closure within any resource is predicated on a successful, safe, asynchronous and race-free declaration and definition across many compute resources. We solve this problem by augmenting the traditional concept of serial sequence points by introducing the notion of *offline* and *online* states. The declaration and definition of parallel closures is only permitted when the resource in question is in the *offline* state – a state when SPM.Python guarantees that the serial component of the resource may not communicate with the outside world and vice versa. So, all resources start off *offline* (Ⓐ, Ⓒ).

On

the Hub, the transition to the *online* state occurs when a parallel (task manager) closure is invoked; the transition back to the *offline* state does not occur until just before the closure concludes. On the Spoke, SPM.Python receives a task from the Hub while *offline* (Ⓒ), and at which point any preloading of Python modules is performed. One side effect of this preloading may be the declaration and definition of parallel closures. Next, the transition to *online* is made before SPM.Python invokes the callback (Ⓓ) for the task; the transition back to *offline* does not occur until just after the callback concludes.

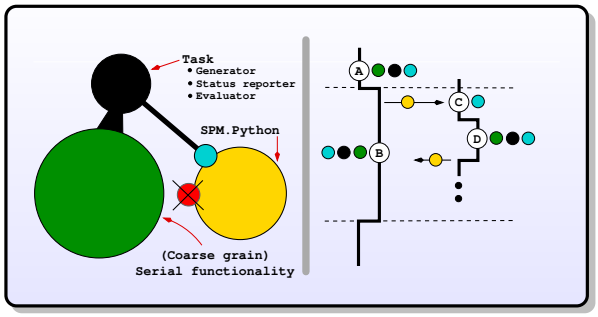

Fig. 3: The architectural and runtime perspectives of coarse grain task manager closures. Note that such closures do not permit tasks to communicate prior to conclusion.

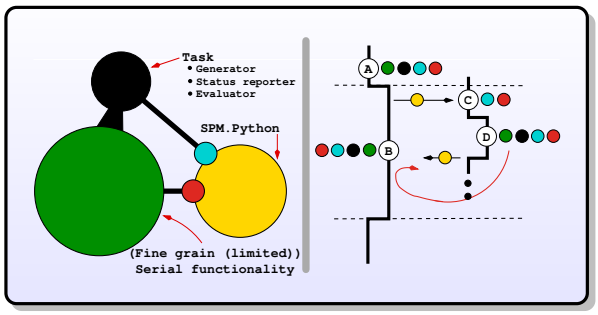

Fig. 4: The architectural and runtime perspective of fine grain (limited) task manager closures. Note that such closures permit tasks to communicate only with the Hub.

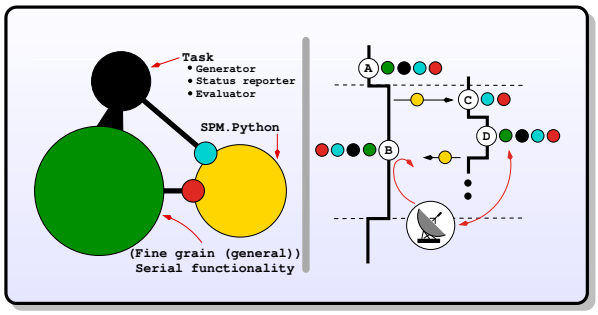

Fig. 5: The architectural and runtime perspective of fine grain (general) task manager closures. Note that such closures permit communication among Spokes and, if appropriate, with the Hub.

Types of Fault-Tolerant Parallel Closures

A key tenet of the serial software ecosystem is the asymptotic parity between the serial compute resources available to the developers and the end-users, which makes possible the reporting, reproduction, and resolution of bugs.

With parallel software, this most fundamental of tenets is violated; software engineers need to be able to produce high-quality parallel software in what is an essentially serial environment, yet be able to deploy the said software in a parallel environment.

SPM.Python addresses this dichotomy by offering a suite of easy to relate to parallel closures. These closures enable the prototyping, validation, and testing of parallel solutions in an essentially serial-like development environment, yet are scalable when exercised in any parallel environment.

Coarse grain

Exploiting coarse grain parallelism is anchored around the asynchronous declaration and definition of a parallel (task manager) closure (●) across all resources (Hub and Spokes). On the Hub, this is depicted by (Ⓐ). On the Spokes, this is only possible prior to the evaluation of a task, as depicted by (Ⓒ), when the modules may be preloaded.

Next,

existing serial functionality (●) may be parallelized by having it be augmented with serial code (●) to:

- generate and submit tasks to the parallel task manager, and handle status reports/exceptions from tasks, as depicted by (Ⓑ)
- evaluate tasks, as depicted by (Ⓓ)

Finally, actual parallelism can commence by invoking the task manager on the Hub with a collection of tasks, and a handle to a pool of resources (Ⓑ). The backend of the task manager would ensure the concurrent scheduling and evaluation of tasks across all Spokes. Note that coarse grain task manager closures do not permit the usage of any form of communication closures (⊗).

Fine grain (limited)

Fine grain (limited) parallelism augments the coarse grain parallelism by allowing tasks to communicate with the Hub prior to their conclusion. The closures (●) that would permit such communication must be declared and defined following the steps reviewed for parallel task manager closures (●).

However,

in order to avoid the vast majority of deadlocks, the communication closures must be designed in a way so that all communication is initiated by the Spokes; the Hub must be restricted to processing incoming messages from the Spokes, and, if appropriate, replying to them.

Fine grain (general)

Fine grain (general) parallelism augments the fine grain (limited) parallelism by permitting communication among Spokes.

However,

in order to avoid the vast majority of deadlocks, the fine grain (general) task manager closures must treat all Spokes under their control as a single unit; the premature termination of any Spoke must be treated as a premature termination of all Spokes.

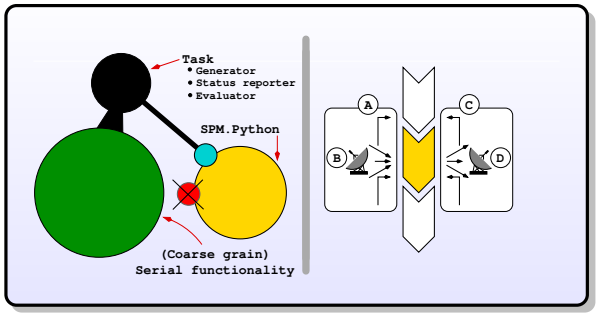

Fig. 6: *The architectural and runtime perspectives of coarse grain parallel exceptions.*

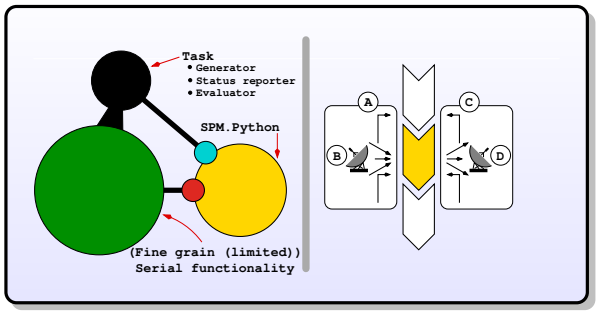

Fig. 7: *The architectural and runtime perspectives of fine grain (limited) parallel exceptions.*

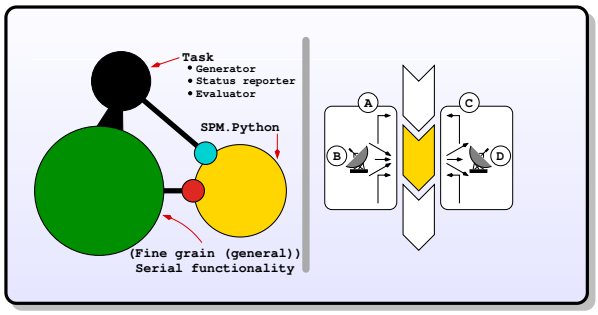

Fig. 8: *The architectural and runtime perspectives of fine grain (general) parallel exceptions.*

Types of Fault-Tolerant Parallel Exceptions

To quote Wikipedia, "exception handling is a construct designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution".

The ability to throw and catch exceptions forms the bedrock of the serial Python language. We will review details of how we extended the basic serial exception infrastructure to account for exceptions that may occur across many compute resources.

Our solution is predicated on the notion that parallel task managers must take ownership of how serial exceptions are handled across all resources under their control. Therefore, unlike in the serial world, the parallel exception handling infrastructure must be customized for each type of parallel task manager.

Coarse grain

Exception handling, as traditionally defined in the serial context, is designed to handle the change in the normal flow of program execution ... a rather straightforward concept given that there is only one call-stack.

However,

when exploiting parallelism, the normal flow of program execution involves multiple resources and, therefore, multiple call-stacks need to be processed in a fault-tolerant manner. Furthermore, in order to enforce various forms of parallel invariants, we need an ability to throw exceptions at any resource, but which may only be caught by the Hub.

Stated another way, in order to make our problem tractable in the context of coarse grain parallelism:

- on a Spoke, any uncaught/uncatchable exception must be treated and reported as final status of the task. Therefore, an exception free execution on the Hub would result in the normal unrolling of the call-stack at the Hub, as depicted by (Ⓐ, Ⓑ).
- on the Hub, any uncaught exception from any callbacks invoked by the task manager must result in the forcible termination and, if appropriate, relaunching of Spokes, as depicted by (Ⓒ, Ⓓ).

Fine grain (limited)

The exception handling infrastructure in the context of fine grain (limited) parallelism may be identical to that for coarse grain parallelism provided stale replies generated by the Hub and meant for some Spoke can be filtered out at the Hub itself.

Fine grain (general)

Given that fine grain task manager closures treat all Spokes as a single unit:

- on a Spoke, any uncaught/uncatchable exception must be treated and reported as final status of all the Spokes. Therefore, an exception free execution on the Hub and all Spokes would result in the normal unrolling of the call-stack at the Hub, as depicted by (Ⓐ, Ⓑ).
- any uncaught/uncatchable exception from any callbacks invoked by the task manager or by any Spoke should result in the forcible termination and, if appropriate, relaunching of Spokes, as depicted by (Ⓒ, Ⓓ).

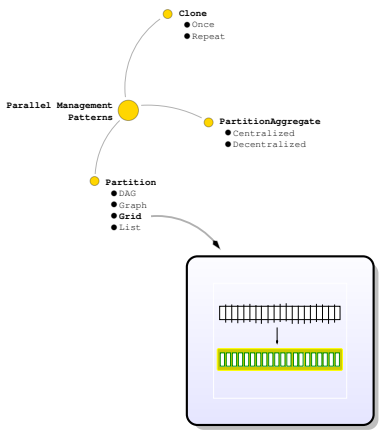

Fig. 9: *Partition/List Parallel Management Pattern.*

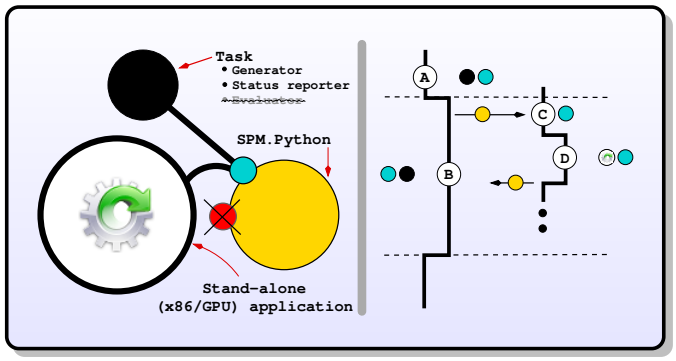

Fig. 10: *The architectural and runtime perspective of launching stand-alone applications in parallel using SPM.Python.*

Problem Decomposition

Understanding the nature of any parallel problem is key to determining the appropriate solution. Parallel Management Patterns (PMPs) provide a framework for decomposing and authoring scalable, fault-tolerant parallel solutions. In other-words, if the end goal is some parallel application, PMPs enable us to classify the journey to the end goal in terms of the nature of parallelism to be exploited, while parallel closures provided by SPM.Python enable us to express the parallelism implied by any PMP.

For the purpose of illustration, we shall review an implementation of the Partition/List PMP, a pattern that captures the essence of how to execute a list of tasks across many compute resources in a fault-tolerant manner.

Problem Statement

Our goal is to invoke the SPM coprocess API:

```
spm.util.coprocess.shell.policyA(cmd     = ...,
                                 timeout = ...,
                                )
```

across multiple resources. We shall capture the context - in the form of arguments needed, and the final result to be returned - of each execution by way of tasks. To that end, we shall augment the aforementioned serial functionality by authoring a scalable, parallel, fault-tolerant Python script made up of the following components:

- declaration of a (task manager) closure at the Hub,
- definition of tasks, processing of status reports, and invocation of task manager at the Hub.

As an aside, note that the backend of our closure will evaluate the task on our behalf ... a process that is rather straightforward given that we would be invoking a built-in method (`shell.policyA`).

Ⓐ Task manager: Declaration and Definition

In order to create (declare and define) an instance of the task manager, we require the Hub to be offline to in order to avoid various types of parallel race conditions. This invariant is captured by the decorator statements on lines 1 and 2.

A natural point in time to perform this initialization step would be when loading the module containing the statements prior to actual usage. In other words, initialization should occur when the file containing `__init` method is imported by the Python interpreter.

The arguments for creating our instance bear highlighting. Each instance of any closure must be unique within a module; hence, the unique string as argument 1. Furthermore, all instances of our closure are defined in terms of two stages. Of these, functionality for stage 1 is expected via a callback; hence argument 2 (`__taskStat`).

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def __init():
4    return spm.pclosure.macro.papply.list.grainCoarse.\
5           policyA.defun(signature = 'signature::Hub',
6                         stage1Cb  = __taskStat,
7                        );
8
9  __pc = __init();
```

```
r"""
 task<list>          :: struct {
   # SPM component ...
   spm               :: struct {
     meta            :: struct {
       label         :: scalar<stringSnippet> = deferred;
       api           :: scalar<ApiMethod>     = deferred;
       apiArgs       :: dict<string,mixed>    = deferred;
       timeout       :: scalar<timeout>       = deferred;
     };

     core            :: struct {
       relaunchPre   :: scalar<bool>          = None;
       relaunchPost  :: scalar<bool>          = None;
       nameHost      :: scalar<auto>          = None;
       whoAmI        :: scalar<auto>          = None;
     };

     stat            :: struct {
       exception     :: scalar<auto>          = None;
       returnValue   :: scalar<record>        = None;
     };
   };
   # non-SPM component ...
 };
 """
```

Fig. 11: *Typedef for the definition of list of tasks.*

```
Exception
 |
 +-- SPMError
      |
      +-- SPMTaskDropped
      +-- SPMTaskLoad
      |    |
      |    +-- SPMTaskLoadUncaught
      |    +-- SPMTaskLoadFatal
      |    +-- SPMTaskLoadKill
      |    +-- SPMTaskLoadTimeout
      |
      +-- SPMTaskEval
           |
           +-- SPMTaskEvalUncaught
           +-- SPMTaskEvalFatal
           +-- SPMTaskEvalKill
           +-- SPMTaskEvalTimeout
```

Fig. 12: *Hierarchy of (parallel) SPM exceptions.*

Ⓐ Task manager: Population and Invocation

Our goal in the function **main** is to be able to invoke the task manager (line 18). However, before doing so, we must populate it with the tasks to be executed. This is achieved by submitting our tasks by way of the API **stage0**, as shown in lines 11 through 16.

Once our task manager is invoked, the Hub transitions to the online state. The transition back to offline does not occur until just prior to the conclusion of the invocation.

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def main(pool,
4           taskApi,
5           taskApiArgs,
6           taskTimeout):
7    # Initialize 'stage0'.
8    __pc.stage0.init.main(typedef = ...); # See Figure 11.
9    hdl = __pc.stage0.payload.tie();
10   # Create a list of tasks
11   for entry in taskApiArgs:
12     hdl.spm.meta.label   = '***'; # Not interested.
13     hdl.spm.meta.api     = taskApi;
14     hdl.spm.meta.apiArgs = entry;
15     hdl.spm.meta.timeout = taskTimeout;
16     hdl.Push();
17   # Invoke the pmanager
18   __pc.stage0.event.manage(pool                 = pool,
19                            nSpokesMin           = ...
20                            nSpokesMax           = ...
21                            timeoutWaitForSpokes = ...
22                            timeoutExecution     = ...
23                            );
24   return;
```

Ⓑ Task manager: (Final) Status Reports

The method **__taskStat** (used when declaring and defining our closure) is automatically invoked by the task manager to process the status report of any task. Note that this method is invoked while the Hub is in the online state. This invariant is captured by the decorator statements on lines 1 and 2.

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def __taskStat(pc):
4    try:
5      hdl         = pc.stage1.payload.tie();
6      returnValue = hdl.spm.stat.returnValue;
7      if (returnValue.Has(attr = 'stdOut')):
8        print("\tstdOut    : %s", returnValue.stdOut);
9      if (returnValue.Has(attr = 'stdErr')):
10       print("\tstdErr    : %s", returnValue.stdErr);
11     if (returnValue.Has(attr = 'stdOutErr')):
12       print("\tstdOutErr: %s", returnValue.stdOutErr);
13   except (SPMTaskDropped,
14           SPMTaskLoad,
15           SPMTaskEval,
16           ), (hdl,):
17     pass;
18
19   return (pc.stage1.event.done(),
20           None,
21           )[-1];
```

Ⓒ Task manager: Preloading of Python modules

Ⓓ Task manager: Task Evaluation

As each task involves the invocation of one of the built-in spm coprocess methods, we do not need to define any method to accept and evaluate any task. Instead, our task manager will automatically evaluate our tasks on the Spokes, and return the respective status reports to the Hub.

```
  GNU/Linux [] spm.3.110602.trial.A.python
                        (Trial Edition)
               Spm.Python 3.110602 / Python 2.6.6
                 [GCC 4.4.3 (64 bit) on linux2]

                             NOTE
                 >>>>      Trial period ends at        <<<<
                 >>>> 24:00 hrs (Pacific Standard Time) <<<<
                 >>>>        July 28,      2011         <<<<

     Type "help", "copyright", "credits", "license" or "spm.Api()" for more information.
            Type "spm.DemoExtract(dirname = ...)" to extract demo scripts.

           Please visit www.mbasciences.com for the latest and growing
         collection of scripts and technical briefs classified in terms of
                         parallel management patterns.

● >>> import pool
  >>> import demo
  >>> taskApi     = spm.util.coprocess.shell.policyA;
  >>> taskApiArgs = \
                    [
                      dict(cmd     = "echo `hostname`",
                           timeout = spm.util.timeout.after(seconds = 1)),

                      dict(cmd     = "echo `hostname`",
                           timeout = spm.util.timeout.after(seconds = 2)),
                    ];
  >>> taskTimeout = spm.util.timeout.after(seconds = 10);
✓ >>> demo.main(pool        = pool.intraAll(),
                taskApi     = taskApi,
                taskApiArgs = taskApiArgs,
                taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut    : lusaka
          stdOut    : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✓ >>> demo.main(pool        = pool.intraOnePerServer(),
                taskApi     = taskApi,
                taskApiArgs = taskApiArgs,
                taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut    : lusaka
          stdOut    : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✈ >>> demo.main(pool        = pool.inter(),
                taskApi     = taskApi,
                taskApiArgs = taskApiArgs,
                taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut    : lusaka
          stdOut    : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✈ >>> demo.main(pool        = pool.interOnePerServer(),
                taskApi     = taskApi,
                taskApiArgs = taskApiArgs,
                taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut    : lusaka
          stdOut    : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
  >>> exit()
  GNU/Linux []
```

Fig. 13: *A typical parallel session of SPM.Python.*

The automatic evaluation of our tasks is aided by the typedef used when initializing **stage0** (at the Hub). Specifically, all Spokes end up executing the pseudo-code:

```
try:
  task.spm.stat.returnValue = apply(task.spm.meta.api,
                                    (),
                                    task.spm.meta.apiArgs);
except e:
  task.spm.stat.exception   = str(e);

return task;
```

SPM.Python Session

Having reviewed our parallel application, we will conclude by describing an actual SPM.Python session. We start off by importing the **pool** module (●). Next we import our parallel application **demo**, and run our application four times before exiting, as illustrated by ✓ and ✈.

The

first two times (marked ✓), we limited ourselves to cores from the server running the Hub. **intraOnePerServer** refers to one unique core on the server.

The

second two times (marked ✈), we limited our selves to cores from potentially different servers. **interOnePerServer** refers to one unique core from each server.

The

fact that the results produced are identical should not be a surprise since our code is a function of a handle to a pool, and not its content. In other words, user code remains unchanged despite having selected four different sets of resources.

Note that, notwithstanding our rather small script, our solution is not only fault-tolerant (thanks to closures), self-cleaning (thanks to robust timeout support), but also robust (thanks to the efficient manner by which parallel invariants are enforced). So, once we have tested our solution in a serial-like environment, we can be sure our solution can be deployed on any cluster. See [8] for a comprehensive list of problem decomposition using other PMPs including self contained and equally powerful examples.

Conclusion

In this paper, we reviewed the technical anatomy of SPM.Python, a scalable parallel version of the serial Python language. We began with a prologue presenting the acid test for general purpose parallel computing. Next, we described the solution to four formerly open technical problems, namely the decoupling of tracking of resources from management of resources; the declaration and definition of parallel closures; the design and architecture of parallel closures that delineate serial and parallel components; and fault-tolerant parallel exception handling. We concluded by illustrating how a parallel problem, once classified in terms of a Parallel Management Pattern (PMP), can be decomposed and easily expressed in terms of SPM.Python's parallel closures.

References

[1] Celery, celeryproject.org
[2] Parallel Python, www.parallelpython.com
[3] Disco, discoproject.org
[4] PaPy, code.google.com/p/papy
[5] PyMPI, mpi4py.sourceforge.net
[6] PyPVM, pypvm.sourceforge.net
[7] Minesh B. Amin,. *Resource Tracking Method and Apparatus*, United States Patent #: 7,926,058 B2, April 12, 2011.
[8] Parallel Management Patterns, www.mbasciences.com/pmp.html

Appendix A

Figures 14 through 16 highlight the manner by which any module can be parallelized using SPM.Python. Specifically, a serial module that computes number of prime numbers within a given range (Figure 14) is parallelized by introducing two wrappers as depicted by Figure 15 (for Spoke), and Figure 16 (for Hub). Recall that SPM.Python has built-in support for multiple different and distinct forms of parallelism. However, for our purpose, we are only interested in the closure that executes a list of tasks in parallel.

```
#
# Serial module to compute prime numbers
#
def am(n):
    #
    # Came across this algo on the internet.
    #
    import math
    n = abs(n)
    i = 2
    while i <= math.sqrt(n):
        if n % i == 0:
            return False
        i += 1
    return True

def ctRange(nMin, nMax):
    if ((nMin % 2) == 0):
        nMin = nMin + 1; # Focus on odd numbers (!)

    nprimes = 0;
    while (nMax > nMin):
        if (am(nMin)):
            nprimes += 1;

        nMin += 2;

    return nprimes;
```

Fig. 14: *Spoke: Original 'serial' module that computes the number of prime numbers given a range.*

```
#
# Compute the number of primes between 3 and 502347 ...
#

@spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amSpoke)
def taskEval(pc):
    from serial import ctRange as ctRange;

    hdl                        = pc.stage2.payload.tie();
    hdl.spm.stat.returnValue = ctRange(nMin = hdl.nMin,
                                       nMax = hdl.nMax,
                                       );

    return (pc.stage2.event.done(),
            None,
            )[-1];
```

Fig. 15: *Spoke: Wrapper around serial functionality. The wrapper is automatically invoked by SPM.Python based on the content of the task's 'spm' sub-structure.*

```
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __init():
    # Create parallel closure (task manager) of the type
    # we are interested in (coarse grain parallel list manager) ...
    return spm.pclosure.macro.pinterp.list.grainCoarse.policyA.defun \
             (signature = 'is_prime::main', # Something unique to module.
              stage1Cb  = __taskStat,
             );

__pc      = __init();
__nprimes = 0;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __taskStat(pc):
    # Callback for incoming status reports ...
    try:
        global __nprimes;

        hdl        = pc.stage1.payload.tie();
        __nprimes += hdl.spm.stat.returnValue;
        print('  --> Rolling count of (# of prime numbers) :: %d' \
              % (__nprimes,));
    except (SPMTaskDropped,
            SPMTaskLoad,
            SPMTaskEval,
            ), (hdl,):
        pass;

    return (pc.stage1.event.done(), # Explicitly let the backend
                                    # know we are done;
            None,
            )[-1];

#
# Compute the number of primes between 3 and 502347
# by dividing the range into 'nBuckets' ...
#
import os;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def main(pool,
         nBuckets = 10,
         ):
    # Initialize 'stage0'.
    global __nprimes;

    assert(nBuckets >= 1);
    __pc.stage0.init.main(typedef = \
        r"""
        task<list>::struct {
          #
          # SPM component ...
          #
          spm::struct {
            meta::struct {
              label         ::scalar<stringSnippet> = deferred;
              path          ::tuple<string>         = deferred;
              modulePreload::tuple<string>          = deferred;
              module        ::scalar<stringSnippet> = deferred;
              timeout       ::scalar<timeout>       = deferred;
            };

            core::struct {
              relaunchPre  ::scalar<bool>           = None;
              relaunchPost ::scalar<bool>           = None;
            };

            stat::struct {
              exception    ::scalar<auto>           = None;
              returnValue  ::scalar<auto>           = None;
            };
          };

          #
          # non-SPM component ...
          #
          nMin             ::scalar<auto>           = deferred;
          nMax             ::scalar<auto>           = deferred;
        };
        """);
    __nprimes = 0;                          # Always reset counter.
    hdl       = __pc.stage0.payload.tie(); # Handle to the payload.
    nMin      = 2;
    for ct in range(0, nBuckets):
        # Initialize 'spm' component so that Spokes know what to
        # preload ...
        hdl.spm.meta.label         = '***';
        hdl.spm.meta.path          = \
            (os.path.dirname(__pc.meta.module.srcDir),);
        hdl.spm.meta.modulePreload = ('is_prime',);
        hdl.spm.meta.module        =  'is_prime';
        hdl.spm.meta.timeout       = \
            spm.util.timeout.after(seconds = 10);
        hdl.nMin                   = nMin; nMin += ((502347) / nBuckets);

        if (ct == (nBuckets - 1)):
            hdl.nMax = 502347;
        else:
            hdl.nMax = nMin;

        hdl.Push();

    #
    # Invoke the pmanager ...
    #
    __pc.stage0.event.manage \        (pool                  = pool,
        nSpokesMin           = spm.env.const.default,
        nSpokesMax           = spm.env.const.default,
        timeoutWaitForSpokes = spm.util.timeout.after(seconds = 2),
        timeoutExecution     = spm.util.timeout.after(seconds = 300),
        );

    return;
```

Fig. 16: *Hub: Creation/population/invocation of parallel (task manager) closure. The backend of the closure, once invoked, would execute as many tasks in parallel as possible using resources within the* **pool**.

Fitting and Estimating Parameter Confidence Limits with Sherpa

Brian Refsdal‡*, Stephen Doe‡, Dan Nguyen‡, Aneta Siemiginowska‡

Abstract—Sherpa is a generalized modeling and fitting package. Primarily developed for the Chandra Interactive Analysis of Observations (CIAO) package by the Chandra X-ray Center, Sherpa provides an Object-Oriented Programming (OOP) API for parametric data modeling. It is designed to use the forward fitting technique to search for the set of best-fit parameter values in parametrized model functions. Sherpa can also estimate the confidence limits on best-fit parameters using a new confidence method or using an algorithm based on Markov chain Monte Carlo (MCMC). Confidence limits on parameter values are necessary for any data analysis result, but can be non-trivial to compute in a non-linear and multi-parameter space. This new, robust confidence method can estimate confidence limits of Sherpa parameters using a finite convergence rate. The Sherpa extension module, pyBLoCXS, implements a sophisticated Bayesian MCMC-based algorithm for simple single-component spectral models defined in Sherpa. pyBLoCXS has primarily been developed in Python using high-energy X-ray spectral data. We describe the algorithm including the features for defining priors and incorporating deviations in the calibration information. We will demonstrate examples of estimating confidence limits using the confidence method and processing simulations using pyBLoCXS.

Index Terms—modeling, fitting, parameter, confidence, mcmc, bayesian

Introduction

Sherpa is an extensible, general purpose modeling and fitting application written in Python and Python C/C++/FORTRAN extensions. Originally developed for users of NASA's Chandra X-ray Observatory, Sherpa has also been used to analyze data from other astronomy missions, and even non-astronomical data. Sherpa provides Python data classes to encapsulate various types of astronomical data sets (spectra, images, time series, light curves). But to provide the greatest flexibility, Sherpa is also designed to read in any data set that can be represented as a collection of arrays. From its first version, Sherpa has been designed to help scientists analyze data from many different sources, and to be extensible by scientific users, to help solve new problems.

Sherpa's main task is to help users fit parametrized models to their data. Sherpa provides a library of physical and mathematical models, also written in Python. These models can be combined in arbitrarily complex expressions, that are interpreted by the Python parser; such expressions can include Sherpa models, arithmetic operators, models written by users in Python, and even other Python functions.

* *Corresponding author: brefsdal@cfa.harvard.edu*
‡ *Smithsonian Astrophysical Observatory*

To compare models and data, Sherpa includes statistics such as least-squares, chi-squared based on Gaussian statistics, and maximum likelihood based on Poisson statistics. As model parameters are varied, Sherpa can then measure whether the new model parameter values improve or worsen the fit to the data, using one of these statistics. Sherpa also provides functions to search parameter space for the set of best-fit parameter values: a non-linear least squares using the Levenberg-Marquardt algorithm [lm]; and the Nelder-Mead simplex algorithm [nm].

However, the analysis is not complete when a user has found a set of best-fit model parameter values consistent with the data. Because of measurement errors and statistical noise, there is some probability distribution in parameter space of parameter values that are consistent with the data. If the user can examine this probability distribution in some way, the user can determine how well the best-fit parameter values are constrained. Such constraints are often summarized as confidence limits, stating that parameters are known to a certain level of confidence [avn1976]. For example, after an examination of parameter space, the user might determine that, for a model having temperature as a parameter, a best-fit temperature of 1.2 keV has 90% confidence limits of +0.2 keV, -0.4 keV. Meaning that if the observation and resulting fit were replicated 1000 times, then in 900 trials the best-fit temperature would be between 1.4 and 0.8 keV. The narrower the confidence limits, the better the constraints on the best-fit parameter value.

In this paper, we describe several methods we make available to Sherpa users and Python programmers to put confidence limits on fitted parameter values. We discuss a confidence limit function included in Sherpa, that examines parameter space near the local minimum representing best-fit parameter values, and that returns the desired confidence limits. We provide an interface to this function allowing users to add their own statistic and fitting functions, making this function available to SciPy users. We also discuss the use of simulations in Sherpa to derive limits from distributions of fitted parameter values after many simulations and fits, and show an example to derive both flux and flux errors from a model fitted to spectral data. Finally, we present a new Python module providing a Bayesian approach to deriving fitted parameter values and confidence limits: pyBLoCXS, a new importable Python module, that allows use of prior distributions on model parameters via extensions to Sherpa statistics classes.

Data Preparation

Sherpa provides native Python data classes that encapsulate 1-D and 2-D data sets, i.e., (x, y) and (x_1, x_2, y) respectively. These

classes can be extended to contain data of higher dimensionality. In all data classes, the x-array(s) are considered to be the independent variable(s), and the y-array is considered the dependent variable. Any model to be fit to the data must take the form f(x, p), where x is the collection of x-array(s) taken from the data, and p is the array of parameter values that may be varied by the Sherpa fitting function. The model returns an array of model values that are compared to the data's y-values.

Sherpa data classes also include error bars on the dependent variable; these error bars are assumed to be symmetric. (Error bars on the independent x-array(s) are not yet supported, and so are not assumed to be significant.) The data classes can contain both statistical and systematic errors; if both are present, they are added in quadrature to provide the error bars on the data. Systematic errors come from measurements; if not provided along with the data, Sherpa assumes the systematic errors are zero. If statistical errors are not measured and provided with the data, then Sherpa can estimate Gaussian errors as needed in χ^2 fitting; or, the user can use one of the maximum likelihood statistics, in which Poisson statistics are assumed; or, the user can do a simple least-squares fit to the data.

Fitting Models to Data

Sherpa models are assumed to be parametrized functions f(x, p), where x is the collection of x-array(s) from the data, and p is the array of model parameters. When the model is calculated, the return value is an array of predicted data values that can be directly compared to the observed data values (that are contained in the data's y-array).

Sherpa includes a model description syntax for users to build composite models that are arbitrarily complex. To support such a powerful feature, the user is not required to provide a function to calculate the derivatives. For least squares fitting using Levenberg-Marquardt, Sherpa estimates the gradient using forward difference approximation (LMDIF) and backward difference approximation if the fit is at an upper parameter boundary. An estimate of the gradient is not needed for fitting using simplex, only the fit statistic value is required.

In some cases, the fit parameters are not necessarily independent and identically distributed (i.i.d.) and correlations between parameters are present. This can lead to non-linear effects and complex parameter spaces, see Figure 1. We present a method designed to calculate confidence intervals in non-linear regression and a Bayesian method to sample the posterior probability distribution.

Confidence Intervals

The optimizer's search for the best-fit parameters stops when the fit statistic or error function has reached an optimal value. For least squares, the optimal value is when the sum of squared residuals is a minimum. For the maximum likelihood estimator, the optimal value is found when the log-likelihood is a maximum. Once the best-fit parameter values are found, users typically determine how well constrained the parameter values are at a certain confidence level by calculating confidence intervals for each parameter. The confidence level is a value of the fit statistic that describes a constraint on the parameter value. The confidence interval is the range that likely contains the parameter value at which the fit statistic reaches its confidence level while other parameters reach new best-fit values. See Figure 2. For example, consider

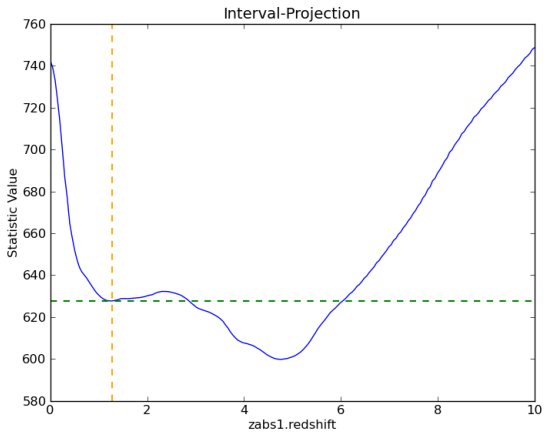

Fig. 1: *A local minima*

calculating the confidence intervals at a value of $\sigma = 1$, or 68% confidence. If the observed data is re-sampled and the model is fit again with new data, there would be a 68% chance that the confidence intervals would constraint the parameter value. The narrower the confidence interval, the more the model parameter value becomes accurately constrained.

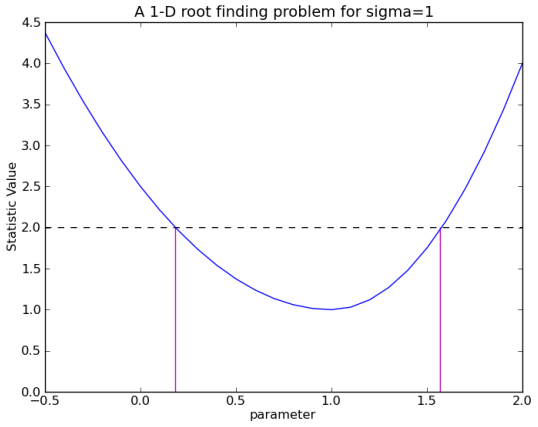

Fig. 2: *A closeup view of a local minima*

In the neighborhood of the fit statistic minimum, the multi-dimensional parameter space can take the shape of an asymmetric paraboloid. The confidence intervals are calculated for each selected parameter independently by viewing the parameter space along the current parameter's dimension. This view can be represented as a 1-D asymmetric parabola, see Figure 2. Suppose that x_0 represents a parameter's best-fit value. Its associated confidence intervals are represented as $x_0 \pm^{\delta_1}_{\delta_2}$ where $\delta_1 \neq \delta_2$ in non-linear parameter spaces, so each confidence limit must be calculated independently. In turn, the statistic value should equal an amount of σ^2 (where σ represents the degree of confidence) at each confidence interval $x_0 + \delta_1$ and $x_0 - \delta_2$ as other parameters vary to new best-fit values. The degree to which the confidence limit is bounded can be characterized by the shape of the well in a multi-dimensional parameter space. A well that is a deep-and-narrow corresponds to a tight confidence interval while a well that is shallow-and-broad represents a wider confidence interval.

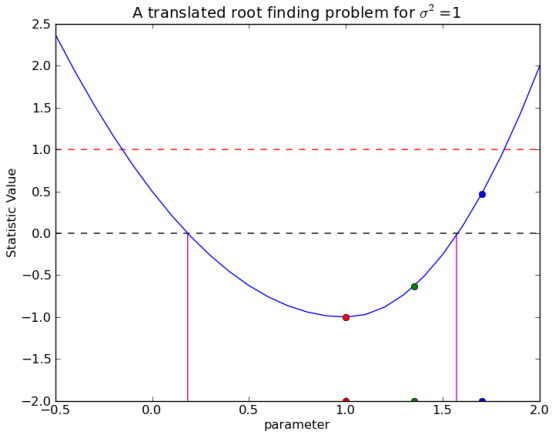

Fig. 3: *The confidence intervals can be reduced to a root solving problem by translating the y-axis by an amount equal to σ^2 and selecting points along the fit statistic curve.*

Method for Determining Confidence

Calculating the confidence interval for a selected fit parameter can be transformed into a one dimensional root finding problem with the correct coordinate translation. By simply translating the parameter dimension by an amount equal to σ^2, the confidence intervals now become x-axis intercepts in the parameter dimension. This is an important step in the algorithm because a change in sign will bracket the root. The green and blue points in Figure 3 effectively bracket the requested confidence limit.

Method for Selecting Abscissae

Sherpa's confidence method uses Müller's root finding method to calculate the confidence intervals given three points. Sherpa begins at the best-fit value and calculates points along the fit statistic curve using the covariance, if available, and the secant method. Müller's method is the a good algorithm for finding the root of a curve that is approximated by a parabola near the minimum. We argue that the function curve can be approximated by parabola given that the function can be represented as a Taylor's series. The leading term in series expansion is quadratic since the gradient of the statistic curve can be ignored near the minimum.

The confidence method assumes that the parameter values are located in a minimum approximated by a parabola, that the best-fit is sufficiently far from any parameter boundaries, and that the bracketed parameter interval is larger than the requested machine tolerance.

A Bayesian Approach to Confidence

Fitting Poisson data with χ^2 can lead to biased results. Using likelihood statistics like cash or C do not introduce bias, but lack simple tests for characterizing how well the model fits the data. Such likelihood statistics often require additional methods to validate model selection and to determine "goodness-of-fit". Such methods involve sampling from the posterior probability distribution. Sherpa includes fit statistics derived from the likelihood and complimentary optimization methods, but on its own Sherpa does not include the means to calculate the posterior.

pyBLoCXS is an additional Python module that complements Sherpa to probe the posterior probability and to verify model selection using Bayesian methods. pyBLoCXS is designed to use Markov chain Monte Carlo (MCMC) techniques to explore parameter space at a suspected minimum. pyBLoCXS was originally implemented and tested to handle Bayesian Low-Count X-ray Spectral (BLoCXS) analysis in Sherpa using simple composite spectral models, and additional research is underway to test more complex cases.

The underlying statistical model in pyBLoCXS employs Bayes' Rule 2 where the posterior probability distribution is proportional to the product of the conditional and prior distributions.

$$p(\theta|d,I) = \frac{p(d|\theta,I)p(\theta|I)}{p(d|I)} \tag{1}$$

Where $p(\theta|d,I)$ represents the posterior distribution; $p(d|\theta,I)$, the likelihood; $p(\theta|I)$, the prior; and $p(d|I)$ is considered constant.

$$p(\theta|d,I) \propto p(d|\theta,I)p(\theta|I) \tag{2}$$

Where θ represents the model parameters; d, the observed data; and I, the initial information.

The pyBLoCXS package includes a method `get_draws` to sample the posterior distribution for a specified number of iterations. The loop draws parameter values from a multi-variate Student's t distribution and calculates the likelihood on the parameter proposal given the observed data. The proposal is then accepted or rejected according to the current Metropolis-Hastings acceptance criterion and repeat. See Figure 4 for a graphical representation of the MCMC loop.

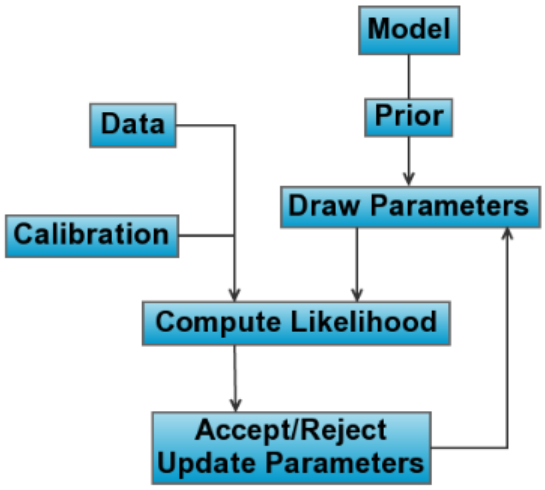

Fig. 4: *The Metropolis-Hastings criterion implemented in pyBLoCXS.*

pyBLoCXS currently has two sampling methods. The Python class, `MH`, implements a Metropolis-Hastings jumping rule characterized by the Student's t distribution based on the input scales, best-fit values, and user-specified degrees of freedom. The second class, `MetropolisMH`, is a variation on `MH` in that it implements a Metropolis-Hastings jumping rule with a Metropolis jumping rule centered on the current draw.

The pyBLoCXS package can be used separately from Sherpa using just Python and NumPy. The main inputs to pyBLoCXS are a callable function to calculate the log-likelihood, an ndarray of best-fit parameter values of size `n`, an ndarray of the multi-variate scales of size `n x n`, and the degrees of freedom. The ndarray of

multi-variate scales is typically the covariance matrix calculated at the best-fit parameter values.

pyBLoCXS is based on the techniques described in the paper [van2001], however, pyBLoCXS implements a different type of sampler. A description of the MCMC methods implemented in pyBLoCXS can be found in Chapter 11 of [gel2004].

Example

The Thurber problem is an example of Non-linear least squares regression from the Statistical Reference Datasets (StRD) at the National Institute of Standards and Technology (NIST). The observed data results from a NIST study of semiconductor electron mobility. The data includes 37 observations with the dependent variable (y) represented as electron mobility and the independent variable (x) as the log of the density.

$$y = f(x;\beta) + \varepsilon = \frac{\beta_1 + \beta_2 x + \beta_3 x^2 + \beta_4 x^3}{1 + \beta_5 x + \beta_6 x^2 + \beta_7 x^3} + \varepsilon \quad (3)$$

$$\vec{p} = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7\} \quad (4)$$

We define a compact high-level UI to access the Sherpa confidence method. The illustrative example below minimizes the Thurber function using least-squares and Sherpa's implementation of Levenberg-Marquardt (LMDIF). The results can be found in Table 1. The fit results agree to 99.99% for all parameters.

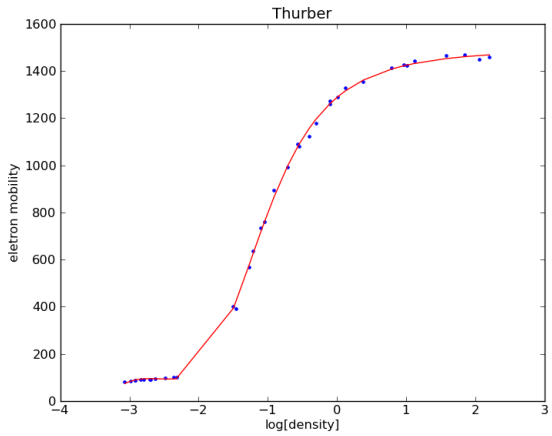

Fig. 5: *Thurber fit*

Loading Data

This example relies on a package asciitable to read columns of text data into NumPy arrays. The Thurber problem defines an equation as the model the function which is written as a vectorized Python function using NumPy ufuncs.

```
import sherpa.ui as ui
import asciitable

tbl = asciitable.read('Thurber.dat',
                      Reader=asciitable.NoHeader,
                      data_start=36,
                      delimiter="\s")

# Columns as NumPy arrays
x = tbl['col2']
y = tbl['col1']
```

Parameter	Certified Values	Sherpa Values	Percentage
β_1	1.2881396800E+03	1.28813971e+03	99.999
β_2	1.4910792535E+03	1.49106665e+03	99.999
β_3	5.8323836877E+02	5.83229092e+02	99.998
β_4	7.5416644291E+01	7.54148565e+01	99.998
β_5	9.6629502864E-01	9.66284739e-01	99.999
β_6	3.9797285797E-01	3.97967752e-01	99.999
β_7	4.9727297349E-02	4.97257372e-02	99.997

TABLE 1: *The best-fit parameters for Thurber problem.*

```
p0 = [1000, 1000, 400, 40, 0.7, 0.3, 0.03]

def calc(p, x):
    xx = x**2
    xxx = x**3
    return ( (p[0] + p[1]*x + p[2]*xx + p[3]*xxx) /
             (1. + p[4]*x + p[5]*xx + p[6]*xxx) )

# define a tolerance
tol = 1.e-9
```

Sherpa Fitting

Below, the Thurber data arrays are loaded into a Sherpa data set using `load_arrays`. The example indicates the fit statistic, optimization method, and defines the `calc` function as the Sherpa model using `load_user_model`. The function `add_user_pars` accepts Python lists that specify the parameter names, initial values, and optionally the parameter limits. A user can fit the model to the data using `fit` and access the best-fit parameter values as a NumPy array `popt`.

```
names = ['b%i' % (ii+1) for ii in range(len(p0))]

ui.load_arrays(1, x, y, ui.Data1D)
ui.set_stat('leastsq')

ui.set_method('levmar')
ui.set_method_opt('gtol', tol)
ui.set_method_opt('xtol', tol)
ui.set_method_opt('ftol', tol)
ui.set_method_opt('epsfcn', tol)

ui.load_user_model(calc, 'mdl')
ui.add_user_pars('mdl', names, p0)
ui.set_model('mdl')

ui.fit()
popt = ui.get_fit_results().parvals
```

Sherpa Confidence Method

The example below highlights the calculation of the asymmetric 1σ confidence limits on seven parameters using `conf` using the C-statistic and simplex. The confidence limits are accessible as NumPy arrays `pmins` and `pmaxes`.

```
ui.set_stat('cstat')
ui.set_method('neldermead')
ui.fit()
ui.conf()

# lower error bars
pmins  = ui.get_conf_results().parmins
```

Parameter	Best Fit	Lower Bound	Upper Bound
β_1	1288.12	-12.1594	12.1594
β_2	1452.67	-73.3571	17.8398
β_3	557.281	-7.09913	34.3927
β_4	70.2984	-10.1567	2.42915
β_5	0.943534	-0.0575953	0.0433009
β_6	0.387899	-0.02639	0.0199346
β_7	0.0403176	-0.0134162	0.00914532

TABLE 2: *The one standard deviation confidence limits for Thurber problem.*

Parameter	Best Fit	Lower Bound	Upper Bound
β_1	1288.12	-12.1594	12.1594
β_2	1452.67	-55.506	55.506
β_3	557.281	-39.7166	39.7166
β_4	70.2984	-7.58595	7.58595
β_5	0.943534	-0.0471354	0.0471354
β_6	0.387899	-0.0217024	0.0217024
β_7	0.0403176	-0.0107599	0.0107599

TABLE 3: *The one standard deviation covariance results for Thurber problem.*

```
# upper error bars
pmaxes = ui.get_conf_results().parmaxes
```

Confidence limits on the example Thurber problem are listed in Table 2.

Sherpa Covariance Method

To compute the covariance matrix, Sherpa first estimates the information matrix by finite differences by reducing a multi-dimensional problem to a series of 1-D problems. Sherpa then iteratively applies second central differencing with extrapolation (Kass 1987). The covariance matrix follows by inverting the information matrix.

The example below calculates the covariance matrix accessible as a NumPy array for the seven parameter values. An estimation of the symmetric confidence limits are found in the NumPy arrays `pmins` and `pmaxes`.

```
ui.covar()

# lower error bars
pmins  = ui.get_covar_results().parmins

# upper error bars
pmaxes = ui.get_covar_results().parmaxes

# where pmins == -pmaxes

# Access the covariance matrix
cov = ui.get_covar_results().extra_output
```

It is important to note that the parameter uncertainties computed by covariance do not consider correlations between parameters and can underestimate or overestimate the true uncertainty. Compare the differences in uncertainties computed by `conf` and `covar` in Tables 2 and 3.

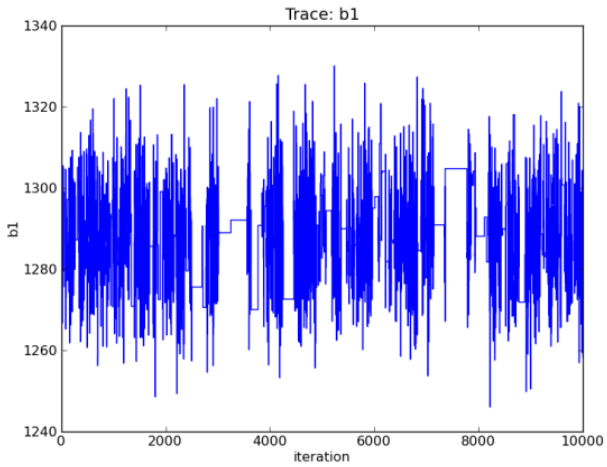

Fig. 6: *A trace plot show the draws for β_1 per iteration*

pyBLoCXS

The example below selects the Metropolis-Hastings using the pyBLoCXS [sem2011] function `set_sampler`. The likelihood and parameter draws are computed using the high level function `get_draws`. The inputs to `get_draws` at the API level are a function to calculate the likelihood, the best-fit parameter values, the covariance matrix centered on the best-fit, the degrees of freedom, and the number of iterations. At the high level, only the number of iterations is needed as input. The other inputs are accessed from Sherpa by pyBLoCXS.

```
import pyblocxs

pyblocxs.set_sampler('MH')
stats, accept, params = pyblocxs.get_draws(niter=1e4)

pyblocxs.plot_trace(params[0], 'b1')
```

pyBLoCXS includes high level plotting functions to display the trace, the cumulative distribution function, and the probability distribution function. The trace plot for β_1 includes gaps in the line that indicate rejected parameter proposals. This example has an acceptance rate of ~24%, well within the accepted range for an MCMC chain.

The `scatter` function in matplotlib can be used to visualize the log-likelihood according to two selected parameters. Using Metropolis-Hastings as the sampler, the density plot is shown in Figure 7 . For parameters β_3 and β_4, a distinct correlation is shown as a long and narrow well.

```
import pylab
pylab.scatter(params[0], params[1],
              c=stats, cmap=pylab.cm.jet)
```

To contrast the previous sampler, selecting Metropolis-Hastings mixed with Metropolis and re-sampling shows a density plot with a larger region of parameter space and distinct tail features in Figure 8.

```
pyblocxs.set_sampler('MetropolisMH')
stats, accept, params = pyblocxs.get_draws(niter=1e4)

pylab.scatter(params[0], params[1],
              c=stats, cmap=cm.jet)
```

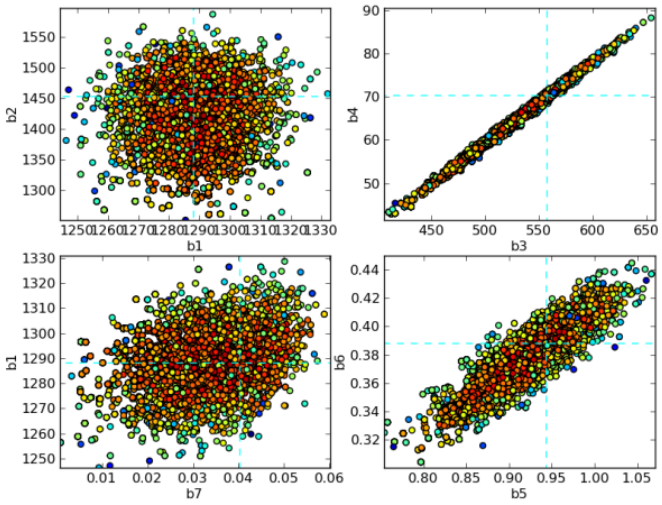

Fig. 7: *Log-likelihood density using Metropolis-Hastings in pyBLoCXS.*

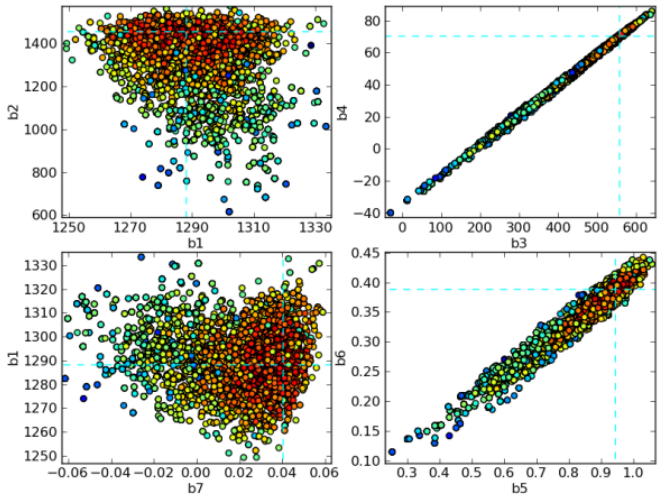

Fig. 8: *Log-likelihood density using Metropolis-Hastings with Metropolis in pyBLoCXS.*

Priors

pyBLoCXS includes a flexible definition of priors for each fit parameter. Priors are important for maximum likelihood analysis to take advantage of priori knowledge such as the range of parameter values. pyBLoCXS assumes each parameter to have a flat or non-informative prior by default.

Using the Sherpa model `normgauss1d`, a Gaussian prior can be added to the first parameter in the set with

```
import sherpa.astro.ui as ui
import pyblocxs

ui.xsapec.therm
ui.normgauss1d.g1
g1.pos=2.5; g1.fwhm=0.5

pyblocxs.set_prior(therm.kT,g1)
pyblocxs.set_sampler_opt('defaultprior',
                         False)
pyblocxs.set_sampler_opt('priorshape',
                         [True, False, False])
pyblocxs.set_sampler_opt('originalscale',
                         [True, True, True])
```

By accepting callable functions, pyBLoCXS can support arbitrary functions representing the parameter prior.

```
import sherpa.astro.ui as ui
import numpy

def lognorm(x, sigma=0.5, norm=1.0, x0=20.):
    xl=numpy.log10(x)+22.
    return (norm/numpy.sqrt(2*numpy.pi)/sigma)*
            numpy.exp(-0.5*(xl-x0)*(xl-x0)/sigma/sigma)

ui.xsphabs.abs1

pyblocxs.set_prior(abs1.NH,lognorm)
pyblocxs.set_sampler_opt('defaultprior',
                         False)
pyblocxs.set_sampler_opt('priorshape',
                         [True, False, False])
pyblocxs.set_sampler_opt('originalscale',
                         [True, True, True])
```

Accounting for Calibration Uncertainties

Future released versions of pyBLoCXS will include methods to incorporate the systematic uncertainties in modeling high energy spectra. These uncertainties which have largely been ignored due to the lack of a comprehensive method, can introduce bias in the calculation of model parameters and can underestimate their variance. Specifically, pyBLoCXS will utilize the calibration uncertainties in the effective area curve for spectral analysis. The effective area for high energy detectors records the sensitivity of the detector as a function of energy.

Calibration samples of the effective area are described in Drake et al. (2006) using Principle Component Analysis (PCA) to represent the curve's variability. Samples of the effective area can also be found using simulations.

pyBLoCXS perturbs the effective area curve by sampling from the calibration information at each iteration in the MCMC loop accurately accounting for the non-linear effects in the systematic uncertainty. With this method, best-fit model parameters values and their uncertainty are estimated more accurately and efficiently using Sherpa and pyBLoCXS.

Conclusion

We describe the Sherpa confidence method and the techniques included in pyBLoCXS to estimate parameter confidence when fit parameters present with correlations or the parameters are not themselves normally distributed. Multi-dimensional parameter space is typically non-uniform and Sherpa provides the user with options to explore its topology. The included code example describes an application of the Sherpa confidence method and the pyBLoCXS sampling method.

Support of the development of Sherpa is provided by National Aeronautics and Space Administration through the Chandra X-ray Center, which is operated by the Smithsonian Astrophysical Observatory for and on behalf of the National Aeronautics and Space Administration contract NAS8-03060.

REFERENCES

[avn1976] Y. Avni. *Energy spectra of X-ray clusters of galaxies*, The Astrophysical Journal, 210:642-646, Dec. 1976.

[fre2001] P. E. Freeman, S. Doe, A. Siemiginowska. *Sherpa: a Mission-Independent Data Analysis Application* SPIE Proceedings, Vol. 4477, p.76, 2001.

[gel2004] A. Gelman et al. *Bayesian Data Analysis* Chapman & Hall Texts in Statistical Science Series, 2nd Ed. 2004.

[lee2011] H. Lee et al. *Accounting for Calibration Uncertainties in X-ray Analysis: Effective Area in Spectral Fitting*, The Astrophysical Journal 731:126, 2011.

[nm] Computer Journal, J.A. Nelder and R. Mead, 1965, vol 7, pp. 308-313.

[pro2002] R. Protassov et al. *Statistics, Handle with Care: Detecting Multiple Model Components with the Likelihood Ratio Test*, The Astrophysical Journal, 571:545-559, May 2002.

[ref2009] B. Refsdal et al. *Sherpa: 1D/2D modeling in fitting in Python* Proceedings of the 8th Python in Science conference (SciPy 2009), G Varoquaux, S van der Walt, J Millman (Eds.), pp. 51-57.

[sem2011] Siemiginowska et al. *pyblocxs: Bayesian Low-Counts X-ray Spectral Analysis in Sherpa*, Astronomical Society of the Pacific Conference Series, 442:439. 2011.

[van2001] D. van Dyk et al. *Analysis of Energy Spectra with Low Photon Counts via Bayesian Posterior Simulation*, The Astrophysical Journal, 548:224, February 2001.

[lm] Lecture Notes in Mathematics 630: Numerical Analysis, G.A. Watson (Ed.), Springer-Verlag: Berlin, 1978, pp. 105-116

Crab: A Recommendation Engine Framework for Python

Marcel Caraciolo‡*, Bruno Melo‡, Ricardo Caspirro‡

Abstract—Crab is a flexible, fast recommender engine for Python that integrates classic information filtering recommendation algorithms in the world of scientific Python packages (NumPy,SciPy, Matplotlib). The engine aims to provide a rich set of components from which you can construct a customized recommender system from a set of algorithms. It is designed for scability, flexibility and performance making use of scientific optimized python packages in order to provide simple and efficient solutions that are acessible to everybody and reusable in various contexts: science and engineering. The engine takes users' preferences for items and returns estimated preferences for other items. For instance, a web site that sells movies could easily use Crab to figure out, from past purchase data, which movies a customer might be interested in watching to. This work presents our inniative in developing this framework in Python following the standards of the well-known machine learning toolkit Scikit-Learn to be an alternative solution for Mahout Taste collaborative framework for Java. Finally, we discuss its main features, real scenarios where this framework is already applied and future extensions.

Index Terms—data mining, machine learning, recommendation systems, information filtering, framework, web

Introduction

With the great advancements of machine learning in the past few years, many new learning algorithms have been proposed and one of the most recognizable techniques in use today are the recommender engines [Adoma2005]. There are several services or sites that attempt to recommend books or movies or articles based on users past actions [Linden2003] , [Abhinandan2007] . By trying to infer tastes and preferences, those systems focus to identify unknown items that are are of interest given an user. Although people's tastes vary, they do follow patterns. People tend to like things that are similar to other items they like. For instance, because a person loves bacon-lettuce-and-tomato sandwiches, the recommender system could guess that he would enjoy a club sandwich, which is mostly the same sandwich, with turkey. Likewise, people tend to like things that similar people like. When a friend entered design school, he saw that just about every other design student owned a Macintosh computer - which was no surprise, as she already a lifetime Mac User. Recommendation is all about predicting these patterns of taste, and using them to discover new and desirable things a person didn't know about.

* *Corresponding author: marcel@muricoca.com*
‡ *Muricoca Labs*

Recommendation engines have been implemented in programming languages such as C/C++, Java, among others and made publicly available. One of the most popular implementations is the open-source recommendation library Taste, which was included in the Mahout framework project in 2008 [Taste] . Mahout is a well-known machine learning toolkit written in Java for building scalable machine libraries [Mahout] . It is specially a great resource for developers who are willing to take a step into recommendation and machine learning technologies. Taste has enabled systematic comparisons between standard developed recommendation methods, leading to an increased visibility, and supporting their broad adoption in the community of machine learning, information filtering and industry. There are also several another publicly available implementations of recommender engines toolkits in the web [EasyRec] , [MyMediaLite]. Each one comes with its own interface, sometimes even not updated anymore by the project owners, a small set of recommendation techniques implemented, and unique benefits and drawbacks.

For Python developers, which has a considerable amount of machine learning developers and researchers, there is no single unified way of interfacing and testing these recommendation algorithms, even though there are some approaches but found incomplete or missing the required set for creating and evaluating new methods [PySuggest], [djangorecommender]. This restraints developers and researchers from fully taking advantage of the recent developments in recommendation engines algorithms as also an obstacle for machine learning researchers that will not want to learn complex programming languages for writing their recommender approaches. Python has been considered for many years a excellent choice for programming beginners since it is easy to learn with simple syntax, portable and extensive. In scientific computing field, high-quality extensive libraries such as Scipy, MatPlotlib and Numpy have given Python an attractive alternative for researchers in academy and industry to write machine learning implementations and toolkits such as Brain, Shogun, Scikit-Learn, Milk and many others.

The reason of not having an alternative for python machine learning developers by providing an unified and easy-to-use recommendation framework motivated us to develop a recommendation engine toolbox that provides a rich set of features from which the developer can build a customized recommender system. The result is a framework, called Crab, with focus on large-scale recommendations making use of scientific python packages such as Scipy, Numpy and Matplotlib to provide simple and efficient solutions for constructing recommender systems that are acessible and reusable in various contexts. Crab provides a generic

interface for recommender systems implementations, among them the collaborative filtering approaches such as User-Based and Item-Based filtering, which are already available for use. The recommender interfaces can be easily combined with more than 10 different pairwise metrics already implemented, like the cosine, tanimoto, pearson, euclidean using Scipy and Numpy basic optimized functions [Breese1998]. Moreover, it offers support for using similarities functions such as user-to-user or item-to-item and allows easy integration with different input domains like databases, text files or python dictionaries.

Currently, the collaborative filtering algorithms are widely supported. In addition to the User-Based and Item-Based filtering techniques, Crab implements several pairwise metrics and provides the basic interfaces for developers to build their own customized recommender algorithms. Finally, several widely used performance measures, such as accuracy, precision, recall are implemented in Crab.

An important aspect in the design of Crab was to enable very large-scale recommendations. Crab is currently being rewritten to support optimized scientific computations by using Scipy and Numpy routines. Another feature concerned by the current maintainers is to make Crab support sparse and large datasets in a way that there is a little as possible overhead for storing the data and intermediate results. Moreover, Crab also aims to support scaling in recommender systems in order to build high-scale, dynamic and fast recommendations over simple calls. It is also planned to support distributed recommendation computation by interfacing with the distributed computation library MrJob written in Python currently developed by Yelp [MrJob]. What sets Crab apart from many other recommender systems toolboxes, is that it provides interactive interfaces to build, deploy and evaluate customized recommender algorithms written in Python running on several platforms such as Linux, BSD, MacOS and Windows.

The outline of this paper is as follows. We first discuss the Crab's main features by explaining the architecture of the framework. Next, we provide our current approach for representing the data in our system and current challenges. Then, we also presents how Crab can be used in production by showing a real scenario where it is already deployed. Finally, we discuss about our plans to handle with distributed recommendation computations. Also, our conclusions and future works are also presented at the end of this paper.

Recommender Engines

Crab contains a recommender engine, in fact, several types beginning with conventional in the literature user-based and item-based recommenders. It provides an assortment of components that may be plugged together and customized to create an ideal recommender for a particular domain. The toolkit is implemented using Python and the scientific environments for numerical applications such as Scipy and NumPy. The decision of choosing those libraries is because they are widely used in scientific computations specially in python programs. Another reason is because the framework uses the Scikit-learn toolkit as dependant, which provides basic components from our recommender interfaces derive [Scikitlearn] . The Figure 1 presents the relationship between these basic components. Not all Crab-based recommenders will look like this -- some will employ different components with different relationships, but this gives a sense of the role of each component.

The Data Model implementation stores and provides access to all the preferences, user and item data needed in the recommendation. The Similarity interface provides the notion of how similar two users or items are; where this could be based on one of many possible pairwise metrics or calculations. Finally, a Recommender interface which inherits the BaseEstimator from Scikit-learn pull all these components together to recommend items to users, and related functionality.

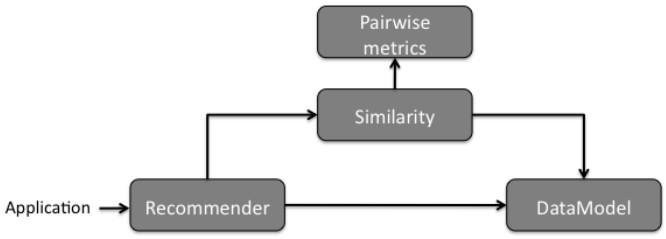

Fig. 1: *Simplified illustration of the component interaction in Crab*

It is easy to explore recommendations with Crab. Let's go through a trivial example. First, we need input to the recommender, data on which to base recommendations. Generally, this data takes the form of preferences which are associations from users to items, where these users and items could be anything. A preference consist of a user ID and an item ID, and usually a number expressing the strength of the user's preference for the item. IDs in Crab can be represented by any type indexable such as string, integers, etc. The preference value could be anything, as long as larger values mean strong positive preferences. For instance, these values can be considered as ratings on a scale of 1 to 5, where the user has assigned "1" to items he can't stand, and "5" to his favorites.

Crab is able to work with text files containing information about users and their preferences. The current state of the framework allows developers to connect with databases via Django's ORM or text files containing the user IDs, product IDs and preferences. For instance, we will consider a simple dataset including data about users, cleverly named "1" to "5" and their preferences for four movies, which we call "101" through "104". By loading this dataset and passing as parameter to the dataset loader, all the inputs will be loaded in memory by creating a Data Model object.

Analyzing the data set shown at Figure 2, it is possible to notice that Users 1 and 5 seem to have similar tastes. Users 1 and 3 don't overlap much since the only movie they both express a preference for is 101. On other hand, users 1 and 2 tastes are opposite- 1 likes 101 while 2 doesn't, and 1 likes 103 while 2 is just the opposite. By using one of recommender algorithms available in Crab such as the User-Based-Filtering with the given data set loaded in a Data Model as input, just run this script using your favorite IDE as you can see the snippet code below.

```
from models.basic_models import FileDataModel
from recommenders.basic_recommenders
   import UserBasedRecommender
from similarities.basic_similarities
   import UserSimilarity
from neighborhoods.basic_neighborhoods
   import NearestUserNeighborhood
from metrics.pairwise import pearson_correlation

user_id = 1
# load the dataset
model = FileDataModel('simple_dataset.csv')
similarity = UserSimilarity(model,
                            pearson_correlation)
neighbor = NearestUserNeighborhood(similarity,
                                   model, 4, 0.0)
```

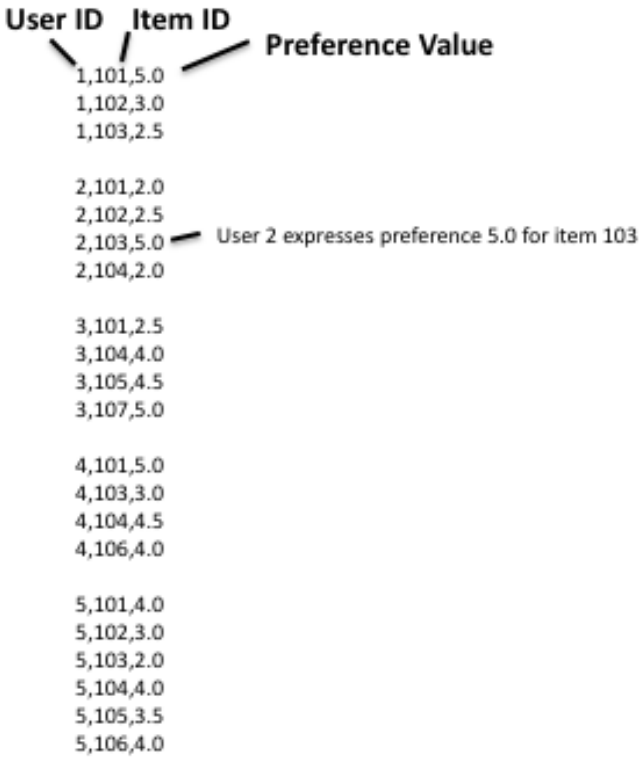

Fig. 2: *Book ratings data set - intro.csv*

```
# create the recommender engine
recommender = UserBasedRecommender(model, similarity,
                                   neighbor, False)
# recommend 1 item to user 1
print recommender.recommend(user_id, 1)
```

The output of running program should be: 104. We asked for one top recommendation, and got one. The recommender engine recommended the book 104 to user 1. This happens because it estimated user 1's preference for book 104 to be about 4.3 and that was the highest among all the items eligible for recommendations. It is important to notice that all recommenders are estimators, so they estimate how much users may like certain items. The recommender worked well considering a small data set. Analyzing the data you can see that the recommender picked the movie 104 over all items, since 104 is a bit more highly rated overall. This can be refforced since user 1 has similar preferences to the users 4 and 5, where both have highly rated.

For small data sets, producing recommendations appears trivial as showed above. However, for data sets that are huge and noisy, it is a different situation. For instance, consider a popular news site recommending new articles to readers. Preferences are inferred from article clicks. But, many of these "preferences" may be noisy - maybe a reader clicked an article but did not like it, or, had clicked the wrong story. Imagine also the size of the data set - perhaps billions of clicks in a month. It is necessary for recommender engines to handle with real-life data sets, and Crab as Mahout is focusing on how to deal with large and sparse data as we will discuss in a future section.

Therefore, before deploying recommender engines in Crab into production, it is necessary to present another main concept in our framework at the next section: representation of data.

Representing Data

Recommender systems are data-intensive and runtime performance is greatly affected by quantity of data and its representation. In Crab the recommender-related data is encapsulated in the implementations of DataModel. DataModel provides efficient access to data required by several recommender algorithms. For instance, a DataModel can provide a count or an array of all user IDs in the input data, or provide access to all preferences associated to an item.

One of the implementations available in Crab is the in-memory implementation DictDataModels. This model is appropriate if the developer wants to construct his data representation in memory by passing a dictionary of user IDs and their preferences for item IDs. One of benefits of this model is that it can easily work with JSON files, which is commonly used as output at web services and REST APIs, since Python converts the json input into a bult-in dictionary.

```
from models.basic_models
   import DictPreferenceDataModel

dataset = {'1':{'101': 3.0, '102': 3.5},
           '2':{'102': 4.0, '103':2.5, '104': 3.5}}

#load the dataset
model = DictPreferenceDataModel(dataset)
print model.user_ids()
#numpy.array(['1','2'])

print model.preference_value('1', '102')
#3.5

print model.preferences_for_item('102')
#numpy.array([('1',3.5),('2',4.0)])
```

Typically the model that developers will use is the FileDataModel - which reads data from a file and stores the resulting preference data in memory, in a DictDataModel. Comma-separated-value or tab-separated files which each line contains one datum: user ID, item ID and preference value are acceptable as input to the model. Zipped and gzipped files will be supported, since they are commonly used for store huge data in a compressed format.

For data sets which ignore the preference values, that is, ignore the strength of preference, Crab also has an appropriate DataModel twin of DictDataModel called BooleanDictDataModel. This is likewise as in-memory DictDataModel implementation, but one which internally does not store the preference values. These preferences also called "boolean preferences" have two states: exists, or does not exist and happens when preferences values aren't available to begin with. For instance, imagine a news site recommending articles to user based on previously viewed article. It is not typical for users to rate articles. So the recommender recommends articles based on previously viewed articles, which establishes some association between user and item, an interesting scenario for using the BooleanDictModel.

```
from models.basic_models
   import DictBooleanDataModel

dataset = {'1':['101','102'],
           '2':['102','103','104']}

#load the dataset
model = DictBooleanDataModel(dataset)

print model.user_ids()
#numpy.array(['1','2'])

print model.preference_value('1', '102')
#1.0 - all preferences are valued with 1.0
```

```
print model.preferences_for_item('102')
#numpy.array([('1',1.0),('2',1.0)])
```

Crab also supports store and access preference data from a relational database. The developer can easily implement their recommender by using customized DataModels integrated with several databases. One example is the MongoDB, a NoSQL database commonly used for non-structured data [MongoDB]. By using MongoEngine, a ORM adapter for integrating MongoDB with Django, we could easily set up a customized Data Model to access and retrieve data from MongoDB databases easily [Django], [MongoEngine]. In fact, it is already in production at a recommender engine using Crab for a brazilian social network called AtéPassar. We will explore more about it in the next sections.

One of the current challenges that we are facing is how to handle with all this data in-memory. Specially for recommender algorithms, which are data intensive. We are currently investigating how to store data in memory and work with databases directly without using in-memory data representations. We are concerned that it is necessary for Crab to handle with huge data sets and keep all this data in memory can affects the performance of the recommender engines implemented using our framework. Crab uses Numpy arrays for storing the matrices and in the organization of this paper at the time we were discussing about using scipy.sparse packages, a Scipy 2-D sparse matrix package implemented for handling with sparse a matrices in a efficient way.

Now that we have discussed about how Crab represents the data input to recommender, the next section will examine the recommenders implemented in detail as also how to evaluate recommenders using Crab tools.

Making Recommendations

Crab already supports the collaborative recommender user-based and item-based approaches. They are considered in some of the earliest research in the field. The user-based recommender algorithm can be described as a process of recommending items to some user, denoted by u, as follows:

```
for every item i that u has no preference for yet

  for every other user v that has preference for i

     compute a similarity s between u and v

     incorporate v's preference for i, weighted by s,
        into a running average

return the top items, ranked by weighted average
```

The outer loop suggests we should consider every known item that the user hasn't already expressed a preference for as a candidate for recommendation. The inner loop suggests that we should look to any other user who has expressed a preference for this candidate item and see what his or her preference value for it was. In the end, those values are averaged to come up with an estimate -- a weighted average. Each preference value is weighted in the average by how similar that user is to the target user. The more similar a user, the more heavily that we weight his or her preference value. In the standard user-based recommendation algorithm, in the step of searching for every known item in the data set, instead, a "neighborhood" of most similar users is computed first, and only items known to those users are considered.

In the first section we have already presented a user-based recommender in action. Let's go back to it in order to explore the components the approach uses.

```
# do the basic imports
user_id = 1

# load the dataset
model = FileDataModel('simple_dataset.csv')

# define the similarity used and the pairwise metric
similarity = UserSimilarity(model,
                            pearson_correlation)

# for neighborhood we will use the k-NN approach
neighbor = NearestUserNeighborhood(similarity,
                                   model, 4, 0.0)

# now add all to the UserBasedRecommender
recommender = UserBasedRecommender(model, similarity,
                                   neighbor, False)

#recommend 2 items to user 1
print recommender.recommend(user_id,2)
```

UserSimilarity encapsulates the concept of similarity amongst users. The UserNeighborhood encapsulates the notion of a group of most-similar users. The UserNeighborhood uses a UserSimilarity, which extends the basic interface BaseSimilarity. However, the developers are encouraged to plug in new variations of similarity - just creating new BaseSimilarity implementations - and get quite different results. As you will see, Crab is not one recommender engine at all, but a set of components that may be plugged together in order to create customized recommender systems for a particular domain. Here we sum up the components used in the user-based approach:

- Data model implemented via DataModel
- User-to-User similarity metric implemented via UserSimilarity
- User neighborhood definition implementd via UserNeighborhood
- Recommender engine implemented via Recommender, in this case, UserBasedRecommender

The same approach can be used at UserNeighborhood where developers also can create their customized neighborhood approaches for defining the set of most similar users. Another important part of recommenders to examine is the pairwise metrics implementation. In the case of the User-based recommender, it relies most of all in this component. Crab implements several pairwise metrics using the Numpy and Scipy scientific libraries such as Pearson Correlation, Euclidean distance, Cosine measure and distance implementations that ignore preferences entirely like as Tanimoto coefficient and Log-likehood.

Another approach to recommendation implemented in Crab is the item-based recommender. Item-based recommendation is derived from how similar items are to items, instead of users to users. The algorithm implemented is familiar to the user-based recommender:

```
for every item i that u has no preference for yet

   for every item j that u has a preference for

       compute a similarity s between i and j

       add u's preference for j, weighted by s,
        to a running average
```

```
return the top items, ranked by weighted average
```

In this algorithm it is evaluated the item-item similarity, not user-user similarity as shown at the user-based approach. Although they look similar, there are different properties. For instance, the running time of an item-based recommender scales up as the number of items increases, whereas a user-based recommender's running time goes up as the number of users increases. The performance advantage in item-based approach is significant compared to the user-based one. Let's see how to use item-based recommender in Crab with the following code.

```
# do the basic imports
user_id = 1

# load the dataset
model = FileDataModel('simple_dataset.csv')

# define the Similarity used and the pairwise metric
similarity = ItemSimilarity(model, euclidean_distance)

# there is no neighborhood in this approach
# now add all to the ItemBasedRecommender
recommender = ItemBasedRecommender(model,
                                   similarity, False)

# recommend 2 items to user 1
print recommender.recommend(user_id,2)
```

Here it employs ItemBasedRecommender rather than UserBasedRecommender, and it requires a simpler set of dependencies. It also implements the ItemSimilarity interface, which is similar to the UserSimilarity that we've already seen. The ItemSimilarity also works with the pairwise metrics used in the UserSimilarity. There is no item neighborhood, since it compares series of preferences expressed by many users for one item instead of by one user for many items.

Now that we have seen some techniques implemented at Crab, which produces recommendations for a user, it is now time to answer another question, "what are the best recommendations for a user ?". A recommender engine is a tool and predicts user preferences for items that he haven't expressed any preference for. The best possible recommender is a tool that could somehow know, before you do, exactly estimate how much you would like every possible item available. The remainder of this section will explore evaluation of a recommender, an important step in the construction of a recommender system, which focus on the evaluating the quality of the its estimated preference values - that is, evaluating how closely the estimated preferences match the actual preferences.

Crab supports several metrics widely used in the recommendation literature such as the RMSE (root-mean-square-error), precision, recall and F1-Score. Let's see the previous example code and instead evaluate the simple recommender we created, on our data set:

```
from evaluators.statistics
    import RMSRecommenderEvaluator

# initialize the recommender
# initialize the RMSE Evaluator
evaluator = RMRecommenderEvaluator()

# using training set with 70% of data and 30% for test
print evaluator.evaluate(recommender,
                         model, 0.7, 1.0)
#0.75
```

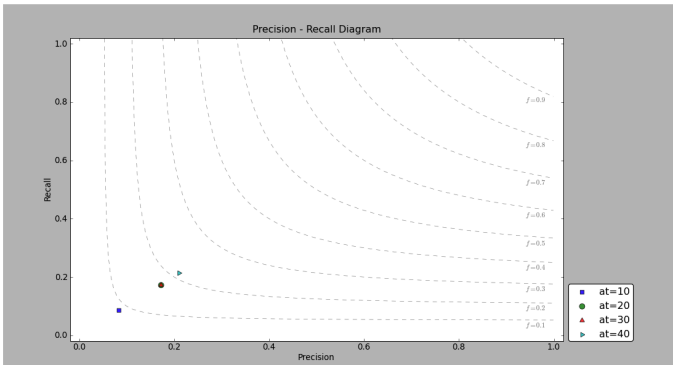

Fig. 3: *PrecisionxRecall Graph with F1-Score.*

Most of the action happens in evaluate(). The RecommenderEvaluator handles splitting the data into a training and test set, builds a new training DataModel and Recommender to test, and compares its estimated preferences to the actual test data. See that we pass the Recommender to this method. This is because the evaluator will need to build a Recommender around a newly created training DataModel. This simple code prints the result of the evaluation: a score indicating how well the Recommender performed. The evaluator is an abstract class, so the developers may build their custom evaluators, just extending the base evaluator.

For precision, recall and F1-Score Crab provides also a simple way to compute these values for a Recommender:

```
from evaluators.statistics
    import IRStatsRecommenderEvaluator

# initialize the recommender
# initialize the IR Evaluator
evaluator = IRStatsRecommenderEvaluator()

# call evaluate considering the top 4 items recommended.
print evaluator.evaluate(recommender, model, 2, 1.0)
# {'precision': 0.75, 'recall': 1.0,
    'f1Score': 0.6777}
```

The result you see would vary significantly due to random selection of training data and test data. Remember that precision is the proportion of top recommendations that are good recommendations, recall is the proportion of good recommendations that appear in top recommendations and F1-Score is a score that analyzes the proportion against precision and recall. So Precision at 2 with 0.75 means on average about a three quarters of recommendations were good. Recall at 2 with 1.0; all good recommendations are among those recommendations. In the following graph at Figure 3, it presents the PrecisionxRecall with F1-Scores evaluated. A brief analysis shows that more training set size grows, more the accuracy score grows. It is important to notice that the evaluator doe not measure if the algorithm is better or faster. It is necessary to make a comparison between the algorithms to check the accuracy specially on other data sets available.

Crab supports several tools for testing and evaluating recommenders in a painless way. One of the future releases will support the plot of charts to help the developers to better analyze and visualized their recommender behavior.

Taking Recommenders to Production

So far we have presented the recommender algorithms and variants that Crab provides. We also presented how Crab handles with

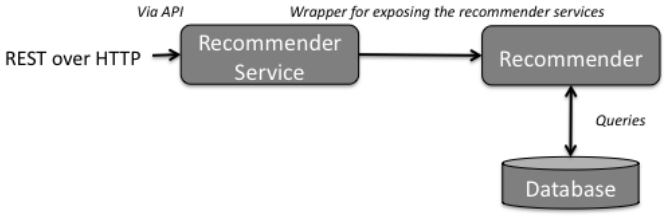

Fig. 4: *Crab Web Services server-side interaction over HTTP*

accuracy evaluation of a recommender. But another important step for a recommender life-cycle is to turn it into a deployable production-ready web service.

We are extending Crab in order to allow developers to deploy a recommender as a stand-alone component of your application architecture, rather than embed it inside your application. One common approach is to expose the recommendations over the web via simple HTTP or web services protocols such as SOAP or REST. One advantage using this service is that the recommender is deployed as a web- accessible service as independent component in a web container or a standalone process. In the other hand, this adds complexity, but it allows other applications written in other languages or running at remote machines to access the service. We are considering use framework web Django with the the Django-Piston RESTful builder to expose the recommendations via a simple API using REST over HTTP [DjangoPiston]. Our current structure is illustrated in Figure 4, which wraps the recommender implementation using the django models and piston handlers to provide the external access.

There is a recommender engine powered by Crab in production using REST APIs to access the the recommendations. The recommender engine uses collaborative filtering algorithms to recommend users, study groups and videos in a brazilian educational social network called AtéPassar [AtePassar] . Besides the suggestions, the recommender was also extended to provide the explanations for each recommendation, in a way that the user not only receives the recommendation but also why the given recommendation was proposed to him. The recommender is in production since January 2011 and suggested almost 60.000 items for more than 50.000 users registered at the network. The following Figure 5 shows the web interface with the recommender engine in action at AtéPassar. One contribution of this work was a new Data Model for integrating with MongoDB database for retrieving and storing the recommendations and it is being rewritten for the new release of Crab supporting Numpy and Scipy libraries.

Crab can comfortably digest medium and small data sets on one machine and produce recommendations in real time. But it still lacks a mechanism that handles a much larger data set. One common approach is distribute the recommendation computations, which will be detailed in the next section.

Distributing Recommendation Computations

For large data sets with millions of preferences, the current approaches for single machines would have trouble processing recommendations in the way we have seen in the last sections. It is necessary to deploy a new type of recommender algorithms using a distributed and parallelized computing approach. One of the most popular paradigms is the MapReduce and Hadoop [Hadoop].

Fig. 5: *AtéPassar recommendation engine powered by Crab Framework*

Crab didn't support at the time of writing this paper distributed computing, but we are planning to develop variations on the item-based recommender approach in order to run it in the distributed world. One of our plans is to use the Yelp framework mrJob which supports Hadoop and it is written in Python, so we may easily integrate it with our framework. One of the main concerns in this topic is to give Crab a scalable and efficient recommender implementation without having high memory and resources consumption as the number of items grows.

Another concern is to investigate and develop other distributed implementations such as Slope One, Matrix Factorization, giving the developer alternatives for choosing the best solution for its need specially when handling with large data sets using the power of Hadoop's MapReduce computations. Another important optimization is to use the JIT compiler PyPy for Python which is being development and will bring faster computations on NumPy [NumpyFollowUp].

Conclusion and Future Works

In this paper we have presented our efforts in building a recommender engine toolkit in Python, which we believe that may be useful and make an increasing impact beyond the recommendation systems community by benefiting diverse applications. We are confident that Crab will be an interesting alternative for machine learning researchers and developers to create, test and deploy their recommendation algorithms writing a few lines of code with the simplicity and flexibility that Python with the scientific libraries Numpy and Scipy offers. The project uses as dependency the Scikit-learn toolkit, which forces the Crab framework to cope with high standards of coding and testing, turning it into a madure

and efficient machine learning toolkit. Discussing the technical aspects, we are also always improving the framework by planning to develop new recommender algorithms such as Matrix Factorization, SVD and Boltzmann machines. Another concern is to bring to the framework not only collaborative filtering algorithms but also content based filtering (content analysis), social relevance proximity graphs (social/trust networks) and hybrid approaches. Finally it is also a requirement to a recommender engine to be scalable, that is, to handle with large and sparse data sets. We are planning to develop a scalable recommendation implementation by using Yelp framework mrJob which supports Hadoop and MapReduce as explained in the previous section.

Our project is hosted at Github repository and it is open for machine learning community to use, test and help this project to grow up. Future releases are planned which will include more projects building on it and a evaluation tool with several plots and graphs to help the machine learning developer better understand the behavior of his recommender algorithm. It is an alternative for Python developers to the Mahout machine learning project written in Java. The source code is freely available under the BSD license at http://github.com/muricoca/crab.

REFERENCES

[Adoma2005] Adomavicius, G.; Tuzhilin, A. *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions*, IEEE Transactions on Knowledge and Data Engineering; 17(6):734–749, June 2005.

[Linden2003] Greg Linden, Brent Smith, and Jeremy York. *Amazon.com Recommendations: Item-to-Item Collaborative Filtering.*, IEEE Internet Computing 7, 1, 76-80, January 2003.

[Abhinandan2007] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram, *Google news personalization: scalable online collaborative filtering.*, In Proceedings of the 16th international conference on World Wide Web (WWW '07). ACM, New York, NY, USA, 271-280, 2007.

[Taste] *Taste - Collaborative Filtering For Java* , accessible at: http://taste.sourceforge.net/.

[Mahout] *Mahout - Apache Machine Learning Toolkit* ,accessible at: http://mahout.apache.org/

[EasyRec] *EasyRec* ,accessible at: http://www.easyrec.org/

[MyMediaLite] *MyMediaLite Recommender System Library*, accessible at: http://www.ismll.uni-hildesheim.de/mymedialite/

[PySuggest] *PySuggest*, accessible at: http://code.google.com/p/pysuggest/

[djangorecommender] *Django-recommender* accessible at: http://code.google.com/p/django-recommender/

[Breese1998] J. S. Breese, D. Heckerman, and C. Kadie. *Empirical analysis of predictive algorithms for collaborative filtering.*, UAI, Madison, WI, USA, pp. 43-52, 1998.

[MrJob] *mrjob*, accessible at: https://github.com/Yelp/mrjob

[Scikitlearn] *Scikit-learn*, accessible at: http://scikit-learn.sourceforge.net/

[MongoDB] *MongoDB*, accessible at: https://www.mongodb.org/

[Django] *Django*, accessible at: https://www.djangoproject.com/

[MongoEngine] *MongoEngine*, accessible at: https://www.mongoengine.org/

[DjangoPiston] *Django-Piston*, accessible at: https://bitbucket.org/jespern/django-piston/wiki/Home

[AtePassar] *AtéPassar*, accessible at: http://atepassar.com

[Hadoop] *Hadoop*, accessible at: http://hadoop.apache.org/

[NumpyFollowUp] *Numpy Follow up*, accessible at: http://morepypy.blogspot.com/2011/05/numpy-follow-up.html

gpustats: GPU Library for Statistical Computing in Python

Andrew Cron‡*, Wes McKinney‡

Abstract—In this work we discuss **gpustats**, a new Python library for assisting in "big data" statistical computing applications, particularly Monte Carlo-based inference algorithms. The library provides a general code generation / metaprogramming framework for easily implementing discrete and continuous probability density functions and random variable samplers. These functions can be utilized to achieve more than 100x speedup over their CPU equivalents. We demonstrate their use in an Bayesian MCMC application and discuss avenues for future work.

Index Terms—GPU, CUDA, OpenCL, Python, statistical inference, statistics, metaprogramming, sampling, Markov Chain Monte Carlo (MCMC), PyMC, big data

Introduction

Due to the high theoretical computational power and low cost of graphical processing units (GPUs), researchers and scientists in a wide variety of fields have become interested in applying them within their problem domains. A major catalyst for making GPUs widely accessible was the development of the general purpose GPU computing frameworks, [CUDA] and [OpenCL], which enable the user to implement general numerical algorithms in a simple extension of the C language to run on the GPU. In this paper, we will restrict our technical discussion to the CUDA architecture for NVIDIA cards, while later commenting on CUDA versus OpenCL.

As CUDA and OpenCL provide a C API for GPU programming, significant portions of the development process can be quite low level and require large amounts of boilerplate code. To address this problem, [PyCUDA] and [PyOpenCL] provide a high-level Python interface to the APIs, while also streamlining the process of writing and testing new GPU functions, or *kernels*. PyCUDA and PyOpenCL compile GPU kernels on the fly and upload them to the card; this eliminates the need to recompile a C executable with each code iteration. The result is a much more rapid and user-friendly GPU development experience, as the libraries take care of much of the boilerplate code for interacting with the GPU. They also provide seamless integration with [NumPy], which allows GPU functionality to integrate easily within a larger NumPy-based computing application. And, since code is compiled on the fly, it is relatively straightforward to implement metaprogramming approaches to dynamically generate customized GPU kernels within a Python program.

In this paper, we discuss some of the challenges of GPU computing and how GPUs can be applied to statistical inference applications. We further show how PyCUDA and PyOpenCL are ideal for implementing certain kinds of statistical computing functions on the GPU.

* *Corresponding author: ajc40@stat.duke.edu*
‡ *Duke University*

Development Challenges in GPU Computing

While a CPU may have 4 or 8 cores, a latest generation GPU may have 256, 512, or even more computational cores. However, the GPU memory architecture is highly specialized to so-called single instruction multiple data (SIMD) problems. This generally limits the usefulness of GPUs to highly parallelizable data processing applications. The developer writes a function, known as a *kernel*, to process a unit of data. The kernel function is then executed once for each unit or chunk of data.

The GPU has a large single *global* memory store (typically 512MB to 4GB) with which data sets can be transferred to and from the CPU memory space. However, each group, or *block*, of threads are assigned a small piece (typically 16K to 64K) of ultra low-latency *shared* cache memory which is orders of magnitude faster than the global memory. Therefore, the main challenge for the developer, outside of writing the kernel function, is structuring the computation to optimally utilize each thread block's shared memory and minimizing reads from and writes to the global memory. Careful coordination of the threads is required to transfer memory efficiently from global to shared. We omit the low-level details of this process and instead refer the interested reader to the CUDA API guide ([NvidiaGuide]). See Figure 1 for a rough diagram of the computing architecture.

As a larger computation is divided up into a *grid* of thread blocks, a typical CUDA kernel takes the following structure:

- Coordinate threads within a block to transfer relevant data for block from global to shared memory
- Perform computation using (fast) shared memory
- Coordinate threads to transfer results back into global memory

Computational Challenges in Likelihood-based Statistical Inference

In most standard and Bayesian statistical models, a probability distribution (or family of distributions) is assumed for each realization of the data. For example, the errors (residuals) in a linear

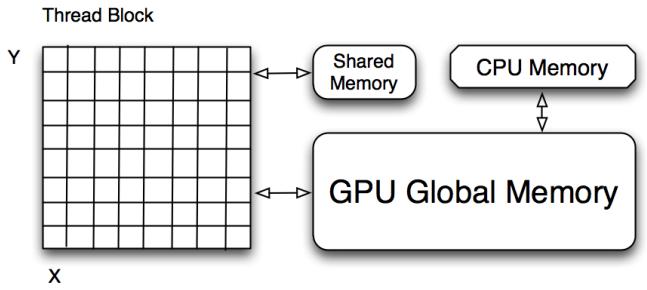

Fig. 1: *Mock-up of GPU computing architecture*

regression problem are assumed to be normally distributed with mean 0 and some variance σ^2. In a standard statistical inference problem given a set of distributional assumptions, the task is to estimate the parameters of those distributions. Under this setup, we can write down the *joint likelihood* for the data in mathematical terms

$$p(x_1,\ldots,x_n|\Theta) = \Pi_{i=1}^{n} p(x_i|\Theta), \tag{1}$$

where Θ represents the unknown parameters of the model, and $p(x_i|\Theta)$ is the probability density for observation x_i. This representation assumes that the data are independent and identically distributed. For example, we may wish to estimate the mean μ and variance σ^2 of a normally distributed population, in which case $\Theta = (\mu, \sigma^2)$ and

$$p(x_i|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x_i-\mu)^2/2\sigma^2} \tag{2}$$

In many statistical inference algorithms, the "goodness of fit" of the parameters Θ is evaluated based on the full data likelihood 1. It is common to use the logged likelihood function as 1 decomposes into a sum of log densities and this also reduces numerical precision problems.

Many numerical algorithms for fitting these likelihood-based models, especially Monte Carlo-based, involve evaluating the log-likelihood function over thousands of iterations. Thus as the size of the observed data grows, computational expense grows *as least* linearly in the number of data points. As above, if the data are assumed to be independently generated, the quantity $\log p(x_i|\Theta)$ for each observation x_i can be evaluated in parallel then summed to compute the full log-likelihood. This becomes a very natural setting for GPUs, and it is quite easy for GPUs to perform even better than large CPU clusters because of the large number of computing cores and very low memory latency. Suchard et al. studied these advantages in the Bayesian mixture model setting and found very promising results (100x speedup) on graphics cards that are now 2 years old ([JCGS]).

Another source of significant computation in statistical applications that we address is that of generating draws from random variables. In many algorithms (e.g. Bayesian Markov Chain Monte Carlo methods), large data sets may require generating thousands or millions of random variables from various probability distributions at each iteration of the algorithm.

Challenges of GPU Computing in Statistical Inference

As mentioned above, a CUDA or OpenCL programmer must be very mindful of the memory architecture of the GPU. There are multiple memory management issues to address, i.e. in CUDA parlance

- *Coalescing* transactions between global and shared memory; this is, coordinating groups of typically 16 to 32 threads to copy a contiguous chunk of 64 or 128 bytes in a single transaction
- Avoiding shared memory *bank conflicts*, i.e. threads competing for read/write access to a shared memory address

To make this more concrete, let's consider the task of implementing the log of the normal probability density function (pdf) 2. Given a data set with N observations, we wish to evaluate the pdf on each point for a set of parameters, i.e. the mean μ and variance σ^2. Thus, all that needs to be passed to the GPU is the data set and the parameters. A C function which can compute the log pdf for a single data point is

```
float log_normal_pdf(float* x, float* params) {
    float std = params[1];
    float xstd = (*x - params[0]) / std;
    return - (xstd * xstd) / 2 - 0.5 * LOG_2_PI
           - log(std);
}
```

In practice, one would hope that implementing a new probability density such as this would be as simple as writing this 4-line function. Unfortunately, to achieve optimal performance, the majority of one's attention must be focused on properly addressing the above data coordination / cache optimization problems. Thus, the full form of a GPU kernel implementing a pdf is typically as follows:

- Coordinate threads to copy (coalesce, if possible) data needed for thread block to shared memory
- Similarly copy parameters needed by thread back to shared memory
- Each thread computes a density result for a single data point, writing results to shared memory
- Copy/coalesce resulting density values back to global memory

Fortunately, the function signature for the vast majority of probability density functions of interest is the same, requiring only *data* and *parameters*. While the actual pdf function is very simple, the rest of the code is much more involved. Since the kernels are structurally the same, we would be interested in a way to reuse the code for steps 1, 2, and 4, which will likely be nearly identical for most of the functions. Were we programming in C, doing so would be quite difficult. But, since we have PyCUDA/PyOpenCL at our disposal, metaprogramming techniques can be utilized to do just that, as we discuss later.

With respect to probability densities, we make a brief distinction between *univariate* (observations are a single floating point value) and *multivariate* (vector-valued observations) distributions. In the latter case, the dimension of each observation (the length of each vector) typically must be passed as well. Otherwise, multivariate densities (e.g. multivariate normal) are handled similarly.

In a more general framework, we might wish to evaluate the pdf for multiple parameters at once, e.g. $(\mu_1,\sigma_1^2),\ldots,..(\mu_K,\sigma_K^2)$. In other words, $N*K$ densities need to be computed. A naive but wasteful approach would be to make K round trips to the GPU for each of the K sets of parameters. A better approach is to divide the data / parameter combinations among the GPU grid to maximize data reuse via the shared memory and perform all $N*K$ density computations in a single GPU kernel invocation. This introduces the additional question of how to divide the problem among thread blocks viz. optimally utilizing shared memory. As the available

GPU resources are device specific, we would wish to dynamically determine the optimal division of labor among thread blocks based on the GPU being used.

Avoiding *bank conflicts* as mentioned above is a somewhat thorny issue as it depends on the thread block layout and memory access pattern. It turns out in the **gpustats** framework that bank conflicts can be avoided with multivariate data by ensuring that the data dimension is not a multiple of 16. Thus, some data sets must be *padded* with arbitrary data to avoid this problem, while passing the true data dimension to the GPU kernel. If this is not done, bank conflicts will lead to noticeably degraded performance. We are hopeful that such workarounds can be avoided with future versions of GPU memory architecture.

For sampling random variables on the GPU, the process is reasonably similar. Just as with computing the density function, sampling requires the same parameters for each distribution to be passed. Many distributions can be derived by transforming draws from a uniform random variable on the interval [0, 1]. Thus, for such distributions it makes most sense to precompute uniform draws (either using the CPU or the GPU) and pass these precomputed draws to the GPU kernel. However, there are widely-used distributions, such as the gamma distribution, which are commonly sampled via *adaptive rejection sampling*. With this algorithm, the number of uniform draws needed to produce a single sample is not known *a priori*. Thus, such distributions would be very difficult to sample on the GPU.

Metaprogramming: probability density kernels and beyond

The **gpustats** Python library leverages the compilation-on-the-fly capabilities of PyCUDA and metaprogramming techniques to simplify the process of writing new GPU kernels for computing probability density functions, samplers, and other related statistical computing functionality. As described above in the normal distribution case, one would hope that writing a new density function would amount to writing the simple `log_normal_pdf` function and having the untidy global-shared cache management problem taken care of by the library. Additionally, we would like to have a mechanism for computing transformed versions of existing kernels. For example, `log_normal_pdf` could be transformed to the unlogged density by applying the exponent function.

To solve these problems, we have developed a prototype object-oriented code generation framework to make it easy to develop new kernels with minimal effort by the statistical user. We do so by taking advantage of the string templating functionality of Python and the CUDA API's support for inline functions on the GPU. These inline functions are known as device functions, marked by `__device__`. Since the data transfer / coalescing problem needs to be only solved once for each variety of kernel, we can use templating to generate a custom kernel for each new device function implementing a new probability density. It is then simple to enable element wise transformations of existing device functions, e.g. taking the `exp` of a logged probability density. In the **gpustats** framework, the code for implementing the logged and unlogged normal pdf is as follows:

```
_log_pdf_normal = """
 __device__ float %(name)s(float* x, float* params) {
    // mean stored in params[0]
    float std = params[1];

    // standardize
    float xstd = (*x - params[0]) / std;
    return - (xstd * xstd) / 2 - 0.5f * LOG_2_PI
           - log(std);
 }
"""
log_pdf_normal = DensityKernel('log_pdf_normal',
                               _log_pdf_normal)
pdf_normal = Exp('pdf_normal', log_pdf_normal)
```

The **gpustats** code generator will, at import time, generate a CUDA source file to be compiled on the fly by PyCUDA. Note that the `%(name)s` template is there to enable the device function to be given an appropriate (and non-conflicting) name in the generated source code, given that multiple versions of a single device function may exist. For example, the `Exp` transform generates a one-line device function taking the `exp` of the logged density function.

Python interface and device-specific optimization

Further work is needed to interface with the generated PyCUDA `SourceModule` instance. For example, the data and parameters need to be prepared in `ndarray` objects in the form that the kernel expects them. Since all of the univariate density functions, for example, have the same function signature, it's relatively straightforward to create a generic function taking care of this often tedious process. Thus, implementing a new density function requires only passing the appropriate function reference to the generic *invoker* function. Here we show what the function implementing the normal (logged and unlogged) pdf on multiple sets of parameters looks like:

```
def normpdf_multi(x, means, std, logged=True):
    if logged:
        cu_func = mod.get_function('log_pdf_normal')
    else:
        cu_func = mod.get_function('pdf_normal')
    packed_params = np.c_[means, std]
    return _univariate_pdf_call(cu_func, x,
                                packed_params)
```

Inside the above `_univariate_pdf_call` function, the attributes of the GPU device in use are examined to dynamically determine the thread block size and grid layout that will maximize the shared memory utilization. This is definitely an area where much time could be invested to determine a more "optimal" scheme.

Reusing data stored on the GPU

Since the above algorithms may be run repeatedly on the same data set, leaving a data set stored on the GPU global device memory is a further important optimization. Indeed, the time required to copy a large block of data to the GPU may be quite significant compared with the time required to execute the kernel.

Fortunately, PyCUDA and PyOpenCL have a `GPUArray` class which mimics its CPU-based NumPy counterpart `ndarray`, with the data being stored on the GPU. Thus, in functions like the above, the user can pass in a `GPUArray` to the function which will circumvent any copying of data to the GPU. Similarly, functions like `normpdf_multi` above can be augmented with an option to return a `GPUArray` instead of an `ndarray`. This is useful as in some algorithms the results of a density calculation may be immediately used for sampling random variables on the GPU. Avoiding round trips to the GPU device memory can result in a significant boost in performance, especially with smaller data sets.

Some basic benchmarks

We show some benchmarks for the univariate and multivariate normal probability density functions, both with and without using `GPUArray` to use data already stored on the GPU. These were carried out with a very modest NVIDIA GTS 250 desktop card, which has 128 CUDA cores (latest generation cards have up to 512). The CPU benchmarks were done on a standard Intel Core i7 930 processor. As you will see, the speedups with larger data sets can be quite dramatic. The reported numbers below are the *speedup*, i.e. the ratio of CPU average runtime divided by GPU average runtime.

Univariate Normal PDF: "Single" indicates that the density values were only computed for a single mean and variance. "Multi" indicates that they were computed for 8 (an arbitrary number) sets of means and variances in one shot. The column header indicates the number of data points.

	1e3	1e4	1e5	1e6
Single	0.2234	1.268	7.951	23.05
Single (GPUArray)	0.2407	1.291	9.359	38.72
Multi	1.46	7.035	26.19	43.73
Multi (GPUArray)	1.79	8.354	30.79	49.26

Multivariate Normal PDF: For this distribution, we used a streamlined C implementation of the density function (nearly identical code to the CUDA kernel) for benchmarking purposes so that it's an apples-to-apples comparison. For the data dimension we chose 15, again arbitrarily. Here we can really see an even greater impact of reusing data on the GPU:

	1e3	1e4	1e5	1e6
Single	0.6998	4.167	12.55	14.09
Single (GPUArray)	0.8465	6.03	32.59	64.12
Multi	3.126	18.41	60.18	63.89
Multi (GPUArray)	3.135	19.8	74.39	82

Application: PyMC integration

Low-hanging fruit for GPU integration in big data applications would be in [PyMC]. This is a library for implementing Bayesian Markov Chain Monte Carlo (MCMC) algorithms. The user describes the generative process for a data set and places prior distributions on the parameters of the generative process. PyMC then uses the well-known Metropolis-Hastings algorithm to approximate samples from the posterior distribution of the parameters given the observed data. A key step in Metropolis-Hastings is the proposal step in which new parameter values are selected via some *proposal distribution*, which is typically based on a symmetric random walk but may be more sophisticated. A new proposed value θ^* for θ is accepted or rejected based on the *acceptance ratio*

$$a^* = \frac{p(\theta^*)p(x|\theta^*)p(\theta^*|\theta)}{p(\theta)p(x|\theta)p(\theta|\theta^*)},$$

where $p(\theta)$ is the prior density for θ, $p(x|\theta)$ is the likelihood, and $p(\theta|\theta^*)$ is the proposal density. Understanding the details of how and why this algorithm works is not important for the scope of this paper. What is important is the fact that the quantity $p(x|\theta)$ is recomputed typically thousands of times to compute samples from the model. If the data x is very large, then the majority of the runtime of the MCMC may be spent recomputing the data likelihood for different parameters.

Enabling all of the PyMC distributions to run in *GPU mode* (so that likelihoods are computed on the GPU) would be very simple as soon as the probability density functions are implemented inside **gpustats**. Based on the above benchmarks, it is clear that integrating **gpustats** with PyMC could significantly reduce the overall runtime of many MCMC models on large data sets.

Conclusions and future work

As **gpustats** currently uses PyCUDA it can only be used with NVIDIA graphics cards. OpenCL, however, provides a parallel computing framework which can be executed on NVIDIA and ATI cards as well as on CPUs. Thus, it will make sense to enable the **gpustats** code generator to emit OpenCL code in the near future. As PyOpenCL is developed in lockstep with PyCUDA, altering the Python interface code to use PyOpenCL should not be too onerous. Using OpenCL currently has drawbacks for statistical applications: most significantly the lack of a pseudorandom number generator equivalent in speed and quality to [CURAND]. For simulation-based applications this can make a big impact. We are hopeful that this issue will be resolved in the next year or two.

Another important addition which would be important to some users is to enable multiple GPUs to be run in parallel to extract even better performance. While this would introduce more latency for small datasets and likely be unnecessary, for processing large data sets, the overhead of calling out to 3 GPUs, for example, would likely be much less than the computation time. Ideally code could be seamlessly run on multiple GPUs. Furthermore, the device memory on the GPU can be small. However, most GPUs allow asynchronous memory copying and thread execution, so a streaming approach can be taken on large datasets that can be partitioned. In some cases, the streaming overhead can be virtually eliminated by the asynchronous calls.

Note that **gpustats** is still in prototype stages, so its API will be highly subject to change. We hope to generate interest in this development direction as it could have an impact in boosting Python's status as a desirable statistical computing environment for big data. An end goal would be to reimplement most of the probability distributions (densities, samplers, etc.) in **scipy.stats** on the GPU and to fully integrate these where possible throughout PyMC and other related libraries. The meta-programming approach offers a development friendly environment that could also be considered a prototype for a useful GPU programming model in general.

Another interesting avenue, but perhaps of less importance for Python programmers, would be the generation of wrapper interfaces to the generated CUDA or OpenCL source module for other programming languages, such as R. However, without the easy-to-use PyCUDA and PyOpenCL bindings this would likely be a fairly significant undertaking.

Acknowledgements

We are grateful to Quanli Wang, Jacob Frelinger, Adam Richards, and Cliburn Chan of Duke University for their comments and useful discussions. Research reported here was partially supported by the National Institutes of Health under grant RC1-AI086032. Any opinions, findings and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the NIH.

REFERENCES

[CUDA] NVIDIA Corporation. CUDA GPU computing framework http://www.nvidia.com/object/cuda_home_new.html

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

[JCGS] M. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron and M. West. *Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures.* Journal of Computational and Graphical Statistics 19 (2010): 419-438 http://pubs.amstat.org/doi/abs/10.1198/jcgs.2010.10016

[NvidiaGuide] NVIDIA Corporation. *Nvidia CUDA: Programming Guide.* (2010), http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[CURAND] NVIDIA Corporation. CURAND Random Number Generator http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf

[PyMC] C. Fonnesbeck, A. Patil, D. Huard, *PyMC: Markov Chain Monte Carlo for Python*, http://code.google.com/p/pymc/

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

[SciPy] E. Jones, T. Oliphant, P. Peterson, http://scipy.org

[PyCUDA] A. Klöckner, http://mathema.tician.de/software/pycuda

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

Using the Global Arrays Toolkit to Reimplement NumPy for Distributed Computation

Jeff Daily[‡*], Robert R. Lewis[§]

Abstract—Global Arrays (GA) is a software system from Pacific Northwest National Laboratory that enables an efficient, portable, and parallel shared-memory programming interface to manipulate distributed dense arrays. Using a combination of GA and *NumPy*, we have reimplemented *NumPy* as a distributed drop-in replacement called Global Arrays in NumPy (*GAiN*). Scalability studies will be presented showing the utility of developing serial *NumPy* codes which can later run on more capable clusters or supercomputers.

Index Terms—Global Arrays, Python, NumPy, MPI

Introduction

Scientific computing with Python typically involves using the *NumPy* package. *NumPy* provides an efficient multi-dimensional array and array processing routines. Unfortunately, like many Python programs, *NumPy* is serial in nature. This limits both the size of the arrays as well as the speed with which the arrays can be processed to the available resources on a single compute node.

For the most part, *NumPy* programs are written, debugged, and run in singly-threaded environments. This may be sufficient for certain problem domains. However, *NumPy* may also be used to develop prototype software. Such software is usually ported to a different, compiled language and/or explicitly parallelized to take advantage of additional hardware.

Global Arrays in NumPy (*GAiN*) is an extension to Python that provides parallel, distributed processing of arrays. It implements a subset of the *NumPy* API so that for some programs, by simply importing *GAiN* in place of *NumPy* they may be able to take advantage of parallel processing transparently. Other programs may require slight modification. This allows those programs to take advantage of the additional cores available on single compute nodes and to increase problem sizes by distributing across clustered environments.

Background

Like any complex piece of software, *GAiN* builds on many other foundational ideas and implementations. This background is not intended to be a complete reference, rather only what is necessary to understand the design and implementation of *GAiN*. Further details may be found by examining the references or as otherwise noted.

* Corresponding author: jeff.daily@pnnl.gov
‡ Pacific Northwest National Laboratory
§ Washington State University

NumPy

NumPy [Oli06] is a Python extension module which adds a powerful multidimensional array class `ndarray` to the Python language. *NumPy* also provides scientific computing capabilities such as basic linear algebra and Fourier transform support. *NumPy* is the de facto standard for scientific computing in Python and the successor of the other numerical Python packages Numarray [Dub96] and numeric [Asc99].

The primary class defined by *NumPy* is `ndarray`. The `ndarray` is implemented as a contiguous memory segment. Internally, all `ndarray` instances have a pointer to the location of the first element as well as the attributes `shape`, `ndim`, and `strides`. `ndim` describes the number of dimensions in the array, `shape` describes the number of elements in each dimension, and `strides` describes the number of bytes between consecutive elements per dimension. The `ndarray` can be either FORTRAN- or C-ordered. Recall that in FORTRAN, the first dimension has a stride of one while it is the opposite (last) dimension in C. `shape` can be modified while `ndim` and `strides` are read-only and used internally, although their exposure to the programmer may help in developing certain algorithms.

The creation of `ndarray` instances is complicated by the various ways in which it can be done such as explicit constructor calls, view casting, or creating new instances from template instances (e.g. slicing). To this end, the `ndarray` does not implement Python's `__init__()` object constructor. Instead, `ndarrays` use the `__new__() classmethod`. Recall that `__new__()` is Python's hook for subclassing its built-in objects. If `__new__()` returns an instance of the class on which it is defined, then the class's `__init__()` method is also called. Otherwise, the `__init__()` method is not called. Given the various ways that `ndarray` instances can be created, the `__new__() classmethod` might not always get called to properly initialize the instance. `__array_finalize__()` is called instead of `__init__()` for `ndarray` subclasses to avoid this limitation.

The element-wise operators in *NumPy* are known as *Universal Functions*, or *ufuncs*. Many of the methods of `ndarray` simply invoke the corresponding ufunc. For example, the operator + calls `ndarray.__add__()` which invokes the ufunc `add`. Ufuncs are either unary or binary, taking either one or two arrays as input, respectively. Ufuncs always return the result of the operation as an `ndarray` or `ndarray` subclass. Optionally, an additional output parameter may be specified to receive the results of the operation. Specifying this output parameter to the ufunc avoids the sometimes unnecessary creation of a new `ndarray`.

Ufuncs can operate on `ndarray` subclasses or array-like objects. In order for subclasses of the `ndarray` or array-like objects to utilize the ufuncs, they may define three methods or one attribute which are `__array_prepare__()`, `__array_wrap__()`, `__array__()`, and `__array_priority__`, respectively. The `__array_prepare__()` and `__array_wrap__()` methods will be called on either the output, if specified, or the input with the highest `__array_priority__`. `__array_prepare__()` is called on the way into the ufunc after the output array is created but before any computation has been performed and `__array_wrap__()` is called on the way out of the ufunc. Those two functions exist so that `ndarray` subclasses can properly modify any attributes or properties specific to their subclass. Lastly, if an output is specified which defines an `__array__()` method, results will be written to the object returned by calling `__array__()`.

Single Program, Multiple Data

Parallel applications can be classified into a few well defined programming paradigms. Each paradigm is a class of algorithms that have the same control structure. The literature differs in how these paradigms are classified and the boundaries between paradigms can sometimes be fuzzy or intentionally blended into hybrid models [Buy99]. The Single Program Multiple Data (SPMD) paradigm is one example. With SPMD, each process executes essentially the same code but on a different part of the data. The communication pattern is highly structured and predictable. Occasionally, a global synchronization may be needed. The efficiency of these types of programs depends on the decomposition of the data and the degree to which the data is independent of its neighbors. These programs are also highly susceptible to process failure. If any single process fails, generally it causes deadlock since global synchronizations thereafter would fail.

Message Passing Interface (MPI)

Message passing libraries allow efficient parallel programs to be written for distributed memory systems. MPI [Gro99a], also known as MPI-1, is a library specification for message-passing that was standardized in May 1994 by the MPI Forum. It is designed for high performance on both massively parallel machines and on workstation clusters. An optimized MPI implementation exists on nearly all modern parallel systems and there are a number of freely available, portable implementations for all other systems [Buy99]. As such, MPI is the de facto standard for writing massively parallel application codes in either FORTRAN, C, or C++.

The MPI-2 standard [Gro99b] was first completed in 1997 and added a number of important additions to MPI including, but not limited to, one-sided communication and the C++ language binding. Before MPI-2, all communication required explicit handshaking between the sender and receiver via `MPI_Send()` and `MPI_Recv()` in addition to non-blocking variants. MPI-2's one-sided communication model allows reads, writes, and accumulates of remote memory without the explicit cooperation of the process owning the memory. If synchronization is required at a later time, it can be requested via `MPI_Barrier()`. Otherwise, there is no strict guarantee that a one-sided operation will complete before the data segment it accessed is used by another process.

mpi4py

mpi4py is a Python wrapper around MPI. It is written to mimic the C++ language bindings. It supports point-to-point communication, one-sided communication, as well as the collective communication models. Typical communication of arbitrary objects in the FORTRAN or C bindings of MPI require the programmer to define new MPI datatypes. These datatypes describe the number and order of the bytes to be communicated. On the other hand, strings could be sent without defining a new datatype so long as the length of the string was understood by the recipient. mpi4py is able to communicate any serializable Python object since serialized objects are just byte streams. mpi4py also has special enhancements to efficiently communicate any object implementing Python's buffer protocol, such as *NumPy* arrays. It also supports dynamic process management and parallel I/O [Dal05], [Dal08].

Global Arrays and Aggregate Remote Memory Copy Interface

The GA toolkit [Nie06], [Nie10], [Pnl11] is a software system from Pacific Northwest National Laboratory that enables an efficient, portable, and parallel shared-memory programming interface to manipulate physically distributed dense multidimensional arrays, without the need for explicit cooperation by other processes. GA compliments the message-passing programming model and is compatible with MPI so that the programmer can use both in the same program. GA has supported Python bindings since version 5.0. Arrays are created by calling one of the creation routines such as `ga.ceate()`, returning an integer handle which is passed to subsequent operations. The GA library handles the distribution of arrays across processes and recognizes that accessing local memory is faster than accessing remote memory. However, the library allows access mechanisms for any part of the entire distributed array regardless of where its data is located. Local memory is acquired via `ga.access()` returning a pointer to the data on the local process, while remote memory is retrieved via `ga.get()` filling an already allocated array buffer. Individual discontiguous sets of array elements can be updated or retrieved using `ga.scatter()` or `ga.gather()`, respectively. GA has been leveraged in several large computational chemistry codes and has been shown to scale well [Apr09].

The Aggregate Remote Memory Copy Interface (ARMCI) provides general-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication). ARMCI operations are optimized for contiguous and non-contiguous (strided, scatter/gather, I/O vector) data transfers. It also exploits native network communication interfaces and system resources such as shared memory [Nie00]. ARMCI provides simpler progress rules and a less synchronous model of RMA than MPI-2. ARMCI has been used to implement the Global Arrays library, GPSHMEM - a portable version of Cray SHMEM library, and the portable Co-Array FORTRAN compiler from Rice University [Dot04].

Cython

Cython [Beh11] is both a language which closely resembles Python as well as a compiler which generates C code based on Python's C API. The Cython language additionally supports calling C functions as well as static typing. This makes writing C extensions or wrapping external C libraries for the Python language as easy as Python itself.

Previous Work

GAiN is similar in many ways to other parallel computation software packages. It attempts to leverage the best ideas for

transparent, parallel processing found in current systems. The following packages provided insight into how *GAiN* was to be developed.

MITMatlab [Hus98], which was later rebranded as Star-P [Ede07], provides a client-server model for interactive, large-scale scientific computation. It provides a transparently parallel front end through the popular MATLAB [Pal07] numerical package and sends the parallel computations to its Parallel Problem Server. Star-P briefly had a Python interface. Separating the interactive, serial nature of MATLAB from the parallel computation server allows the user to leverage both of their strengths. This also allows much larger arrays to be operated over than is allowed by a single compute node.

Global Arrays Meets MATLAB (GAMMA) [Pan06] provides a MATLAB binding to the GA toolkit, thus allowing for larger problem sizes and parallel computation. GAMMA can be viewed as a GA implementation of MITMatlab and was shown to scale well even within an interpreted environment like MATLAB.

IPython [Per07] provides an enhanced interactive Python shell as well as an architecture for interactive parallel computing. IPython supports practically all models of parallelism but, more importantly, in an interactive way. For instance, a single interactive Python shell could be controlling a parallel program running on a supercomputer. This is done by having a Python engine running on a remote machine which is able to receive Python commands.

distarray [Gra09] is an experimental package for the IPython project. distarray uses IPython's architecture as well as MPI extensively in order to look and feel like *NumPy* `ndarray` instances. Only the SPMD model of parallel computation is supported, unlike other parallel models supported directly by IPython. Further, the status of distarray is that of a proof of concept and not production ready.

A Graphics Processing Unit (GPU) is a powerful parallel processor that is capable of more floating point calculations per second than a traditional CPU. However, GPUs are more difficult to program and require other special considerations such as copying data from main memory to the GPU's on-board memory in order for it to be processed, then copying the results back. The GpuPy [Eit07] Python extension package was developed to lessen these burdens by providing a *NumPy*-like interface for the GPU. Preliminary results demonstrate considerable speedups for certain single-precision floating point operations.

A subset of the Global Arrays toolkit was wrapped in Python for the 3.x series of GA by Robert Harrison [Har99]. It illustrated some important concepts such as the benefits of integration with *NumPy* -- the local or remote portions of the global arrays were retrieved as *NumPy* arrays at which point they could be used as inputs to *NumPy* functions like the ufuncs.

Co-Array Python [Ras04] is modeled after the Co-Array FORTRAN extensions to FORTRAN 95. It allows the programmer to access data elements on non-local processors via an extra array dimension, called the co-dimension. The `CoArray` module provided a local data structure existing on all processors executing in a SPMD fashion. The CoArray was designed as an extension to Numeric Python [Asc99].

Design

The need for parallel programming and running these programs on parallel architectures is obvious, however, efficiently programming for a parallel environment can be a daunting task. One area of research is to automatically parallelize otherwise serial programs and to do so with the least amount of user intervention [Buy99]. *GAiN* attempts to do this for certain Python programs utilizing the *NumPy* module. It will be shown that some *NumPy* programs can be parallelized in a nearly transparent way with *GAiN*.

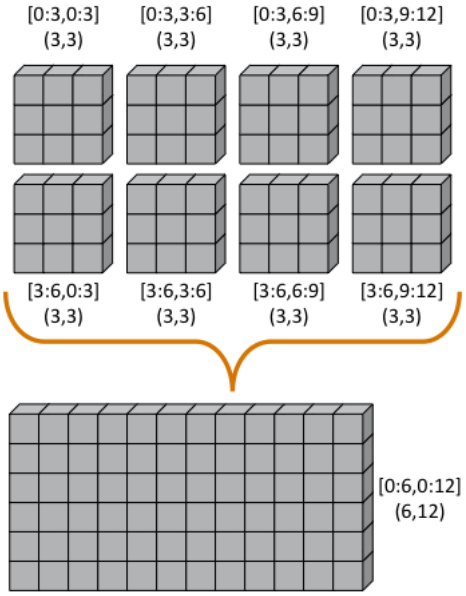

Fig. 1: *Each local piece of the* `gain.ndarray` *has its own shape (in parenthesis) and knows its portion of the distribution (in square brackets). Each local piece also knows the global shape.*

There are a few assumptions which govern the design of *GAiN*. First, all documented *GAiN* functions are collective. Since Python and *NumPy* were designed to run serially on workstations, it naturally follows that *GAiN*, running in an SPMD fashion, will execute every documented function collectively. Second, only certain arrays should be distributed. In general, it is inefficient to distribute arrays which are relatively small and/or easy to compute. It follows, then, that *GAiN* operations should allow mixed inputs of both distributed and local array-like objects. Further, *NumPy* represents an extensive, useful, and hardened API. Every effort to reuse *NumPy* should be made. Lastly, GA has its own strengths to offer such as processor groups and custom data distributions. In order to maximize scalability of this implementation, we should enable the use of processor groups [Nie05].

A distributed array representation must acknowledge the duality of a global array and the physically distributed memory of the array. Array attributes such as `shape` should return the global, coalesced representation of the array which hides the fact the array is distributed. But when operations such as `add()` are requested, the corresponding pieces of the input arrays must be operated over. Figure 1 will help illustrate. Each local piece of the array has its own shape (in parenthesis) and knows its portion of the distribution (in square brackets). Each local piece also knows the global shape.

A fundamental design decision was whether to subclass `ndarray` or to provide a work-alike replacement for the entire `numpy` module. The *NumPy* documentation states that `ndarray` implements `__new__()` in order to control array creation via constructor calls, view casting, and slicing. Subclasses implement `__new__()` for when the constructor is called directly, and `__array_finalize__()` in order to set additional attributes

or further modify the object from which a view has been taken. One can imagine an `ndarray` subclass called `gainarray` circumventing the usual `ndarray` base class memory allocation and instead allocating a smaller `ndarray` per process while retaining the global `shape`. One problem occurs with view casting -- with this approach the other `ndarray` subclasses know nothing of the distributed nature of the memory within the `gainarray`. *NumPy* itself is not designed to handle distributed arrays. By design, ufuncs create an output array when one is not specified. The first hook which *NumPy* provides is `__array_prepare__()` which is called *after the output array has been created*. This means any ufunc operation on one or more `gainarray` instances without a specified output would automatically allocate the entire output on each process. For this reason alone, we opted to reimplement the entire `numpy` module, controlling all aspects of array creation and manipulation to take into account distributed arrays.

We present a new Python module, `gain`, developed as part of the main Global Arrays software distribution. The release of GA v5.0 contained Python bindings based on the complete GA C API, available in the extension module `ga`. The GA bindings as well as the `gain` module were developed using Cython. With the upcoming release of GA v5.1, the module `ga.gain` is available as a drop-in replacement for *NumPy*. The goal of the implementation is to allow users to write

```
import ga.gain as numpy
```

and then execute their code using MPI e.g.

```
mpiexec -np 4 python script.py
```

In order to succeed as a drop-in replacement, all attributes, functions, modules, and classes which exist in `numpy` must also exist within `gain`. Efforts were made to reuse as much of `numpy` as possible, such as its type system. As of GA v5.1, arrays of arbitrary fixed-size element types and sizes can be created and individual fields of C `struct` data types accessed directly. *GAiN* is able to use the `numpy` types when creating the GA instances which back the `gain.ndarray` instances.

GAiN follows the owner-computes rule [Zim88]. The rule assigns each computation to the processor that owns the data being computed. Figures 2 and 3 illustrate the concept. For any array computation, *GAiN* bases the computation on the output array. The processes owning portions of the output array will acquire the corresponding pieces of the input array(s) and then perform the computation locally, *calling the original NumPy routine* on the corresponding array portions. In some cases, for example if the output array is a view created by a slicing operation, certain processors will have no computation to perform.

The *GAiN* implementation of the `ndarray` implements a few important concepts including the dual nature of a global array and its individual distributed pieces, slice arithmetic, and separating collective operations from one-sided operations. When a `gain.ndarray` is created, it creates a Global Array of the same shape and type and stores the GA integer handle. The distribution on a given process can be queried using `ga.distribution()`. The other important attribute of the `gain.ndarray` is the *global_slice*. The global_slice begins as a list of `slice` objects based on the original `shape` of the array.

```
self.global_slice = [slice(0,x,1) for x in shape]
```

Slicing a `gain.ndarray` must return a view just like slicing a `numpy.ndarray` returns a view. The approach taken is to

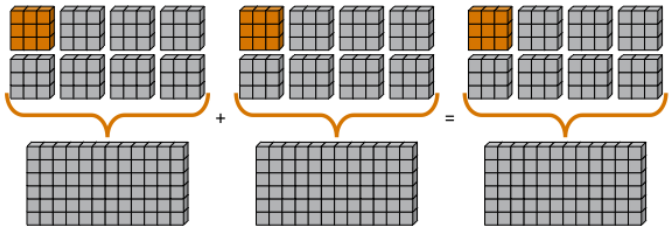

Fig. 2: *Add two arrays with the same data distribution. There are eight processors for this computation. Following the owner-computes rule, each process owning a piece of the output array (far right) retrieves the corresponding pieces from the sliced input arrays (left and middle). For example, the corresponding gold elements will be computed locally on the owning process. Note that for this computation, the data distribution is the same for both input arrays as well as the output array such that communication can be avoided by using local data access.*

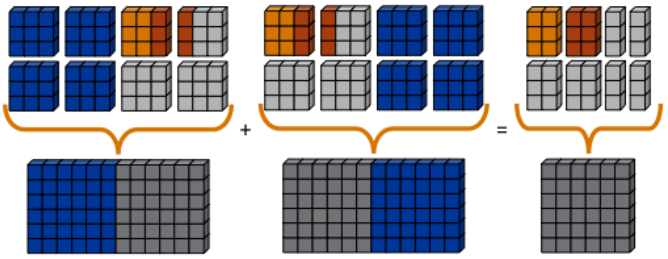

Fig. 3: *Add two sliced arrays. There are eight processors for this computation. The elements in blue were removed by a slice operation. Following the owner-computes rule, each process owning a piece of the output array (far right) retrieves the corresponding pieces from the sliced input arrays (left and middle). For example, the corresponding gold elements will be computed locally on the owning process. Similarly for the copper elements. Note that for this computation, the data for each array is not equivalently distributed which will result in communication.*

apply the `key` of the `__getitem__(key)` request to the `global_slice` and store the new `global_slice` on the newly created view. We call this type of operation *slice arithmetic*. First, the `key` is *canonicalized* meaning `Ellipsis` are replaced with `slice(0,dim_max,1)` for each dimension represented by the `Ellipsis`, all `slice` instances are replaced with the results of calling `slice.indices()`, and all negative index values are replaced with their positive equivalents. This step ensures that the length of the `key` is compatible with and based on the current shape of the array. This enables consistent slice arithmetic on the canonicalized keys. Slice arithmetic effectively produces a new `key` which, when applied to the same original array, produces the same results had the same sequence of keys been applied in order. Figures 4 and 5 illustrate this concept.

When performing calculations on a `gain.ndarray`, the current `global_slice` is queried when accessing the local data or fetching remote data such that an appropriate `ndarray` data block is returned. Accessing local data and fetching remote data is performed by the `gain.ndarray.access()` and `gain.ndarray.get()` methods, respectively. Figure 6 illustrates how `access()` and `get()` are used. The `ga.access()` function on which `gain.ndarray.access()` is based will always return the entire block owned by the calling process. The returned piece must be further sliced to appropriately match the

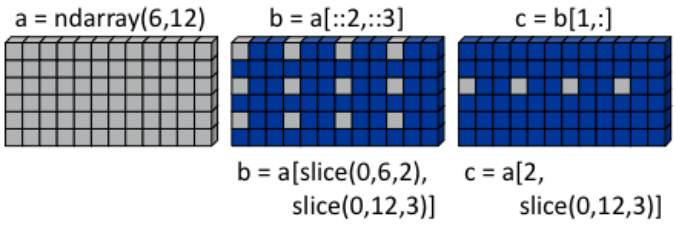

Fig. 4: *Slice arithmetic example 1. Array* `b` *could be created either using the standard notation (top middle) or using the* canonicalized *form (bottom middle). Array* `c` *could be created by applying the standard notation (top right) or by applying the equivalent canonical form (bottom right) to the original array* `a`*.*

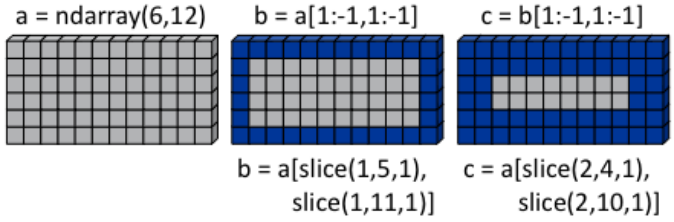

Fig. 5: *Slice arithmetic example 2. See the caption of Figure 4 for details.*

current `global_slice`. The `ga.strided_get()` function on which `gain.ndarray.get()` method is based will fetch data from other processes without the remote processes' cooperation i.e. using one-sided communication. The calling process specifies the region to fetch based on the current view's `shape` of the array. The `global_slice` is adjusted to match the requested region using slice arithmetic and then transformed into a `ga.strided_get()` request based on the global, original shape of the array.

Recall that GA allows the contiguous, process-local data to be accessed using `ga.access()` which returns a C-contiguous `ndarray`. However, if the `gain.ndarray` is a view created by a slice, the data which is accessed will be contiguous while the view is not. Based on the distribution of the process-local data, a new slice object is created from the `global_slice` and applied to the accessed `ndarray`, effectively having applied first the `global_slice` on the global representation of the distributed array followed by a slice representing the process-local portion.

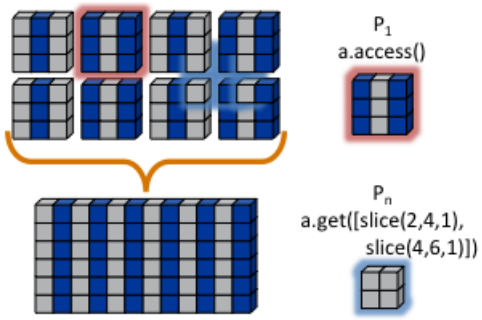

Fig. 6: `access()` *and* `get()` *examples. The current* `global_slice`*, indicated by blue array elements, is respected in either case. A process can access its local data block for a given array (red highlight). Note that* `access()` *returns the entire block, including the sliced elements. Any process can fetch any other processes' data using* `get()` *with respect to the current* `shape` *of the array (blue highlight). Note that the fetched block will not contain the sliced elements, reducing the amount of data communicated.*

After process-local data has been accessed and sliced as needed, it must then fetch the remote data. This is again done using `ga.get()` or `ga.strided_get()` as above. Recall that one-sided communication, as opposed to two-sided communication, does not require the cooperation of the remote process(es). The local process simply fetches the corresponding array section by performing a similar transformation to the target array's `global_slice` as was done to access the local data, and then translates the modified `global_slice` into the proper arguments for `ga.get()` if the `global_slice` does not contain any `step` values greater than one, or `ga.strided_get()` if the `global_slice` contained `step` values greater than one.

One limitation of using GA is that GA does not allow negative stride values corresponding to the negative `step` values allowed for Python sequences and *NumPy* arrays. Supporting negative `step` values for *GAiN* required special care -- when a negative `step` is encountered during a slice operation, the slice is applied as usual. However, prior to accessing or fetching data, the slice is inverted from a negative `step` to a positive `step` and the `start` and `stop` values are updated appropriately. The `ndarray` which results from accessing or fetching based on the inverted slice is then re-inverted, creating the correct view of the new data.

Another limitation of using GA is that the data distribution cannot be changed once an array is created. This complicates such useful functionality as `numpy.reshape()`. Currently, *GAiN* must make a copy of the array instead of a view when altering the shape of an array.

Translating the `numpy.flatiter` class, which assumes a single address space while translating an N-dimensional array into a 1D array, into a distributed form was made simpler by the use of `ga.gather()` and `ga.scatter()`. These two routines allow individual data elements within a GA to be fetched or updated. Flattening a distributed N-dimensional array which had been distributed in blocked fashion will cause the blocks to become discontiguous. Figure 7 shows how a 6×6 array might be distributed and flattened. The `ga.get()` operation assumes the requested patch has the same number of dimensions as the array from which the patch is requested. Reshaping, in general, is made difficult by GA and its lack of a redistribute capability. However, in this case, we can use `ga.gather()` and `ga.scatter()` to fetch and update, respectively, any array elements in any order. `ga.gather()` takes a 1D array-like of indices to fetch and returns a 1D `ndarray` of values. Similarly, `ga.scatter()` takes a 1D array-like of indices to update and a 1D array-like buffer containing the values to use for the update. If a `gain.flatiter` is used as the output of an operation, following the owner-computes rule is difficult. Instead, pseudo-owners are assigned to contiguous slices of the of 1D view. These pseudo-owners gather their own elements as well as the corresponding elements of the other inputs, compute the result, and scatter the result back to their own elements. This results in additional communication which is otherwise avoided by true adherence to the owner-computes rule. To avoid this inefficiency, there are some cases where operating over `gain.flatiter` instances can be optimized, for example with `gain.dot()` if the same `flatiter` is passed as both inputs, the `base` of the `flatiter` is instead multiplied together element-wise and then the `gain.sum()` is taken of the resulting array.

Fig. 7: *Flattening a 2D distributed array. The block owned by a process becomes discontiguous when representing the 2D array in 1 dimension.*

Evaluation

The success of *GAiN* hinges on its ability to enable distributed array processing in *NumPy*, to transparently enable this processing, and most importantly to efficiently accomplish those goals. Performance Python [Ram08] "perfpy" was conceived to demonstrate the ways Python can be used for high performance computing. It evaluates *NumPy* and the relative performance of various Python extensions to *NumPy*. It represents an important benchmark by which any additional high performance numerical Python module should be measured. The original program `laplace.py` was modified by importing `ga.gain` in place of `numpy` and then stripping the additional test codes so that only the `gain` (`numpy`) test remained. The latter modification makes no impact on the timing results since all tests are run independently but was necessary because `gain` is run on multiple processes while the original test suite is serial. The program was run on the chinook supercomputer at the Environmental Molecular Sciences Laboratory, part of Pacific Northwest National Laboratory. Chinook consists of 2310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Opteron processors. Each node has 32 Gbytes of memory for 4 Gbytes per core. Fast communication between the nodes is obtained using a single rail Infiniband interconnect from Voltaire (switches) and Melanox (NICs). The system runs a version of Linux based on Red Hat Linux Advanced Server. *GAiN* utilized up to 512 nodes of the cluster, using 4 cores per node.

In Figure 8, *GAiN* is shown to scale up to 2K cores on a modest problem size. *GAiN* is also able to run on problems which are not feasible on workstations. For example, to store one 100,000x100,000 matrix of double-precision numbers requires approximately 75GB.

During the evaluation, it was noted that a lot of time was spent within global synchronization calls e.g. `ga.sync()`. The source of the calls was traced to, among other places, the vast number of temporary arrays getting created. Using GA statistics reporting, the original `laplace.py` code created 912 arrays and destroyed 910. Given this staggering figure, an array cache was created. The cache is based on a Python `dict` using the shape and type of the arrays as the keys and stores discarded GA instances represented by the GA integer handle. The number of GA handles stored per shape and type is referred to as the cache depth. The `gain.ndarray` instances are discarded as usual. Utilizing the cache keeps the GA memory from many allocations and deallocations but primarily avoids many synchronization calls. Three cache depths were tested, as shown in Table 1. The trade-off of using this cache is that if the arrays used by an application vary wildly in size or type, this cache will consume too much memory. Other hueristics could be developed to keep the cache from using too much memory e.g. a maximum size of the cache, remove the least used arrays, remove the least recently used. Based on the success of the GA cache, it is currently used by *GAiN*.

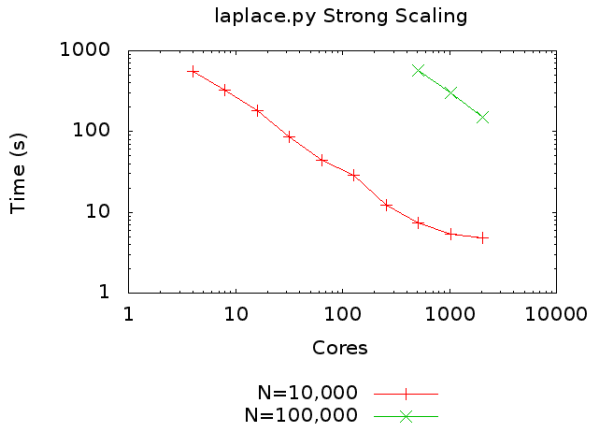

Fig. 8: *`laplace.py` for N=10,000 and N=100,000. For N=10,000, one matrix of double-precision numbers is approximately 0.75GB. For this problem,* GAiN *scales up to 2K cores. For N=100,000, one matrix of double-precision numbers is approximately 75GB. In addition to handling this large-scale problem,* GAiN *continues to scale up to 2K cores.*

No Cache	Depth-1 Cache	Depth-2 Cache	Depth-3 Cache
912/910	311/306	110/102	11/1

TABLE 1: *How array caching affects GA array creation/destruction counts when running `laplace.py` for 100 iterations. The smaller numbers indicate better reuse of GA memory and avoidance of global synchronization calls, at the expense of using additional memory.*

Conclusion

GAiN succeeds in its ability to grow problem sizes beyond a single compute node. The performance of the perfpy code and the ability to drop-in *GAiN* without modification of the core implementation demonstrates its utility. As described previously, *GAiN* allows certain classes of existing *NumPy* programs to run using *GAiN* with sometimes as little effort as changing the import statement, immediately taking advantage of the ability to run in a cluster environment. Once a smaller-sized program has been developed and tested on a desktop computer, it can then be run on a cluster with very little effort. *GAiN* provides the groundwork for large distributed multidimensional arrays within *NumPy*.

Future Work

GAiN is not a complete implementation of the *NumPy* API nor does it represent the only way in which distributed arrays can be achieved for *NumPy*. Alternative parallelization strategies besides the owner-computes rule should be explored. GA allows for the get-compute-put model of computation where ownership of data is largely ignored, but data movement costs are increased. Task parallelism could also be explored if load balancing becomes an issue. The GA cache should be exposed as a tunable parameter. Alternative temporary array creation strategies could be developed such as lazy evaluation.

Acknowledgment

A portion of the research was performed using the Molecular Science Computing (MSC) capability at EMSL, a national scientific user facility sponsored by the Department of Energy's Office

of Biological and Environmental Research and located at Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

REFERENCES

[Apr09] E. Apra, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas. *Liquid water: obtaining the right answer for the right reasons*, Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis, 66:1-7, 2009.

[Asc99] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*, UCRL-MA-128569, 1999.

[Beh11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. *Cython: The Best of Both Worlds*, Computing in Science Engineering, 13(2):31-39, March/April 2011.

[Buy99] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*, Vol. 1, Prentice Hall PTR, 1 edition, May 1999.

[Dai09] J. Daily. *GAiN: Distributed Array Computation with Python*, Master's thesis, Washington State University, Richland, WA, August 2009.

[Dal05] L. Dalcin, R. Paz, and M. Storti. *MPI for python*, Journal of Parallel and Distributed Computing, 65(9):1108-1115, September 2005.

[Dal08] L. Dalcin, R. Paz, M. Storti, and J. D'Elia. *MPI for python: Performance improvements and MPI-2 extensions*, Journal of Parallel and Distributed Computing, 68(5):655-662, September 2005.

[Dot04] Y. Dotsenko, C. Coarfa,. and J. Mellor-Crummmey. *A Multi-Platform Co-Array Fortran Compiler*, Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, 29-40, 2004.

[Dub96] P. F. Dubois, K. Hinsen, and J. Hugunin. *Numerical Python*, Computers in Physics, 10(3), May/June 1996.

[Ede07] A. Edelman. *The Star-P High Performance Computing Platform*, IEEE International Conference on Acoustics, Speech, and Signal Processing, April 2007.

[Eit07] B. Eitzen. *Gpupy: Efficiently using a gpu with python*, Master's thesis, Washington State University, Richland, WA, August 2007.

[Gra09] B. Granger and F. Perez. *Distributed Data Structures, Parallel Computing and IPython*, SIAM CSE 2009.

[Gro99a] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, second edition, MIT Press, November 1999.

[Gro99b] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, 1999.

[Har99] R. J. Harrison. *Global Arrays Python Interface*, http://www.emsl.pnl.gov/docs/global/old/pyGA/, December 1999.

[Hus98] P. Husbands and C. Isbell. *The Parallel Problems Server: A Client-Server Model for Interactive Large Scale Scientific Computation*, 3rd International Meeting on Vector and Parallel Processing, 1998.

[Nie00] J. Nieplocha, J. Ju, and T. P. Straatsma. *A multiprotocol communication support for the global address space programming model on the IBM SP*, Proceedings of EuroPar, 2000.

[Nie05] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang. *Exploiting processor groups to extend scalability of the GA shared memory programming model*, Proceedings of the 2nd conference on Computing Frontiers, 262-272, 2005.

[Nie06] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. *Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit*, International Journal of High Performance Computing Applications, 20(2):203-231, 2006.

[Nie10] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and J. Ju. *The Global Arrays User's Manual*.

[Oli06] T. E. Oliphant. *Guide to NumPy*, http://www.tramy.us/, March 2006.

[Pal07] W. Palm III. *A Concise Introduction to Matlab*, McGraw-Hill, 1st edition, October 2007.

[Pan06] R. Panuganti, M. M. Baskaran, D. E. Hudak, A. Krishnamurthy, J. Nieplocha, A. Rountev, and P. Sadayappan. *GAMMA: Global Arrays Meets Matlab*, Technical Report. ftp://ftp.cse.ohio-state.edu/pub/tech-report/ 2006/TR15.pdf

[Per07] F. Perez and B. E. Granger. *IPython: a System for Interactive Scientific Computing*, Computing in Science Engineering, 9(3):21-29, May 2007.

[Pnl11] Global Arrays Webpage. http://www.emsl.pnl.gov/docs/global/

[Ram08] P. Ramachandran. *Performance Python*, http://www.scipy.org/PerformancePython, May 2008.

[Ras04] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, and E. Jones. *Co-array Python: A Parallel Extension to the Python Language*, Euro-Par, 632-637, 2004.

[Zim88] H. P. Zima, H. Bast, and M. Gerndt. *SUPERB: A tool for semi-automatic MIMD/SIMD Parallelization*, Parallel Computing, 6:1-18, 1988.

Vision Spreadsheet: An Environment for Computer Vision

Scott Determan‡*

Abstract—Vision Spreadsheet is an environment for computer vision. It combines a spreadsheet with computer vision and scientific python. The cells in the spreadsheet are images, computations on images, measurements, and plots. There are many built in image processing and machine learning algorithms and it extensible by writing python functions and importing them into the spreadsheet.

Index Terms—computer vision, spreadsheet, OpenCV

Introduction

Vision Spreadsheet is an application designed to explore and solve computer vision problems. It provides a visual environment and a familiar computational tool set to enable creative prototyping of computer vision algorithms. A novel interface using a spreadsheet of images encourages interactive and exploratory algorithm design. Computational scientists can leverage their existing knowledge of python, NumPy, SciPy, OpenCV, VIGRA, and other familiar technologies. Vision Spreadsheet aims to make the techniques of computer vision accessible to a wider audience.

Vision Spreadsheet is modeled after familiar numerical spreadsheets, such as MS Excel and Apple Numbers. In a numerical spreadsheet, each cell contains a number, with cells relating to each other by numerical expressions. In Vision Spreadsheet, each cell contains an image, with cells relating to each other by computer vision operations. As in a traditional spreadsheet, changes propagate automatically through the cells. Complex vision algorithms can be built-up cell-by-cell, interactively, with continuous visual feedback into the intermediate steps.

The cells within Vision Spreadsheet relate to each other through expressions that operate on images. For example, if the image in cell a2 is the dilation of the image in cell a1, this is expressed as "dilate(a1)". The power of these expressions comes from the large library of functions available, including all of the image processing and machine learning algorithms from OpenCV. Furthermore, users can easily add their own functions using python, NumPy, and SciPy.

Vision Spreadsheet provides many tools to make it easier to explore the solution space of a vision problem.

- source images can be loaded, reload or looped through
- all cell expression can be edited interactively
- all changes to cells are automatically propagated through the spreadsheet
- function parameters can be bound to GUI controls for interactive exploration
- cells can contain graphs and tables containing measurements and statistics from images

Vision Spreadsheet is the product of years of development and many more years of experience working in the field of computer vision. I feel it provides an excellent environment for exploring solutions to computer vision problems. It is difficult to get a feel for an interactive environment by reading a paper. Visit http://visionspreadsheet.com to download the application for free or to watch videos of Vision Spreadsheet in action.

Overview of Vision Spreadsheet

Figure 1 shows a screen shot of Vision Spreadsheet. There are four main areas to the GUI: the grid of cells, the current cell's statement, the shell, and the current cell's GUI controls.

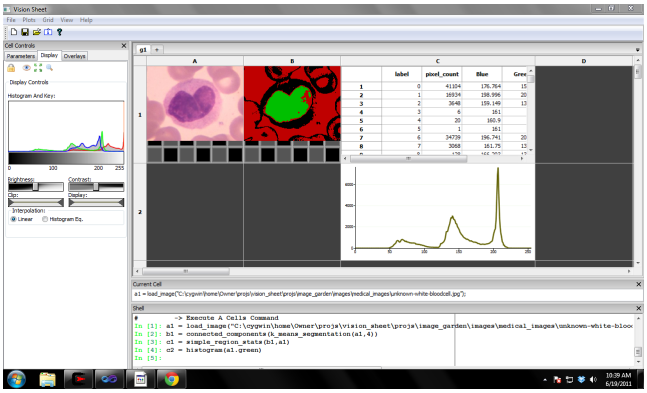

Fig. 1: *Vision Spreadsheet cells contain images, measurements, and plots.*

Numerical spreadsheets contain a grid of numbers and labels. Vision Spreadsheet's grid of cells contains images, computations on images, measurements, and plots. In a numerical spreadsheet, if a cell contains the sum of a column of numbers and a number in the column changes then the total automatically updates itself. Similarly, in Vision Spreadsheet if a cell changes (for instance by loading a new image or changing an algorithm parameter) then all of the cells that depend on the changed cell will update themselves.

* *Corresponding author: scott.determan@gmail.com*
‡ *Vision Spreadsheet*

The current cell's statement is a single-line control used to show what statement was used to create the current cell. This statement may be edited and updated in this control.

The shell is a modified IPython shell used to specify what a cells contains. The shell is also used to write new spreadsheet functions in python.

The GUI controls area contains display parameters, overlays, and controls bound to algorithm parameters for the current cell.

Specifying a Cell's Content

Just like in a numerical spreadsheet, the content of each cell in the spreadsheet grid is defined by an expression. Expressions are entered by typing them into the shell or current cell's statement control. The syntax of Vision Spreadsheet's cell expressions should feel familiar to any spreadsheet user. But unlike a numerical spreadsheet, Vision Spreadsheet's expressions operates on images. A typical statement looks like this:

```
some_cell = some_function(parameter1,parameter2)
```

For example, to define cell b1 as the erosion of the image in cell a1 you would enter the following expression into the shell:

```
b1 = erode(a1)
```

After entering this expression, cell b1 will display the image which is an erosion of the image in cell a1. If you manually load a new image into cell a1, then the image in cell b1 will automatically update as the erosion of the new image.

The power in the expression language comes from the large library of available computer vision functions. In fact, all of the image processing and machine learning functions from OpenCV are available. This allows professionals to leverage their existing knowledge of this powerful library.

The arithmetic operators are available and follow the usual syntax and precedence rules. A typical call with an operator looks like (where someop is +, -, <, etc.):

```
some_cell = parameter1 someop parameter2
```

Functions may be nested, so one way to run a morphological open would be:

```
b1 = dilate(erode(a1))
```

Morphological open is already a built in functions; the above was only an example.

There are also a few special functions, like if and select.

Vision Spreadsheet supports multiple tabs per sheet. Cells in another tab are in another namespace, and can be referenced using the namespace syntax:

```
namespace_name::variable_name
::variable_name # global namespace
```

Sheets start with g and are sequentially numbered, so the following code is used to refer to sheet g1 cell a1:

```
g1::a1
```

Literal data sets are specified with the following syntax:

```
[1,2,3,4]
[[1,2,3],[4,5,6],[7,8,9]]
```

Literal dictionaries are specified with the following syntax:

```
{'name':'Scott','weight':150,'location':[512,700]}
```

Keys must be a string. Values can be any supported data type (dictionaries, data sets, data frames, etc.).

Expressions can be an arbitrarily complex combination of functions and arithmetic operators. But just like in a numerical spreadsheet, cell expressions work best as simple one-line expressions. For more complex programs, use python mode within Vision Spreadsheet.

Binding Parameters to GUI Controls

A primary goal of Vision Spreadsheet is to allow interactive exploration of vision problems. One of the most powerful tools to do this is to bind GUI controls to parameters in a cell expression. This allows users to have a value in a cell expression that comes from a GUI control, such as a slider control. The user can manipulate the GUI control to affect the value in the expression. Because Vision Spreadsheet automatically propagates this change through the spreadsheet, users can very quickly see the effect that a particular parameter has on the result of an algorithm.

The best way to explain this feature is to look at an example. Consider thresholding an image. There are a couple of threshold operators, but the simplest is the '>.' operator. Load an image in cell a1. Next, threshold it by typing:

```
b1 = a1 >. 128
```

This creates an image where values greater than 128 are set to 255 and values less than or equal to 128 are set to zero. One way to decide on a threshold value would be to keep typing in numbers until the threshold image looked good. A better way is to bind the parameter to a GUI control, like a slider. The following command does this:

```
b1 = a1 >. slider(128,0,255)
```

This creates a slider with a default value of 128, a min value of 0, and a max value of 255. If the threshold image is the current cell, then the cell controls pane on the left of the GUI will contain a slider (see figure 2). This slider is used to interactively change the parameter to the threshold function.

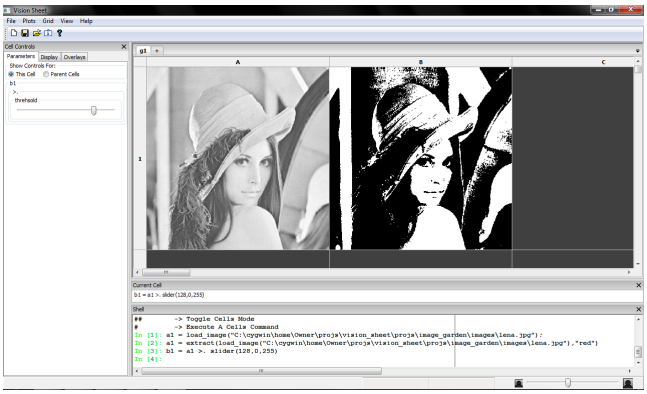

Fig. 2: *GUI controls may be bound to algorithm parameters. Here a slider is bound to a threshold.*

There are many other types of GUI controls that may be bound to parameters, such as: radio buttons, sliders, spin controls, combo boxes and movie controls (radio button are particularly useful to bind to file names so different images may be easily loaded into a cell).

Multiple GUI controls may be used to control a single function parameter. If the function calls to create the GUI controls are nested, then they will control the same parameter. For example, to have a spin control and a slider control the threshold:

```
b1 = a1 >. spin(slider(128,0,255))
```

Python Mode

Python is a fantastic language for exploratory computer vision. Vision Spreadsheet is tightly integrated with python and gives users full access to an IPython shell within Vision Spreadsheet. This integration gives users all of the power and familiarity of python combined with the visual feedback and interactivity of Vision Spreadsheet. Users can extend Vision Spreadsheet by adding new functions they implement in python. Users also have full access to the Vision Spreadsheet environment from within python, allowing them to access and update cells interactively from within the IPython shell.

To toggle the shell to/from IPython mode, type "##" in the shell. Inside the IPython shell, you will have access to the vis_sheet module. The vis_sheet module provides full access to the Vision Spreadsheet environment from within python. The IPython shell at the bottom of the GUI supports two modes, cells mode and python mode. To toggle between the two modes, type '##' and hit return. Cells mode is the default mode. Python mode is just a regular IPython shell with two differences: typing '##' will toggle to cells mode and there is a module called 'vis_sheet' that can be used to interact with the spreadsheet.

There are at least two interesting activities to do in python mode:

1) Extend the spreadsheet with new functions.
2) Get values from the spreadsheet, muck around with them interactively in python, and set the values back into the spreadsheet.

Here is how to add a new function to the spreadsheet. Change to python mode by typing '##'. The shell should now have a black background. Define a subtraction function as follows:

```
def my_subtract(a,b): return a-b
import vis_sheet
vis_sheet.add_python_op(my_subtract)
```

Change back to cells mode by typing '##' (the shell should now have a white background). Load an image in cell a1, erode it an put it in b1, and subtract b1 from a1 using the new function:

```
c1 = my_subtract(a1,b1)
```

Cell c1 will contain the edges from the image in cell a1. Note that the images in the spreadsheet are automatically converted to NumPy arrays before they are passed to user defined functions. The parameters a and b will be NumPy arrays. If the result is a NumPy array, it will automatically be converted to an image.

To get or set values in the spreadsheet from python mode, use the following functions:

```
import vis_sheet
vis_sheet.get_var_data('a1')
vis_sheet.set_var_data('b1',some_python_var)
```

Data Structures

There are three main data structures in vision spreadsheet: images, data frames, and statistical models.

Images are the most important data structure. An image is a two dimensional array of vectors. All the elements of an image are of the same numeric type. Images with element types of uchar through double are supported. Many image types are supported, for example: grayscale, color (rgb, brg, hsi, cie lab, etc.), and depth images (from the Kinect camera, for example). When an image is passed to a user defined python function it is automatically converted into a NumPy array.

Data frames are modeled after R's data frame structure. Data frames are used to store measurements on images and to overlay images with shapes and regions of interest. It is a table where each column in the table may have a different type. So a single data frame may have a column of numbers and a column of strings. Supported column types are: numeric (uchar through double), boolean, string, and region of interest. Like R's data frames, rows may contain missing data. Data frames also support R's notion of factor columns. Factor columns are usually used to specify responses when training classifiers. Unlike R, vision spreadsheet supports grouping columns into a hierarchy. This is useful for storing higher-level objects in a data frame. For example, rectangles are stored in a data frame by grouping together four numeric columns. These rectangles may then be overlaid and edited on an image.

The last major data structure is a statistical model. Statistical models are used to classify objects in images. There are two main functions to a statistical model: train and predict. The train function takes a statistical model, a data frame of features, and a data frame of responses. It returns the newly trained model. The predict function takes a model and data frame. It returns a prediction for each row in the data frame.

There are other data types in vision spreadsheet, but many problems in computer vision can be solved using only these three data types.

Conclusion

I described a new environment for interactively working with computer vision. I am optimistic that this will be a useful and productive environment for many types of users. However, at this point no one except myself has used Vision Spreadsheet. The key to making the environment useful is to have real users try to solve real problems with it. My goal in presenting this paper is to get people using the spreadsheet so they can provide the feedback I need to make Vision Spreadsheet as useful as I know it can be. Please try it out.

I had planned on releasing Vision Spreadsheet shortly before the conference. I did not make this deadline, but I am very close. When it is released, you can go to http://visionspreadsheet.com to download it for free.

Thank You

I owe thanks to many great open source projects. I especially want to thank the following projects (alphabetical order): ANTLR[1], boost[2], CMake[3], IPython[4] [IPy], OpenKinect[5], NumPy[6], OpenCV[7], python[8], SciPy[9], SWIG[10], VIGRA[11],

wxPython[12], and wxWidgets[13].

REFERENCES

[IPy] Fernando Perez, Brian E. Granger, "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.

1. http://www.antlr.org ANTLR is used to build the parser for the cells language.
2. http://www.boost.org. Boost is used for many utility routines in the c++ code.
3. http://www.cmake.org. CMake is the build/test/package system.
4. http://ipython.org IPython is the shell.
5. http://openkinect.org OpenKinect is the interface to the Kinect camera.
6. http://numpy.scipy.org NumPy arrays are used to interface between the internal data structures in vision spreadsheet and python.
7. http://opencv.willowgarage.com OpenCV provides many of the image processing and machine learning algorithms.
8. http://python.org. The spreadsheet is extended through python.
9. http://www.scipy.org SciPy makes it easy and efficient for the user to extend the spreadsheet and manipulate images and data frames.
10. http://www.swig.org SWIG is used to wrap OpenCV functions into Vision Spreadsheet.
11. http://hci.iwr.uni-heidelberg.de/vigra VIGRA provides many image processing algorithms.
12. http://www.wxpython.org. wxPython is the python interface to the wxwidgets library.
13. http://wxwidgets.org. wxWidgets is the GUI library.

Constructing scientific programs using SymPy

Mark Dewing[‡*]

Abstract—We describe a method for constructing scientific programs where SymPy is used to model the mathematical steps in the derivation. With this workflow, each step in the process can be checked by machine, from the derivation of the equations to the generation of the source code. We present an example based on computing the partition function integrals in statistical mechanics.

Index Terms—SymPy, code generation, metaprogramming

Introduction

Writing correct scientific programs is a difficult, largely manual process. Steps in the process include deriving of the constituent equations, translating those into source code, and testing the result to ensure correctness. One challenging aspect of testing is untangling the cause of errors. For example, if the result appears incorrect, it is hard to determine whether the problem is with the algorithm, or a mistake was made in the derivation, or a simple transcription error in writing the source code. Confidence in the correctness of the program can be increased if these steps can be checked by computer.

The process of scientific programming also includes the pursuit of performance. Often the code needs to be heavily modified or rewritten to take best advantage of various target systems. Each modification introduces the possibility of further errors, and must be checked.

A standard approach to create a high-level description of the problem that is specialized to the particular domain, often through a Domain Specific Language (DSL), and then transform this to a representation in a general-purpose programming language. This is used in systems such as FEniCS [FEniCS] (representing PDE's) and the Tensor Contraction Engine [TCE] (representing matrix operations in quantum chemistry).

Since the specifications for scientific software are expressed largely as mathematical equations, our high-level description will be formed from a symbolic mathematics representation. A symbolic mathematics package is well-suited for this representation. In addition to this representation of the high-level description, we will model the derivation steps that lead to a computationally useful form.

The approach taken in this work is to operate on a symbolic representation of the scientific program, and then programmatically transform it into the target system. The specifications for scientific software are expressed largely as equations, and are ideally suited for a symbolic mathematics package. We use SymPy [SymPy], a symbolic mathematics package written in Python, for this part of the process.

The target system is likely a source code representation (C, Fortran, Python, etc), but could encompass more than that. For instance, C might be used to target the CPU or GPU, but the source code might look quite different in those cases. Or the user may call different libraries for the same function to compare performance.

* Corresponding author: markdewing@gmail.com
‡ Intel

Modeling Derivations

The first goal is to model on the computer a set of steps similar to those used when manually performing the derivation. Deriving the equations using in scientific software is similar to a proof where there is a series of logically justified steps connecting each expression until the final result is reached.

The OpenMathDocuments (OMDoc) project [OMDoc] is a representation for mathematics that describes a higher level than expressions in MathML. For instance, it has representations for proofs and lemmas. Similarly, for scientific computation we need to represent structures a higher level. One major difference between proofs and derivations in scientific software is that some steps are approximations.

The steps can be categorized as exact transformations, approximations, or specializations. An exact transform leaves the equality satisfied. Some types of exact transforms are rearranging terms, multiplying by factors, and identities (which operates only on one side of the equation). A specialization is specifying a physical or model parameter, such as the number of spatial dimensions, the number of particles, interaction potential, etc.

Finally, the results can be displayed in rendered mathematics (we use MathML or MathJax in web pages) to make the operation of each step and the results clearly visible.

Implementations

Modeling Derivations

For the implementation, the basic class named `derivation` has a constructor that takes an initial equation (lhs and rhs). The primary method is `add_step`, which takes an operation (or list of operations) to perform and a textual description of the operation(s). There are series of classes for various operations, such `approx_lhs`, which replaces the left-hand side of the equation with a new value. Also there is `add_term`, which adds the same term to both sides of the equation.

Code Generation

It is easy to start generating code by simply printing the statements of the target language. However, for greater generality we will use a model of the target language. Currently this work has (incomplete) language models for Python and C.

At the lowest level of transforming expressions, we developed a pattern-matching syntax that concisely captures some of the SymPy idioms.

The `Match` object matches a SymPy expression. The `__call__` method matches the first argument as the type of the expression. Subsequent arguments are variables to be bound to arguments. If the argument is a tuple, it is matched recursively on that argument. In this way the pattern for a tree structure can be built up concisely.

Variables that can match (for later binding) are members of an `AutoVar` class. This class creates member variables upon first access, and they are bound when the match succeeds.

Here is an example fragment of part of the SymPy to Python expression transformation, that matches addition, subtraction, and the reciprocal. SymPy normalizes subtraction as adding two expressions where the subtractand is multiplied by negative one. (That is, $a-b$ is represented as $-1*b+a$). Matching subtraction requires a nested pattern, which is shown here as well.

```
from sympy import Add, Mul, Pow, S
from derivation_modeling.codegen.lang_py import \
    py_expr, py_num
from derivation_modeling.codegen.pattern_match \
     import AutoVar, Match

class expr_to_py(object):
 def __call__(self, e):
   v = AutoVar()
   m = Match(e)

   # subtraction
   if m(Add, (Mul, S.NegativeOne, v.e1), v.e2):
     return py_expr(py_expr.PY_OP_MINUS,
        expr_to_py(v.e2), expr_to_py(v.e1))

   # addition
   if m(Add, v.e1, v.e2):
        return py_expr(py_expr.PY_OP_PLUS,
            expr_to_py(v.e1), expr_to_py(v.e2))

   # reciprocal
   if m(Pow, v.e2, S.NegativeOne):
        return py_expr(py_expr.PY_OP_DIVIDE,
            py_num(1.0), expr_to_py(v.e2))
```

Examples

Simple derivation

The Euler method is the simplest method for solving a differential equation. The steps involve a finite difference approximation to the derivative, rearranging terms, and the result is

$$f_1 = f_0 + h * 2 * x$$

The derivation is the following code:

```
from sympy import Function, Symbol, diff, sympify
from derivation_modeling import derivation, \
    approx_lhs, mul_factor, add_term

f = Function('f')
x = Symbol('x')
df = diff(f(x),x)
fd = sympify('(f_1 - f_0)/h')
d = derivation(df,2*x)

d.add_step(approx_lhs(fd),
    'Approximate derivative with finite difference')
d.add_step(mul_factor(h),'Multiply by h')
d.add_step(add_term(f0),'Move f_0 term to left side')
```

This can be output to MathML (or MathJax) for display in a web browser, which looks approximately like the following:

$$\frac{\partial}{\partial x}\mathrm{f}(x) = 2 * x$$

Approximate derivative with finite difference

$$\frac{f_1 - f_0}{h} = 2 * x$$

Multiply by h

$$f_1 - f_0 = 2 * xh$$

Move f_0 term to left side to get the final result

$$f_1 = f_0 + 2 * xh$$

Quadrature

For one of the simplest quadrature formulas, we use the trapezoidal rule [Trapezoid]. The derivation part consists of starting from the rule for single interval, and extending it to a series of intervals. (The rules for a single interval can be derived from interpolating polynomials, but we didn't start there)

The starting point for the derivation in Python is to define all the symbols, and the initial expression, then manipulate the expression so the function evaluation of each point is used only once.

```
from sympy import Symbol, Function, IndexedBase, Sum
from derivation_modeling import derivation, identity

i = Symbol('i',integer=True)
n = Symbol('n',integer=True)

I = Symbol('I')
f = Function('f')
h = Symbol('h')
x = IndexedBase('x')

# definitions of split_sum, adjust_limits,
#    peel_terms not shown
# split_sum - expand the sum of terms into a term of sums
# adjust_limits - adjust the expressions in the
#  summation variable.  This allows matching
#  the index used in the summand among different sums.
# peel_terms - move terms from the either end of the sum
#   to be an explicit term this allows the sum limits
#   to match and be combined.

trap = derivation(I, Sum(h/2*(f(x[i])+f(x[i+1])), (i,1,n)))
trap.add_step(identity(split_sum),'Split sum')
trap.add_step(identity(adjust_limits),'Adjust limits')
trap.add_step(identity(peel_terms),'Peel terms')
```

The LaTeX representation for the steps was copied from the generated output.

Start with a sum of single interval formulas

$$I = \sum_{i=1}^{n} \frac{1}{2} h\left(\mathrm{f}(x[i]) + \mathrm{f}(x[i+1])\right)$$

Split into two sums ('Split sum')

$$I = \sum_{i=1}^{n} \frac{1}{2} h\,\mathrm{f}(x[i]) + \sum_{i=1}^{n} \frac{1}{2} h\,\mathrm{f}(x[i+1])$$

Adjust the limits so the functions in the sum have compatible indices ('Adjust limits')

$$I = \sum_{i=0}^{n-1} \frac{1}{2} h \mathrm{f}(x[i]) + \sum_{i=1}^{n} \frac{1}{2} h \mathrm{f}(x[i])$$

Peel off some terms so the sum limits match, and combine the sums. ('Peel terms')

$$I = \frac{1}{2} h \mathrm{f}(x[0]) + \frac{1}{2} h \mathrm{f}(x[n]) + 2 \sum_{i=1}^{n-1} \frac{1}{2} h \mathrm{f}(x[i])$$

Now we have the final expression and can move to the transformation step. The approach to multiple dimensional integrals will be iterated one-dimensional integrals.

Partition Function

We start with the partition function from statistical mechanics [Partition]. It incorporates the interactions between particles (think of particles in a box), and contains all the thermodynamic information about a system. The dimension of the integral rises with the number of particles. The complexity for the convergence of grid-based methods is exponential in the number of dimensions, and they quickly become overwhelmed. The convergence of Monte Carlo methods is independent of dimension, and are commonly used to compute these integrals. However, it would be still be useful to use a grid method for a small number of particles as a way to check the Monte Carlo algorithms.

The derivation starts as follows:

```
partition_function =
   derivation(Z,Integral(exp(-V/(k*T)),R))
```

Where V is the inter-particle potential, T is the temperature, k is Boltzmann's constant, and Z is the symbol for the partition function. All of these are defined as SymPy `Symbol`.

Once again, the LaTeX has been copied from the output (although some steps have been combined for space)

$$Z = \int e^{-\frac{V}{Tk}} dR$$

It is conventional to work with the dimensionless inverse temperature, $\beta = kT$. Create the definition and insert into the integral.

```
beta_def = definition(Beta, 1/(k*T), T)
partition.function.add_step(
    replace_definition(beta_def),
    'Insert definition of beta')
```

The rendered output is

$$Z = \int e^{-V\beta} dR$$

To support multiple child derivations branching from a single parent, there is a method to support starting a new derivation from the final step of the previous one. Specialize to two particles - the `specialize_integral` transform replaces the integration variables, and the the `replace` transform replaces the specified variables (using a SymPy `subs`).

```
n2 = partitition_function.new_derivation()
n2.add_step(specialize_integral(R,(r1,r2)),
    'specialize to N=2')
n2.add_step(replace(V,V2(r1,r2)),
    'replace potential with N=2')
```

The rendered output is

$$Z = \int \int e^{-\beta \mathrm{V}(r_1,r_2)} dr_1 dr_2$$

Change variables and switch to a potential that depends only on the magnitude of the interparticle distance

```
r_cm = Vector('r_cm',dim=2)
r_12 = Vector('r_12',dim=2)

r_12_def = definition(r_12, r2-r1)
r_cm_def = definition(r_cm, (r1+r2)/2)

V12 = Function('V')

n2.add_step(specialize_integral(r1,(r_12,r_cm)),
    'Switch variables')
n2.add_step(replace(V2(r1,r2),V12(r_12)),
    'Specialize to a potential that depends only
     on interparticle distance')
n2.add_step(replace(V12(r_12),V12(Abs(r_12))),
    'Depend only on the magnitude of the distance')
```

The rendered output is

$$Z = \int \int e^{-\beta \mathrm{V}(|r_{12}|)} dr_{12} dr_{cm}$$

Integrate out the center of mass (or fixed coordinate) (This step could be performed by SymPy, but isn't right now)

```
Vol = Symbol('Omega')
n2.add_step(do_integral(Vol, [r_12]),
    'Integrate out r_cm (this step is still a hack)')
```

The rendered output is

$$Z = \Omega \int e^{-\beta \mathrm{V}(|r_{12}|)} dr_{12}$$

Decompose into vector components and specify limits. The `identity` transform modifies the right-hand side of the equation without changing its validity. The `decompose` operation takes an expression involving vectors and replaces it with the expression in terms of vector components. The `add_limits` transform adds upper and lower limits to the previously indefinite integral.

```
L = Symbol('L')
n2.add_step(identity(decompose),
    'Decompose into vector components')
n2.add_step(identity(add_limits(-L/2,L/2)),
    'Add integration limits')
```

The rendered output is

$$Z = \Omega \int_{-L/2}^{L/2} \int_{-L/2}^{L/2} e^{-\beta \mathrm{V}\left(\sqrt{r_{12x}^2 + r_{12y}^2}\right)} dr_{12x} dr_{12y}$$

Specialize to the Lennard-Jones potential

```
lj_expr = 4*(1/r**12 - 1/r**6)
lj_pot = derivation(V(r),lj_expr)
n2.add_step(replace_func(V12,lj_pot.final()),
    'Specialize to the LJ potential')
```

$$\mathrm{V}(r) = \frac{4}{r^{12}} - \frac{4}{r^6}$$

And get

$$Z = \Omega \int_{-\frac{1}{2}L}^{\frac{1}{2}L} \int_{-\frac{1}{2}L}^{\frac{1}{2}L} e^{-\beta \left(\frac{4}{\left(r_{12x}^2 + r_{12y}^2\right)^6} - \frac{4}{\left(r_{12x}^2 + r_{12y}^2\right)^3} \right)} dr_{12x} dr_{12y}$$

Insert numerical values for the box size and temperature.

```
L = 2.0
n2.add_step(replace('L',L),
    'Insert value for box size')
n2.add_step(replace('Omega',L*L),
    'Insert value for box volume')
n2.add_step(replace('beta',1.0),
    'Insert value for temperature')
```

$$Z = 4.0 \int_{-1.0}^{1} \int_{-1.0}^{1} e^{-4.0 \frac{1}{\left(r_{12x}^2 + r_{12y}^2\right)^6} + 4.0 \frac{1}{\left(r_{12x}^2 + r_{12y}^2\right)^3}} dr_{12x} dr_{12y}$$

Now we have an integral that is completely specified numerically[1]. It can be evaluated by an existing quadrature routine in SymPy, by another another package (`scipy.quadrature.dblquad`), or by the trapezoidal rule code we derived earlier.

Code Generation

As an example of the language model, the classic 'Hello World' program in python is

```
from derivation_modeling.codegen.lang_py import
    py_expr, py_expr_stmt, py_function_call, \
    py_function_def, py_if, py_print_stmt, \
    py_stmt_block, py_string, \
    py_var


body = py_stmt_block()

hello_func = py_function_def('hello')
hello_func.add_statement(
   py_print_stmt(py_string("Hello, World")))
body.add_statements(hello_func)
main = py_if(
    py_expr(py_expr.PY_OP_EQUAL,
        py_var('__name__'), py_string('__main__')))
main.add_true_statement(
   py_expr_stmt(py_function_call('hello')))
body.add_statements(main)

f = open('hello_py.py','w')
f.write(body.to_string())
f.close()
```

The generated output is

```
def hello():
   print "Hello, World"
if __name__ == "__main__":
   hello()
```

For C, the program is

```
from derivation_modeling.codegen.lang_c import
   c_block, c_function_call, c_function_def, \
   c_func_type, c_int, c_num, c_return, c_stmt, \
   c_string, pp_include

body = c_block()
body.add_statement(pp_include('stdio.h'))
main_body = c_block()

main = c_function_def(
  c_func_type(c_int('main')), main_body)

main_body.add_statement(
  c_stmt(c_function_call("printf",
          c_string("Hello, World\\n"))))

main_body.add_statement(c_return(c_num(0)))
body.add_statement(main)

f = open('hello_c.c','w')
f.write(body.to_string())
f.close()
```

The generated program is

```
#include <stdio.h>
int main(){
  printf("Hello, World\n");
  return 0;
}
```

The code and examples described here can be found in the author's `derivation_modeling` repository on GitHub:
https://github.com/markdewing/derivation_modeling

Discussion

The example derivations presented here are fairly simple and linear. In reality, the connections are more complex. For instance, one is often interested in multiple properties (energy, pressure, distribution functions) that may branch off the original derivation or have a separate thread of steps, but eventually, for efficiency they should all be evaluated in the same integral.

The pattern-matching style makes the lower levels of expression translation fairly clear, but the the translations at the next level up (combining the source code statements) is not very transparent yet. An important future step is enhancing debugging by making the connections between the code generator and the generated code clearer.

Other Work

For solving partial differential equations, there is FEniCS [FEniCS] project and the SAGA (Scientific computing with Algebraic and Generative Abstractions) project [SAGA] .

Ignition [Ignition],[Terrel11]_ is a library that provides support for writing and combining DSL's for describing problems (or aspects of problems)

Part of this work is modeling the target language for code generation. Several other projects for modeling programming languages include Pivot [Pivot], a project for modeling C++. CodeBoost [CodeBoost] is the code transformation portion of the SAGA system. PyCUDA [PyCUDA] is a potential target system, and it also has an associated model of C and CUDA for generation of code [CodePy]

Conclusions

We presented a snapshot of some work on some software blocks necessary for a system of scientific computing, including modeling a derivation, transforming to a source code representation, and code generation.

References

[CodeBoost] http://codeboost.org/
[CodePy] http://mathema.tician.de/software/codepy
[FEniCS] http://www.fenicsproject.org
[Ignition] http://andy.terrel.us/ignition/
[OMDoc] http://www.omdoc.org
[Partition] http://en.wikipedia.org/wiki/Partition_function_%28statistical_mechanics%29
[Pivot] http://parasol.tamu.edu/pivot/
[PyCUDA] http://mathema.tician.de/software/pycuda
[TCE] Tensor Contraction Engine http://www.csc.lsu.edu/~gb/TCE/
[Terrel11] A. Terrel. *From Equations to Code: Automated Scientific Computing* Computing in Science and Engineering 13(2):78-982, March 2011
[Trapezoid] See http://en.wikipedia.org/wiki/Trapezoidal_rule or any numerical analysis textbook
[SAGA] http://www.ii.uib.no/saga/
[SymPy] http://sympy.org/

1. There is a division-by-zero error at $r = 0$ that must be avoided, either by offsetting one limit slightly, or by capping the potential for small r. This latter step has not been added to the definition of the potential yet.

Using Python, Partnerships, Standards and Web Services to provide Water Data for Texans

Dharhas Pothina‡*, Andrew Wilson‡

Abstract—Obtaining time-series monitoring data in a particular region often requires a significant effort involving visiting multiple websites, contacting multiple organizations and dealing with a variety of data formats. Although there has been a large research effort nationally in techniques to share and disseminate water related time-series monitoring data, development of a usable system has lagged. The pieces have been available for some time now, but a lack of vision,expertise, resources and software licensing requirements have hindered uptake outside of the academic research groups. The Texas Water Development Board is both a data provider and large user of data collected by other entities. As such, using the lessons learned from the last several years of research, we have implemented an expandable infrastructure for sharing water data in Texas. In this paper, we discuss the social, institutional and technological challenges in creating a system that allows discovery, access, and publication of water data from multiple federal, state, local and university sources and how we have used Python to create this system in a resource limited environment.

Index Terms—time-series, web services, waterml, data, wofpy, pyhis, HIS, hydrologic information system, cyberinfrastructure

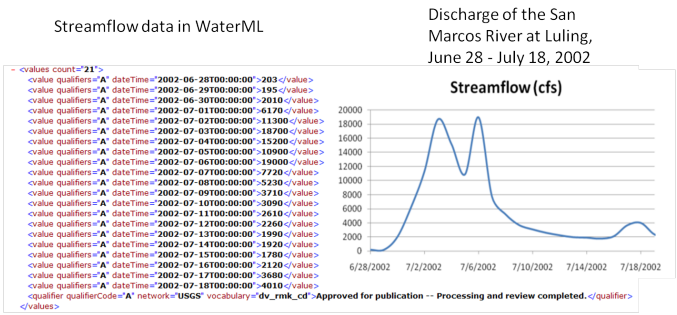

Fig. 1: *An example of a WaterML format file and the data it contains.*

Introduction

A wealth of physical, chemical and biological data exists for Texas' lakes, rivers, coastal bays and estuaries and the near-shore Gulf of Mexico, but unfortunately much of it remains unused by researchers and governmental entities because it is not widely disseminated and its existence not advertised. Historical data is typically stored by multiple agencies in a variety of incompatible formats, often poorly documented. Locating, retrieving and assembling these datasets into a form suitable for use in scientific studies often consumes a significant portion of many studies. Hence, despite having access to this information, much of the data remain underutilized in the hydrologic sciences due in part to the time required to access, obtain, and integrate data from different sources [Goodall2008] .

As both a consumer of water related data delivered by other entities and as a data provider whose data needs to be disseminated to external users the Texas Water Development Board (TWDB) has been researching the use of new technologies for data sharing for several years. The Texas Hydrologic Information System (Texas HIS) was born as a research project funded by the Texas Water Development Board (TWDB) and implemented by the University of Texas at Austin's Center for Research in Water Resources (UTCRWR) as a working prototype to facilitate data discovery, access, and publication of water related time-series monitoring data from multiple entities [Whiteaker2010]. It was built as an extension of the national Hydrologic Information System (HIS) that was developed by the Consortium of Universities for the Advancement of Hydrologic Science, Inc. (CUAHSI) [Tarboton2009]. This prototype proved the viability of using web services along with xml based data format standards to reliably exchange data and showed how once the infrastructure was put in place, powerful standards based tools could be developed to facilitate data discovery and access [Ames2009].

Using Standards - WaterML, WaterOneFlow and the Observations Data Model

CUAHSI-HIS provides web services, tools, standards and procedures that enhance access to more and better data for hydrologic analysis [Tarboton2009]. CUAHSI-HIS has established a web service design called WaterOneFlow as a standard mechanism for the transfer of hydrologic data between hydrologic data servers (databases) and users. Web services streamline the often time-consuming tasks of extracting data from a data source, transforming it into a usable format and loading it in to an analysis environment [Maidment2009]. All WaterOneFlow web services return data in a standard format called WaterML (Figure 1). The specifics of WaterML are documented as an Open Geospatial Consortium, Inc., discussion paper [[Zaslavsky2007].

To publish data in CUAHSI-HIS, a data source provides access to their data via a WaterOneFlow web service. CUAHSI-HIS also includes mechanisms for registering WaterOneFlow web services so that users can discover and use them. Data sources often store

* *Corresponding author: dharhas.pothina@twdb.state.tx.us*
‡ *Texas Water Development Board*

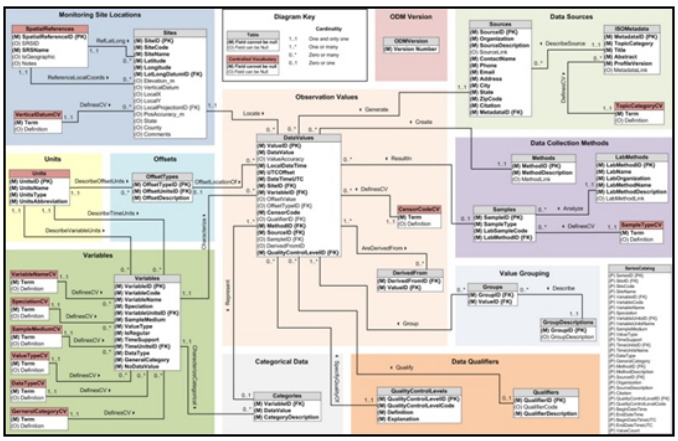

Fig. 2: *The Observations Data Model Schema.*

their data locally in a CUAHSI-HIS Observations Data Model (ODM) database (2), where ODM is a database design for storing hydrologic time series data reported at discrete point locations [Horsburgh2008]. ODM databases, ODM data loaders, and a special version of the WaterOneFlow web service specifically designed to work with ODM as its underlying data source are all available for free on the HIS website (http://his.cuahsi.org).

In addition, CUAHSI-HIS provides several free community supported clients to search and retrieve data from WaterOneFlow compliant services. These include a Microsoft Excel plugin called HydroExcel and a desktop GIS softare called HydroDesktop [Ames2010].

Barriers to Adoption

Although the technology seemed mature and had participation from several universities and a few large federal agencies like the United States Geological Service(USGS), uptake outside of this group was low. Data providers are often resource poor in staff, technical knowledge, time and money. In most cases, they already have a system for collecting, storing and disseminating data that works for their particular needs. In order to convince them to become part of a system like the CUAHSI-HIS, the cost of sharing their data has to be as low as possible and they have to be educated on the benifits their organization will receive through being part of the system. A review of the experiences from building the prototype Texas HIS system showed that there are significant barriers to wide scale adoption.

While there is nothing intrinsic to the CUAHSI-HIS that requires a particular software stack, for historical reasons all currently available software for both serving data and retrieving data from the CUAHSI-HIS system was built on a Microsoft .Net software stack and in some cases also needs commercial licenses. Hence, data providers who could not or did not want to use this software stack needed to write an in-house implementation of WaterOneFlow web services from the ground up. In addition, client side tools were also not cross-platform had were built for specific use cases and could not be easily adapted for alternate needs.

Changing Paradigms

Building a custom implementation of WaterOneFlow web services to attach to a datasource is a non trivial endeavour. It requires and understanding of the web services, XML and the particulars of the WaterML and WaterOneFlow. Hence, the paradigm followed by most participating data providers is to manipulate their data into an ODM database hosted on an MSSQL server. CUAHSI-HIS has a prebuilt WaterOneFlow implementation that can then be used to serve data. This approach requires that the data provider either adopt the ODM as their internal structure for storing data or they must build a translator and periodically dump data from their in-house database to the ODM database on a regular basis. The ODM schema is designed to hold data from multiple sources and hence is often much more complicated than most data providers in-house database schemas. It also excludes data providers that use non Microsoft operating systems.

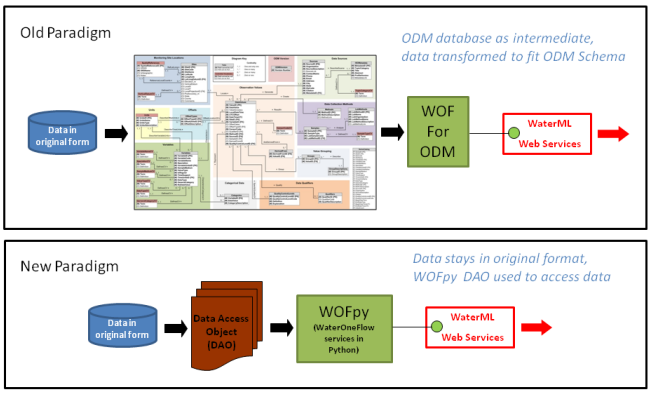

Fig. 3: *Comparison of changing paradigms.*

Lowering these barriers requires flexible cross-platform software that can be relatively easily adapted to each organizations needs. In addition, participation in data sharing should not require large changes to an organizations internal data systems. Based on these requirements, two python modules were developed, WOFpy for serving data as WaterOneFlow services and pyhis as the basis for building customized data access tools.

Using Python to serve water data - WOFpy

WaterOneFlow in Python or WOFpy implements a reduced ODM data model that maps to WaterML objects. It has an implementation of both REST and SOAP web services that are compliant to the WaterOneFlow specification. This is done through the use of the Flask and SOAPlib python packages. On the backend, Data Access Objects (DAO's) are used to connect the services to the underlying database or storage mechanism. Through the use of the sqlalchemy python package DAO's can be written for any database backend that sqlalchemy supports. This allows a large degree of flexibility in attaching the web services to disparate data sources. Figure 4 shows the basic architecture of WOFpy.

WOFpy can be used to serve data from flat files, a variety of database backends and even as an on-the-fly translator of web services that use other standards.

Using python to retrieve data - pyhis

Existing CUAHSI-HIS clients are not cross-platform and are GUI based, pyhis is a command line python package that was developed

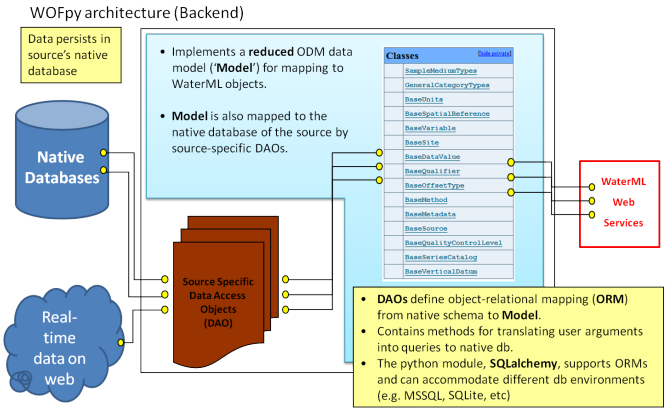

Fig. 4: Architecture of WOFpy.

to allow access to WaterOneFlow services with requiring knowledge of how the underlying web services architecture works. Pyhis uses suds to retrieve data and caches downloaded data to a local sqlite database using sqlachemy. Using pyhis more complicated scripts can be built to conduct spatial analysis or retrieve data automatically for use in real time forcast models.

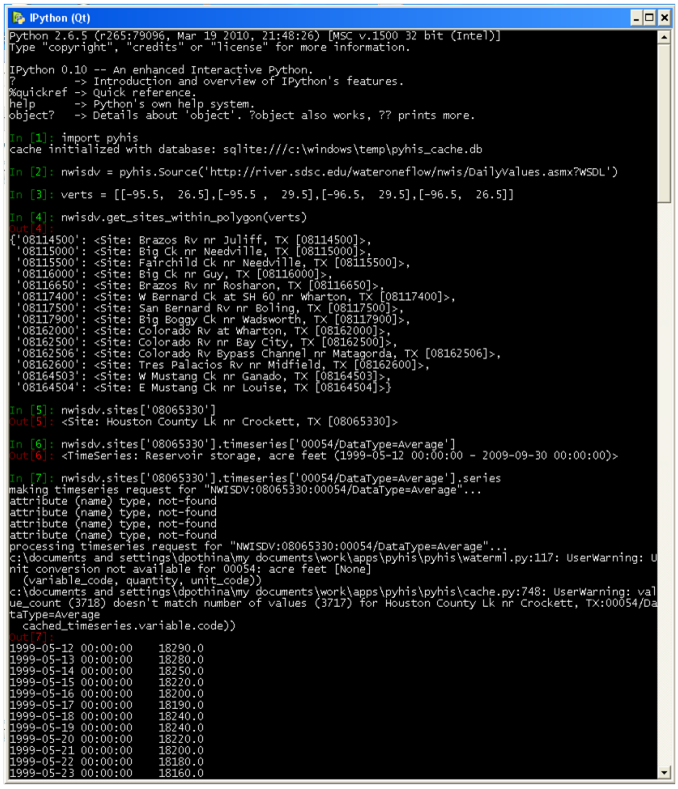

Fig. 5: Example of using pyhis within ipython to retrieve data from the USGS National Water Information System.

Water Data For Texas

Although the development of WOFpy and pyhis has lowered the resource requirements of sharing data, a level of resources is still required. To overcome this, TWDB has formed partnerships with three agencies, The Texas Commission on Environmental Quality (TCEQ), the Texas Parks and Wildlife Department (TPWD) and the Conrad Blucher Institute for Surveying and Science(CBI) to

Fig. 6: Water Data for Texas logo.

serve their data as WaterOneFlow services. This is being done either through scheduled data dumps or using web scapers or on-the-fly web service translations [Pothina2011]. In addition, TWDB is partnering with the Texas Natural Resource Information System (TNRIS) to build a web based map interface that can be used by the general public to find and download water data through a easy to use interface. A high level design schematic of the entire system is presented in Figure 7.

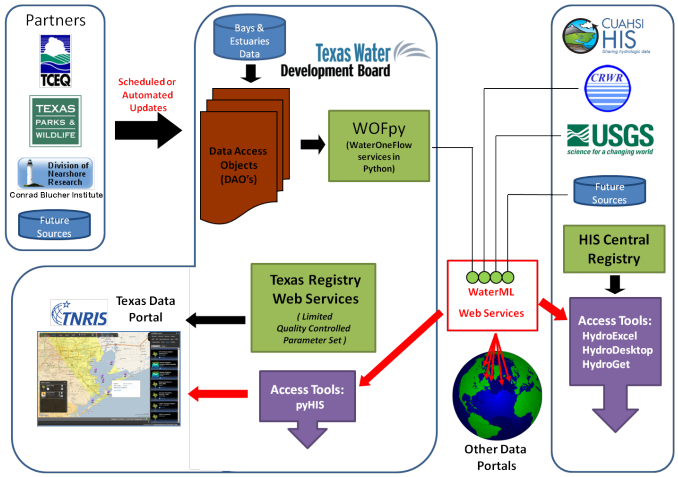

Fig. 7: Water Data for Texas Framework.

This system has been now branded *Water Data for Texas* and will reside at the url http://waterdatafortexas.org once completed.

Conclusions

Water Data for Texas is a community effort to build a robust, sustainable system for the sharing of water data across Federal, State and local entities. Parts of the system are live now with the rest expected to be completed by the Fall of 2011. Currently the system will provide access to all Nation CUAHSI-HIS datasets as well as data from the TCEQ, TPWD, CBI and TWDB. It is expected that new water related data sets will become available as more organizations choose to participate.

Python is an integral part of building this Texas-specific HIS that employs partnerships with Federal and Texas agencies to share water data. The system inherits the national CUAHSI-HIS technology and provides additional tools and services to provide ease of use and a level of quality control for partners and clients.

In order to foster continued development and uptake of the technology in a community supported environment WOFpy and pyhis are being released under a BSD open source license. Development is currently taking place under the swtools organization on GitHub (https://github.com/organizations/swtools).

REFERENCES

[Zaslavsky2007] Zaslavsky, I., D. Valentine and T. Whiteaker, (2007), "CUAHSI WaterML," OGC 07-041r1, Open Geospatial Consortium Discussion Paper, http://portal.opengeospatial.org/files/?artifact_id=21743.

[Goodall2008] Goodall, J. L., J. S. Horsburgh, T. L. Whiteaker, D. R. Maidment and I. Zaslavsky, *A first approach to web services for the National Water Information System*, Environmental Modeling and Software, 23(4): 404-411, doi:10.1016/j.envsoft.2007.01.005.

[Horsburgh2008] Horsburgh, J. S., D. G. Tarboton, D. R. Maidment and I. Zaslavsky, (2008), "A Relational Model for Environmental and Water Resources Data," Water Resour. Res., 44: W05406, doi:10.1029/2007WR006392.

[Ames2009] Ames, D. P., J. Horsburgh, J. Goodall, T. Whiteaker, D. Tarboton and D. Maidment, (2009), *Introducing the Open Source CUAHSI Hydrologic Information System Desktop Application (HIS Desktop)*, 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, ed. R. S. Anderssen, R. D. Braddock and L. T. H. Newham, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation, July 2009, p.4353-4359, http://www.mssanz.org.au/modsim09/J4/ames.pdf.

[Maidment2009] Maidment, D. R., R. P. Hooper, D. G. Tarboton and I. Zaslavsky, (2009), "Accessing and Sharing Data Using CUAHSI Water Data Services," in Hydroinformatics in Hydrology, Hydrogeology and Water Resources, Edited by I. Cluckie, Y. Chen, V. Babovic, L. Konikow, A. Mynett, S. Demuth and D. A. Savic, Proceedings of Symposium JS4 held in Hyderabad, India, September 2009, IAHS Publ. 331, Hyderabad, India, p.213-223, http://iahs.info/redbooks/331.htm.

[Tarboton2009] Tarboton, D. G., J. S. Horsburgh, D. R. Maidment, T. Whiteaker, I. Zaslavsky, M. Piasecki, J. Goodall, D. Valentine and T. Whitenack, (2009) , *Development of a Community Hydrologic Information System*, 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, ed. R. S. Anderssen, R. D. Braddock and L. T. H. Newham, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation, July 2009, p.988-994, http://www.mssanz.org.au/modsim09/C4/tarboton_C4.pdf.

[Ames2010] Ames, D. P., J. Kadlec, and J. Horsburgh, (2010), "HydroDesktop: A Free and Open Source Platform for Hydrologic Data Discovery, Visualization, and Analysis", Francisco Olivera (Editor), 2010 AWRA Spring Specialty Conference: Geographic Information Systems (GIS) and Water Resources VI. American Water Resoruces Association, TPS-10-1, ISBN 1-882132-82-3, http://his.cuahsi.org/documents/conference-awra2010/Ames_abs_13.pdf.

[Whiteaker2010] Whiteaker, T., D. Maidment, D. Pothina, J. Seppi, E. Hersh, and W. Harrison, (2010), "Tesas Hydrologic Information System", Francisco Olivera (Editor), 2010 AWRA Spring Specialty Conference: Geographic Information Systems (GIS) and Water Resources VI. American Water Resoruces Association, TPS-10-1, ISBN 1-882132-82-3, http://his.cuahsi.org/documents/conference-awra2010/DavidMaidment_9eb7f8b0_6581.pdf.

[Pothina2011] Pothina D., A. Wilson *Building a Coastal Geodatabasefor the State of Texas*, Report submitted to the Texas General LandOffice and the Mineral Management Service, Coastal Impact AssistanceProgram Grant Award #M09AF15208, July 2011.

PyModel: Model-based testing in Python

Jonathan Jacky[‡*]

Abstract—In unit testing, the programmer codes the test cases, and also codes assertions that check whether each test case passed. In model-based testing, the programmer codes a "model" that generates as many test cases as desired and also acts as the oracle that checks the cases. Model-based testing is recommended where so many test cases are needed that it is not feasible to code them all by hand. This need arises when testing behaviors that exhibit history-dependence and nondeterminism, so that many variations (data values, interleavings, etc.) should be tested for each scenario (or use case). Examples include communication protocols, web applications, control systems, and user interfaces. PyModel is a model-based testing framework in Python. PyModel supports on-the-fly testing, which can generate indefinitely long nonrepeating tests as the test run executes. PyModel can focus test cases on scenarios of interest by composition, a versatile technique that combines models by synchronizing shared actions and interleaving unshared actions. PyModel can guide test coverage according to programmable strategies coded by the programmer.

Index Terms—testing, model-based testing, automated testing, executable specification, finite state machine, nondeterminism, exploration, offline testing, on-the-fly testing, scenario, composition

Introduction

Model-based testing automatically generates, executes, and checks any desired number of test cases, of any desired length or complexity, given only a fixed amount of programming effort. This contrasts with unit testing, where additional programming effort is needed to code each test case.

Model-based testing is intended to check *behavior*: ongoing activities that may exhibit history-dependence and nondeterminism. The correctness of behavior may depend on its entire history, not just its most recent action. This contrasts with typical unit testing, which checks particular *results*, such as the return value of a function, given some arguments.

It is advisable to check entire behaviors, not just particular results, when testing applications such as communication protocols, web services, embedded control systems, and user interfaces. Many different variations (data values, interleavings etc.) should be tested for each scenario (or use case). This is only feasible with some kind of automated test generation and checking.

Model-based testing is an automated testing technology that uses an executable specification called a *model program* as both the test case generator and the oracle that checks the results of each test case. The developer or test engineer must write a model program for each implementation program or system they wish to test. They must also write a *test harness* to connect the model program to the (generic) test runner.

With model program and test harness in hand, developers or testers can use the tools of the model-based testing framework in various activities: Before generating tests from a model, it is helpful to use an *analyzer* to validate the model program, visualize its behaviors, and (optionally) perform safety and liveness analyses. An *offline test generator* generates test cases and expected test results from the model program, which can later be executed and checked by a *test runner* connected to the implementation through the test harness. This is a similar workflow to unit testing, except the test cases and expected results are generated automatically. In contrast, *on-the-fly testing* is quite different: the test runner generates the test case from the model as the test run is executing. On-the-fly testing can execute indefinitely long nonrepeating test runs, and can accommodate nondeterminism in the implementation or its environment.

To focus automated test generation on scenarios of interest, it is possible to code an optional *scenario machine*, a lightweight model that describes a particular scenario. The tools can combine this with the comprehensive *contract model program* using an operation called *composition*. It is also possible to code an optional *strategy* in order to improve test coverage according to some chosen measure. Some useful strategies are already provided.

Model-based testing supports close integration of design and analysis with testing. The analyzer is similar to a model checker; it can can check safety, liveness, and temporal properties. And, the *same models* are used for these analyses as for automated testing. Moreover, the models are written in the *same language* as the implementation.

PyModel is an open-source model-based testing framework for Python [PyModel11]. It provides the PyModel Analyzer `pma`, the PyModel Graphics program `pmg` for visualizing the analyzer output, and the PyModel Tester `pmt` for generating, executing, and checking tests, both offline and on-the-fly. It also includes several demonstration samples, each including a contract model program, scenario machines, and a test harness.

The PyModel framework is written in Python. The models and scenarios must be written in Python. It is often convenient, but not required, if the system under test is also written in Python, because it can be easier to write the test harness in that case.

Traces and Actions

We need to describe behavior. To show how, we discuss the Alternating Bit Protocol [ABP11], a simple example that exhibits history-dependence and nondeterminism. The protocol is designed to send messages over an unreliable network. The sender keeps

* *Corresponding author: jon@uw.edu*
‡ *University of Washington*

sending the same message, labeled with the same bit (1 or 0), until the receiver acknowledges successful receipt by sending back the same bit. The sender then complements the bit and sends a new message labeled with the new bit until it receives an acknowledgement with that new bit, and so on. When the connection starts up, both ends send bit 1. The sender labels the first real message with 0.

A sample of behavior is called a *trace*. A trace is a sequence of *actions*, where each action has a name and may have arguments (so actions resemble function calls). The alternating bit protocol has only two actions, named `Send` and `Ack`. Each action has one argument that can take on only two values, `0` or `1`. (We abstract away the message contents, which do not affect the protocol behavior.) Here are some traces that are allowed by the protocol, and others that are forbidden:

```
Allowed     Allowed     Allowed     Forbidden   Forbidden
-------     -------     -------     ---------   ---------
Send(0)     Send(1)     Send(1)     Send(0)     Send(0)
Ack(0)      Send(1)     Send(1)     Ack(0)      Ack(1)
Send(1)     Ack(1)      Ack(1)      Send(0)     Send(1)
Ack(1)      Send(0)     Send(1)     Ack(0)      Ack(1)
            Ack(1)      Ack(1)
            Ack(1)      Send(1)
            Send(0)
            Ack(0)
```

Traces like these might be collected by a test harness connected to the sender. The `Send` are *controllable actions* invoked by the sender while the `Ack` are *observable actions* that are observed by monitoring the network. (If the test harness were connected to the receiver instead, the `Send` would be the observable action and the `Ack` would be controllable.)

Finite Models

A model is an executable specification that can generate traces (to use as test cases) or check traces (to act as an oracle). To act as a specification, the model must be able to generate (or accept) any allowed trace and must not be able to generate any forbidden trace (it must reject any forbidden trace).

The alternating bit protocol is *finite* because there are only a finite number of actions (only a finite number of possible values for each action argument). Therefore this protocol can be modeled by a *finite state machine* (FSM), which can be represented by a graph where the edges represent actions and the nodes represent states (Figure 1). Every allowed trace can be obtained by traversing paths around this graph. In the figure, some of the nodes have doubled borders. These are the *accepting states* where traces are allowed to stop. A trace that stops in a non-accepting state is forbidden. If no accepting states are specified, all states are considered accepting states.

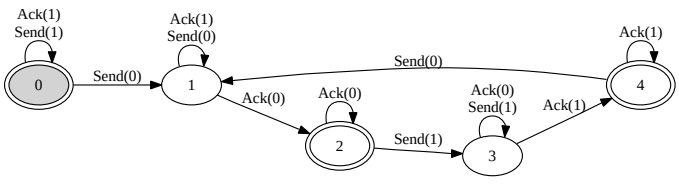

Fig. 1: *Alternating bit protocol represented by a finite state machine (FSM)*

In PyModel, a finite state machine is represented by its graph: a tuple of tuples, where each tuple represents a state transition, the current state (a node), the action (an edge), and the next state (another node):

```
graph = ((0, (Send, (1,),), 0),
         (0, (Ack, (1,),), 0),
         (0, (Send, (0,),), 1),
         (1, (Ack, (0,),), 2),
         ... etc. ...
         (4, (Send, (0,),), 1))
```

The PyModel Graphics program `pmg` generated Figure 1 from this code.

Most interesting systems are infinite and cannot be described by finite state machines. In PyModel, finite state machines are most often used to describe *scenario machines* that are composed with infinite *contract model programs* to focus test case generation on scenarios of interest.

Infinite Models

Most interesting systems require infinite models. A system requires an infinite model when it has an infinite number of actions. This occurs whenever any of its action arguments are drawn from types that have an infinite number of values: numbers, strings, or compound types such as tuples.

Simple systems can be infinite. Consider a stack, a last-in first-out queue which provides a `Push` action that puts a value on top of the stack and a `Pop` action that removes the value from the top of the stack and returns it. Here are some allowed traces:

```
Push(1,)      Push(1,)      Push(1,)
Push(2,)      Pop(), 1      Push(2,)
Push(2,)      Push(2,)      Push(2,)
Push(1,)      Pop(), 2      Push(1,)
Pop(), 1      Push(1,)      Push(1,)
Pop(), 2      Pop(), 1      Push(1,)
Pop(), 2      Push(2,)      Push(2,)
Push(2,)      Pop(), 2      Push(2,)
Push(1.)      Push(1,)      Push(1,)
Push(1,)      Pop(), 1      Push(1,)
```

In PyModel, an infinite model is expressed by a Python module with an *action function* for each action and variables to represent the *state*, the information stored in the system. In this example, the state is a list that stores the stack contents in order. Constraints on the ordering of actions are expressed by providing each action with an optional *guard* or *enabling condition*: a Boolean function that is true for all combinations of arguments and state variables where the action is allowed to occur. In this example, `Push` is always enabled so no enabling function is needed; `Pop` is only enabled in states where the stack is not empty. Here is the model, as coded in the module `Stack`:

```
stack = list()       # State

def Push(x):         # Push is always enabled
  global stack
  stack.insert(0,x)

def Pop():           # Pop requires an enabling condition
  global stack
  result = stack[0]
  del stack[0]
  return result

def PopEnabled():    # Pop enabled when stack not empty
  return stack
```

Analysis

It can be helpful to visualize the behavior of a model program. The PyModel Graphics program `pmg` can generate a graph from

finite state machine, as in Figure 1. The PyModel Analyzer `pma` generates a finite state machine from an infinite model program, by a process called *exploration* which is a kind of concrete state model-checking. In order to finitize the model program, it is necessary to limit the action arguments to finite *domains* and it may also be necessary to limit the state by *state filters*, Boolean functions which the state must satisfy. Exploration in effect performs exhaustive testing of the model program over these finite domains, generating all possible traces and representing them compactly as an FSM.

Here we define a domain that limits the arguments of `Push` to the domain `0, 1`; we also define a state filter that limits the stack to fewer than four elements:

```
domains = { Push: {'x':[0,1]} }

def StateFilter():
  return len(stack) < 4
```

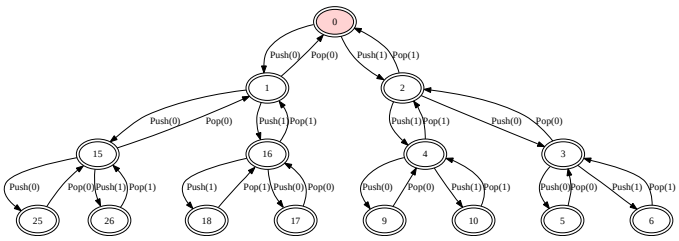

Fig. 2: *FSM for finitized Stack model program, generated by exploration.*

Subject to these limitations, `pma` generates a finite state machine that is rendered by `pmg` (Figure 2).

Every trace allowed by the (finitized) model can be obtained by traversing paths around the graph. This is useful for validation: you can check whether the graph allows the expected behaviors.

Safety and Liveness

In addition to providing visualization, the analyzer can check other properties. *Safety analysis* checks whether anything bad can happen. You specify safety requirements by defining a *state invariant*, a Boolean function on state variables that is supposed to be satisfied in every state. The analyzer checks the invariant in every state reached during exploration and marks *unsafe states* where the invariant is violated. *Liveness analysis* checks whether something good will happen. You specify liveness requirements by defining an *accepting state condition*, a Boolean function on state variables that is supposed to be satisfied in the states where a trace ends. The analyzer checks the accepting state condition in every state and marks the terminal states (which have no outgoing actions) where the condition is violated; these are *dead states* from which an accepting state cannot be reached. Since exploration is exhaustive, these analyses are conclusive; they are machine-generated proofs that the safety and liveness properties hold (or not) for the model program over the given finite domains.

Test Harness

In order to execute tests, it is necessary to write a *test harness* that connects the model program to the test runner `pmt`. The test harness usually encapsulates the implementation details that are abstracted away from the model. It is often convenient, but not required, if the implementation under test is also written in Python, because it can be easier to write the test harness in that case.

Here is a fragment of the code from the harness for testing a web application. As it happens, the server code of the web application that we are testing here is in PHP, not Python, but this is not an inconvenience because the test harness acts as a remote web client, using the Python standard library module `urllib`, among others. The model includes `Initialize`, `Login`, and `Logout` actions, among others:

```
def TestAction(aname, args, modelResult):
  ...

  if aname == 'Initialize':
    session = dict() # clear out cookies/session IDs

  elif aname == 'Login':
    user = users[args[0]]
    ...
    password = passwords[user] if args[1] == 'Correct'
                               else wrongPassword
    postArgs = urllib.urlencode({'username':user,
                                 'password':password})
    # GET login page
    page = session[user].opener.open(webAppUrl).read()
    ...
    if result != modelResult:
      return 'received Login %s, expected %s' % \
                 (result, modelResult)

  elif aname == 'Logout':
    ...
```

Offline Testing

Offline testing uses a similar workflow to unit testing, except the test cases and expected results are generated automatically from the model program.

Traces can be used as test cases. The PyModel Tester `pmt` can generate traces from a (finitized) model program; these include the expected return values from function calls, so they contain all the information needed for testing. Later, `pmt` can act as the test runner: it executes the generated tests (via the test harness) and checks that the return values from the implementation match the ones in the trace calculated by the model program.

On-the-fly Testing

In *On-the-fly testing* the test runner `pmt` generates the test case from the model as the test run is executing. On-the-fly testing can execute indefinitely long nonrepeating test runs. On-the-fly testing is necessary to accommodate nondeterminism in the implementation or its environment.

Accommodating nondeterminism requires distinguishing between *controllable actions* (functions that the test runner can call via the test harness), and *observable actions* (events that the test harness can detect). For example, when testing the sender side of the alternating bit protocol, `Send` is controllable and `Ack` is observable. Handling observable actions may require asynchronous programming techniques in the test harness.

Strategies

During test generation, alternatives arise in every state where multiple actions are enabled (that is, where there are multiple outgoing edges in the graph of the FSM). Only one action can be chosen. The algorithm for choosing the action is called a *strategy*. In PyModel, the default strategy is random choice among the

enabled actions. It is also possible to code an optional *strategy* in order to improve test coverage according to some chosen measure.

Some useful strategies are already provided. The `ActionNameCoverage` strategy chooses different actions, while the `StateCoverage` strategy attempts to reach unvisited states. Here are some test cases generated from the stack model using different strategies:

```
Random          Action name     State
(default)       coverage        coverage
--------        --------        --------
Push(1,)        Push(1,)        Push(1,)
Push(2,)        Pop(), 1        Push(2,)
Push(2,)        Push(2,)        Push(2,)
Push(1,)        Pop(), 2        Push(1,)
Pop(), 1        Push(1,)        Push(1,)
Pop(), 2        Pop(), 1        Push(1,)
Pop(), 2        Push(2,)        Push(2,)
Push(2,)        Pop(), 2        Push(2,)
Push(1.)        Push(1,)        Push(1,)
Push(1,)        Pop(), 1        Push(1,)
```

Composition

Composition is a versatile technique that combines models. PyModel uses it for scenario control, validation, and program structuring. All of the PyModel commands can accept a list of models to be composed in any context where they expect a model.

Composition combines two or more models to form a new model, the *product*. (In the following discussion and examples, just two models are composed.)

$$M_1 \times M_2 = P$$

When the product is explored, or is used to generate or check traces, PyModel in effect executes the composed models in parallel, synchronizing on shared actions and interleaving unshared actions. A shared action occurs in both models, an unshared action occurs in only one. A shared action must be simultaneously enabled in both models in order to execute in the product. This results in synchronizing the execution of the shared actions. This usually has the effect of limiting or restricting behavior, in effect filtering it (Figure 3). This is useful for both scenario control and validation, as we shall see.

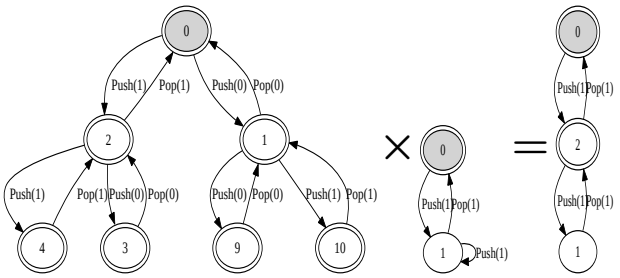

Fig. 3: *Composition synchronizes on shared actions.*

An unshared action can execute in the product whenever it is enabled in its own model. This results in interleaving the execution of the unshared actions in the product. This usually has the effect of enlarging the behavior, in effect multiplying it (Figure 4). This can be useful as a structuring technique for building up complex models from simpler ones.

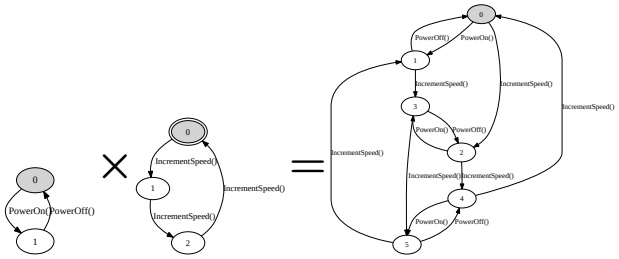

Fig. 4: *Composition interleaves unshared actions.*

Notice that a state is an accepting state in the product if and only if it is an accepting state in both of the composed models.

Scenario Control

A difficulty with any automated testing method is generating too many tests. We need *scenario control* to limit test runs to scenarios of interest. We can achieve this by composing the comprehensive *contract model program*, usually a Python module with state variables etc., with a particular *scenario machine*, usually an FSM.

$$Contract \times Scenario = Product$$

In this example (Figure 5), the contract model program (on the far left) allows many redundant, uninteresting startup and shutdown paths. We would like to intensively test just the few interesting actions in this model. We create a scenario machine (on the near left) that specifies a single path through startup and shutdown, and omits the interesting actions. When we compose the two models, the startup and shutdown actions are shared so the two models must synchronize, which forces the product to follow the sequences in the scenario. The interesting actions are unshared, so they are free to interleave, and the product can execute these as long as they are enabled. The product (on the right) will only generate traces that are interesting for this test purpose.

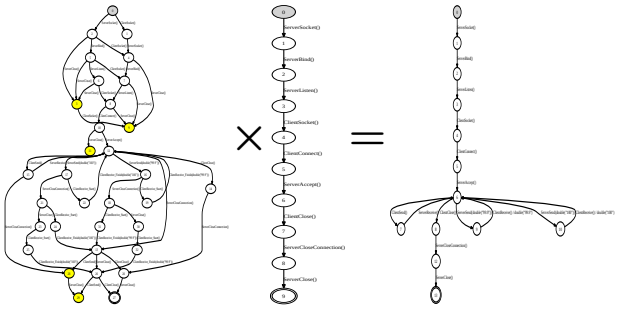

Fig. 5: *Composition with a scenario can eliminate uninteresting paths from tests.*

Validation

A model program is just a program so it is necessary to *validate* it: to confirm that it expresses the intended behaviors. As already noted, simply inspecting the graphs generated by the analyzer can be helpful for this.

Composition also supports a more rigorous validation procedure analogous to unit testing. Composing a contract model program with a scenario machine results in a product that reaches an accepting state if and only if the model allows the behaviors described by the scenario, that is, if the model can execute the scenario. If the model cannot execute the scenario, the product will not reach an accepting state. Therefore, a collection of scenarios that are each known *a priori* to be allowed or forbidden can act as a unit test suite for a model program. Composing the model with each scenario in turn is, in effect, executing the unit test suite.

Figures 3 and 5 both show examples where the model program can execute the scenario. In Figure 6 we compose the stack model with a scenario that executes `Push(1)` followed by `Pop(),0`. This is forbidden, because pop should only return the value that was most recently pushed. As expected, we see that the product only contains the push action because it is unable to synchronize on the pop action, which is not enabled in the model. The product does not reach an accepting state, which shows that the model does not allow this scenario.

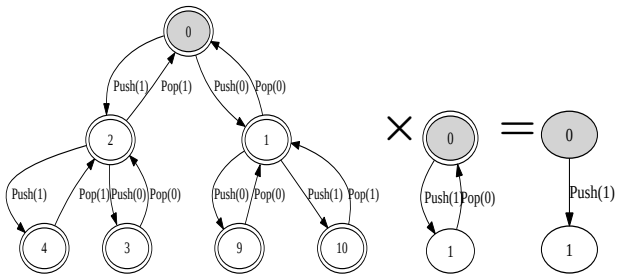

Fig. 6: *Composition with a forbidden scenario cannot reach an accepting state.*

This technique can be use to check a model program for any property that can be expressed by a finite state machine, including any temporal logic formula. Exploration with composition is similar to model checking, and is a powerful complement to the state-based safety and liveness analyses described earlier.

Conclusions

Model-based testing can encourage different approaches to testing. It encourages on-the-fly testing --- but in general, on-the-fly test runs are not reproducible, due to nondeterminism. It suggests extending testing to noninvasive monitoring or *run time verification* --- if the test harness supports observable actions, the test runner can check log files or monitor network traffic for conformance violations.

The most intruiging prospect might be better integration of design and analysis with testing. Exploration with composition is like model checking; it can can check for safety, liveness, and temporal properties. And, the *same models* are used for these analyses as for automated testing. Moreover, the models are written in the *same language* as the implementation, which could make them accessible to developers and test engineers, not just formal methods experts.

Model-based testing has been used on large projects in industry, but only *post-hoc*. Test engineers were given informal documentation and an implementation to test, and then reverse-engineered the models [Grieskamp08]. A more rational workflow might be to write the model *before* writing the implementation, analyze and tweak the design, then implement and test.

Related work

The techniques described in this paper can be expressed in any programming language. More detailed explanations and examples, using the NModel framework for C# [NModel11], appear in [Jacky08]. Another view of model-based testing appears in [Utting07]. Model checking is discussed in [Peled01].

References

[ABP11] Alternating Bit Protocol, Wikpedia, accessed June 2011. http://en.wikipedia.org/wiki/Alternating_bit_protocol

[Grieskamp08] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F.L. Wurden. Model-based quality assurance of Windows protocol documentation. In: *ICST*, pages 502-506. IEEE Computer Society, 2008.

[Jacky08] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*, Cambridge University Press, 2008.

[NModel11] NModel software, accessed June 2011. http://nmodel.codeplex.com/

[Peled01] Doron Peled. *Software Reliability Methods*, Springer, 2001.

[PyModel11] PyModel software, accessed June 2011. http://staff.washington.edu/jon/pymodel/www/

[Utting07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: a Tools Approach*, Morgan-Kaufmann, 2007.

Automation of Inertial Fusion Target Design with Python

Matthew Terry‡*, Joseph Koning‡

Abstract—The process of tuning an inertial confinement fusion pulse shape to a specific target design is highly iterative process. When done manually, each iteration has large latency and is consequently time consuming. We have developed several techniques that can be used to automate much of the pulse tuning process and significantly accelerate the tuning process by removing the human induced latency. The automated data analysis techniques require specialized diagnostics to run within the simulation. To facilitate these techniques, we have embedded a loosely coupled Python interpreter within a pre-existing radiation-hydrodynamics code, Hydra. To automate the tuning process we use numerical optimization techniques and construct objective functions to identify tuned parameters.

Index Terms—inertial confinement fusion, python, automation

Inertial Confinement Fusion

Inertial confinement fusion (ICF) is a means to achieve controlled thermonuclear fusion by way of compressing hydrogen to extremely large pressures (GBar), temperatures (10's keV) and densities (100x solid density). ICF capsule are typically small (~1 mm radius) spheres composed of several layers of cryogenic hydrogen, plastic, metal or other materials. These extreme conditions are reached by illuminating the capsule with a very high intensity (100's TW) driver. This compresses the shell to more than 100 times solid density and accelerates the radially converging shell to very high velocity (300 km/s). As the shell stagnates, a fusion burn wave propagates from a central, low-density, high temperature region to a surrounding high-density, low temperature fuel region. The inertia of the fuel keeps it intact long enough for a significant fraction of the fuel to burn.

There are several approaches to achieving a significant fusion burn, but for this paper consider the shock ignition [Betti2007] approach with the capsule directly driven by lasers. The capsule is a spherical shell of frozen deuterium-tritium ("DT ice"), coated with plastic or another ablator material. The region within the DT ice is filled with DT gas at the vapor pressure. Laser beams directly illuminate the target and deposit energy in the outer most layer called the ablator. The ablation of the ablator supplies the pressure to drive the implosion. We assume a spherically symmetric illumination of the capsule with the total incident power varying in time. The power vs time profile is referred to as the "pulse shape."

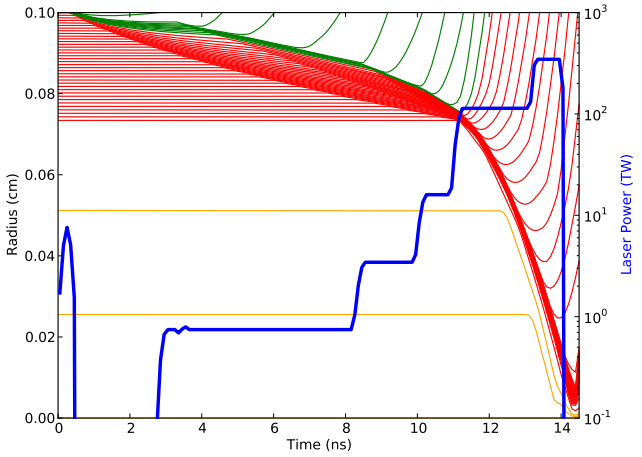

Fig. 1: *A Radius-Time plot of the capsule implosion with the incident laser power overlay. Lines plot the trajectory of fluid particle boundaries. Lines are color coded by material.*

We divide the pulse shape into three logical sections, which correspond to the three phases of the capsule implosion dynamics. The first section is called the "pre-pulse" and is responsible for shock compressing the DT shell to high density. The pre-pulse consists of a short duration, high intensity spike in the laser power (the "picket") and three pedestals, each with increasing laser power. The pre-pulse is followed by the main pulse, which accelerates the shell to moderate implosion velocity (~300 km/s). When the imploding shell stagnates, it forms a central, low density, high temperature hot spot and a surrounding high density, low temperature shell.

The final section of the pulse shape is the igniter pulse. The igniter pulse consists of another pedestal of very high intensity. This section launches a strong shock that arrives just as the shell is stagnating and further heats the hot spot as well as prevents the low pressure shell from coming into pressure equilibrium with the high pressure hot spot. The combination of the stagnation of the shell and the timely arrival of the igniter shock lifts the temperature of the central hot spot above the 12 keV threshold needed to initiate a fusion burn wave. This burn wave propagates into the cold shell where it produces most of the fusion yield.

While restricting our attention to laser shock shock ignition,

* *Corresponding author: terry10@llnl.gov*
‡ *Lawrence Livermore National Laboratory*

there is a lot of potential variability in the composition and structure of capsules and in the pulse shape. Capsule should have sufficient ablator to drive the implosion, but not in excess. Capsule materials must anticipate the effect of fluid instabilities and laser absorption. The capsule should have realistic fabrication tolerances. Laser powers must be set to produce shocks of an appropriate strength and pulse features should be appropriately timed. Additionally, there are several physical processes important in describing an implosion. Due to all of these sources of complexity, ICF targets are designed using sophisticated multi-physics codes, such as Hydra [Marinak1996]. Extensive simulation, helps identify interesting capsule/pulse shapes before resorting to expensive and difficult experiments. The process of designing a capsule is highly iterative, time consuming, interactive process. In this paper we describe the use of and modifications of Hydra to automate significant sections of the target design process. Specifically, we consider the situation where a capsule design and the pulse shape power levels are specified and the timing of the pulse shape is not specified.

When tuning the pre-pulse, we regard the picket as a fixed quantity functioning as a time fiducial for synchronizing the remaining pre-pulse shocks. The picket exists to increase implosion stability by heating the plasma corona, which increase lateral thermal conduction, which in turn smooths out non-uniformities in the deposition of laser energy. These are not effects that can be resolved in one dimensional (1D) simulations, but the picket effects the 1D dynamics and must be included. The pre-pulse pedestals should have their start times set such that their associated shocks reach the gas/ice interface within 50 ps as the picket shock [Munro2001]. Spacing the pre-pulse shocks in this way, prevents them from coalescing in the ice and unnecessarily shock heating the fuel. Also, shocks gain strength with radial convergence, so ensuring that the pre-pulse shocks escape the fuel while it is at large radius helps minimize the shock heating.

The main pulse should be timed to get the maximum fuel confinement for a fixed amount of energy. The appropriate measure of fuel confinement is its peak areal density ($\rho R = \int \rho(r)dr$). Should the fuel ignite, the burn fraction is approximated by $f \approx \frac{\rho R}{\rho R+7}$ [Fraley1974]. Finally, the igniter pulse should be timed so that the target ignites robustly. We implement this as maximizing the fusion yield.

Automatic Tuning

We adopt the general strategy that a tuned pulse can be constructed by serially adding tuned pulse segments. Additionally we require that each property of a pulse segment can be "tuned" by numerically optimizing an appropriately chosen objective function. Our automated pulse tuner ("autotuner") is structured around an iteration over pairs of pulse properties and objective functions. These properties are the start times of the pulse segments and are initially turned off. The tuner iterates through each pulse segment, numerically optimizing it based on its associated objective function. In addition to fixed power levels, the combined energy delivered in the pre-pulse and compression pulse is constrained. The total igniter pulse energy is also pre-determined. It is important to realize that the sequence of properties and choice of objective functions embodies a strategy to achieve the desired target behavior. The automation of this strategy does not guarantee the tuned pulse will produce the desired performance characteristics, just that the design strategy was faithfully executed.

In addition to a sequence of parameters and a definition of an objective function, an autotuing program requires other software infrastructure. It needs to transform parameter values to input files and run directories. The autotuing program needs to gather the appropriate information from a simulation needed by the objective functions. Finally, it needs reasonably efficient numeric optimization routines.

We generate Hydra input files from a Python proxy class that wraps a nearly complete Hydra input file. The proxy has simple pre-processor like capabilities for modifying simple input file statements and for injecting more complicated structures into the input file. For complicated structures, like the laser source specification, it delegates responsibly to special purpose objects. These object follow the convention that `str(obj)` produces a string formatted for inclusion in a Hydra input file. This convention allows objects that define the `__str__()` to lazily evaluate their Hydra representation, while actual strings can be inserted with no boilerplate.

Certain objective functions require very high sampling rates and thus must be run within a running simulation. For this purpose, Hydra has an embedded Python interpreter. Since our tuning program and Hydra's embedded interpreter use the same programming language, it is relatively easy for the control program and Hydra to share data structures. There are two obvious methods: object serialization with the pickle module and object reconstruction using `repr()`. Reconstructed objects are easily modified and more explicit, so we use that method.

All of the optimizations use a simple eight way parallel direct search method. In terms of the number of function evaluations, direct search is less efficient than Newton-like methods, direct search is very inefficient. Typical optimizations requires 32 functions evaluations. Converging to the same tolerance using the BFGS method requires only 12 function evaluations. However, the inefficient direct search method requires only 4 iterations, compared to the 12 iterations with BFGS. We are satisfied with the current performance, but recognize that the use of more sophisticated sampling techniques would likely reduce the number of iterations or the number of parallel function evaluations.

Hydra's Parallel Python Interpreters

Hydra is a massively parallel multi-physics code in use since 1993. The code combines hydrodynamics with radiation diffusion, laser ray trace, and several more packages necessary for ICF design and has over 40 users at national laboratories and universities.

Hydra users set up simulations using a built-in interpreter. The existing interpreter provides access to the program parameters and provides functions to access and manipulate the data in parallel. Users can also access and alter the state while the simulation is running through a message interface that runs at a specific cycle, time or if a specific condition is met.

To improve functionality, the Python interpreter was added to Hydra. Python was chosen due to the mature set of embedding API and extending tools and the large number of third party libraries. The Python interpreter was added by embedding instead of extending Python itself. This choice was made due to the large number of existing input files that could not be easily ported to a new syntax. The Simplified Wrapper and Interface Generator (SWIG) [Beazley2003] interface generator is used to wrap the Hydra C++ classes and C functions.

Users can send commands to the Python interpreter using two separate methods: a custom interactive interpreter based on the

CPython interpreter and a file-based Python code block interpreter. The Hydra code base is based on the message passing interface (MPI) library. This MPI library allows for efficient communication of data between processors in a simulation. The embedded interactive and file based methods must have access to the Python input source on all of the processors. The MPI library is used to broadcast a line read from stdin or a file on the root processor to all of the other processors in the simulation. The simplest method to provide an interactive parallel Python interpreter would be to override the `PyOS_ReadlineFunctionPointer` in the Python code base. This function cannot be overridden for non-interactive processes due to a check for an interactive tty. An alternative interactive Python interpreter was developed to handle the parallel stdin access and Python code execution. For parallel file access the code reads the entire file in as a string and broadcasts it to all of the other processors. The string is then sent through the embedded Python interpreter function `PyRun_SimpleString`. This C function will take a char pointer as the input and run the string through the same parsing and interpreter calls as a file using the Python program. One limitation of the `PyRun_SimpleString` call is the lack of exception information. To alleviate this issue a second method was implemented uses `Py_CompileString` then `PyEval_EvalCode`. The `Py_CompileString` uses a file name or input file information to give a better location for the exception.

The existing Hydra interpreter is the dominant interpreter and must be given control when Python is not in use. The interactive Python interpreter must check for Hydra control commands as well as compiling, executing and checking errors on Python code. The custom interactive interpreter first reads a line from stdin in parallel. Readline support is enabled which gives the user line editing and history support similar to running the Python program interactively. The line is then checked for any Hydra specific control sequences and compiled through the `Py_CompileStringFlags`. If the line compiled with no errors then it is executed using the `PyEval_EvalCode` command. Any errors in compiling or exceptions are checked for a block continuation indicator, syntax error or EOF. Exceptions will be displayed as in Python and available in the output of all the processors.

With the above embedded Python support users can run arbitrary Python code through the Python interpreter. One of the mandates of the effort to embed the Python interpreter was to provide an enhanced version of the existing Hydra interpreter. In order to provide this functionality Python must be able to access the information in the running Hydra simulation. This is accomplished by wrapping the Hydra data structures, functions, and parameters using SWIG and exposing them through the `hydra` Python extension module. The code created by SWIG includes a C++ file compiled into Hydra as a Python extension library and a Python interface file that is serialized and compiled into the Hydra code.

The `hydra` Python module allows users to access and manipulate the Hydra state. Hydra has several types of integer and floating point arrays ranging from one to three dimensional. The multi dimensional arrays have an additional index to indicate the block in the block-structured mesh. The block defines a portion of the mesh on which the zonal, nodal, edge, and face based information is defined, these meshes can consist of several blocks. The blocks are then decomposed into sub-blocks or domains depending on how many processors will be used in the simulation. Access to the multi-block parallel data structures is provided by structures wrapped by C++ interface objects and then wrapped in SWIG using the numerical Python, `numpy`, module to provide the array object in Python.

Users control the simulation by scheduling messages that conditionally execute based on cycle number, time or specific states. These messages can be redefined from Python to steer the simulation while it is running. In addition to the messages, there is a callback functionality that will run a user defined Python function after every simulation cycle has completed. An arbitrary number of callable Python objects can be registered in the code.

Objects in the top level, `__main__`, state are saved to a restart file. This restart file is a portable file object written through the mesh and file I/O library silo [SILO2011]. The Python component of the restart information is a binary string created through the pickle interface augmented with a state saving module. The Python module used for the state saving functionality is the `savestate` module by Oren Tirosh [Tirosh2008]. This module has been augmented with the addition of `numpy` support and None and Ellipsis singleton object support.

Multiple versions of the Hydra code are available to users at any given time. In order to add additional functionality and maintain version integrity, the `hydra` Python module is embedded in the Hydra code as a frozen module. The Python file resulting from the SWIG generator is marshaled using a script based on the freeze module in the Python distribution. This guarantees the modules are always available even if the `sys.path` is altered.

Embedded Diagnostics and Objective Functions

Embedding a Python interpreter within Hydra adds significant capability. One of the first applications was to add a fluid characteristic tracker. Characteristics are eigenvectors of the Euler fluid equations and represent the highest possible signal speed. Characteristics located near a shock, the characteristic will naturally drift toward the shock front or be swept up in int, consequently they can be used to identify the location of the shock front without the difficulty of post processing the moving Lagrangian mesh. The following initial value problem describes the radial location of the characteristic as the flow evolves: $\dot{r} = v(r) - c_s(r)$. $u(r)$ and $c_s(r)$ are the flow velocity and sound speed at the characteristic's current location r. Our characteristic tracker implementation is aware of the pulse shape and starts tracking a new characteristic for each significant feature of the pulse shape. Characteristic positions must be updated every cycle and the tracker is registered as a callback.

Since the tracker is updated every cycle, it is easy to trigger other events based on the behavior of the characteristic. The first use is trigger the simulation to end just after shock breakout time. This is very important as Hydra's only other relevant mechanism for ending the simulation is a maximum simulation time. Burn is explicitly turned off for these scans, so Hydra's burn rate monitor is not relevant. Setting a time limit either leads to under-estimating the shock breakout time and stopping the calculation before gathering important information or setting the maximum time to be very large and wasting many compute cycles. Additionally, we use the location of characteristics to set the frequency Hydra writes output files. Different stages of the simulation have disparate time scales and it is useful to add resolution only when it is needed.

The most important application of the characteristic tracker is producing smooth, non-noisy measurements of the shock breakout time for the shock syncing objective function. To construct a

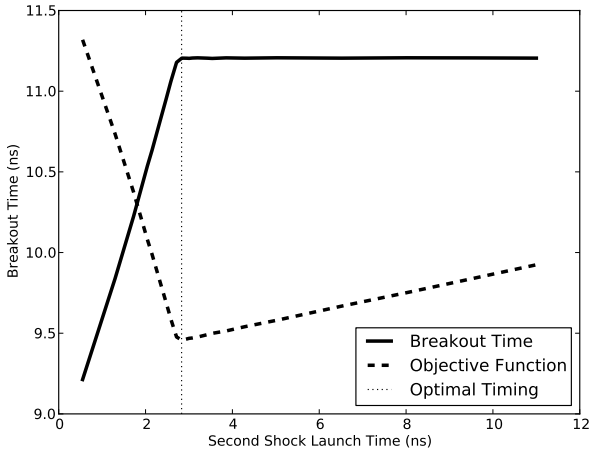

Fig. 2: *Breakout time for a scan of the start time of the second shock. Notice that the objective function minimum accurately locates the inflection point in the breakout vs start time plot.*

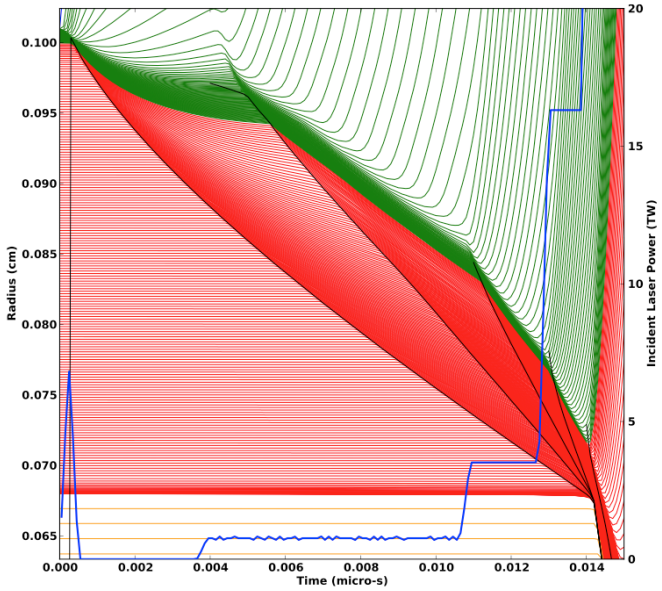

Fig. 3: *R-T plot showing optimal timing of pre-pulse shocks. Shock fronts are identified with black lines.*

shock syncing objective function, first consider the case of two radially converging shocks launched at two different times from comparable radii. The second shock is faster since the wake of the first is warmer and the sound speed is larger. The second shock will eventually overtake the first. If we define a "shock breakout time" as when the first shock enters the gas region, we can plot the shock breakout time as a function of the launch time of the second shock (black line in 2). The appropriate objective function should maximize the breakout time (recognizing that it saturates for large launch times) while also minimizing the launch time of the second shock. We construct an aggregate objective function as a linear combination of the two constraints ($f(t) = \omega t - b(t)$). We find an tuned value of $0.01m$, where m is the slope between the end points of the search region. The parallel direct search optimization method typically converges within four iterations.

Recall from the first section the pre-pulse launches four

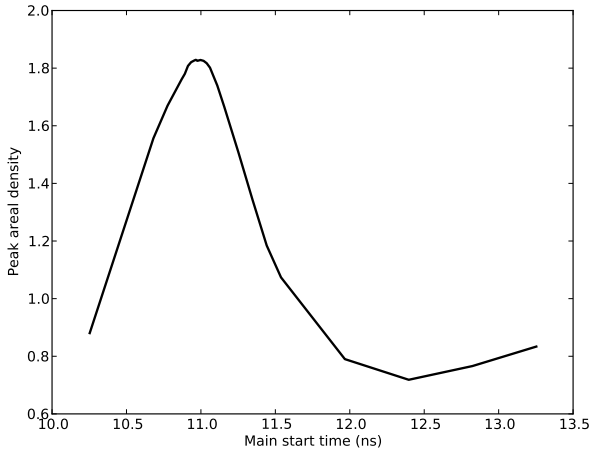

Fig. 4: *Tuning peak areal density*

shocks, all of which should coalesce at the gas-ice interface at the same time. Figure 3 shows the convergence of the pre-pulse shocks well within the required 50 ps tolerance. It should be noted that this shock syncing method only relies on tracking the first shock. Characteristics will sometimes fail to locate the shock if they are located in a region with heat sources that are not sonically coupled to the plasma. Deeply penetrating x-rays, supra-thermal electrons and heavy ion beams are examples. However, it is expected that the ablator and the DT shell should provide sufficient insulation for the picket shock tracker to locate its shock.

Another important embedded diagnostic monitors the fuel areal density (ρR). When tuning the main pulse, the diagnostic monitors the DT ρR, reports the peak value and stops the calculation when the current ρR has fallen to 50% of the peak value. The maximum ρR sets the start time of the main pulse. The igniter pulse start time is tuned by maximizing the fusion yield. Figure 4 shows a peak ρR of $1.8 \mathrm{g/cm}^2$ with a time width of 500ps. Peak ρR is typically found within three iterations. The width in the peak corresponds to mistiming robustness.

Hydra is already well suited for tuning the igniter pulse for maximum fusion yield and needs no additional diagnostics. Hydra monitors the burn rate and has triggers to end the calculation upon completion of burn. Hydra also reports the total fusion yield.

Conclusions

Tuning an ICF pulse to a target is normally a labor intensive, high latency process. We described the desired properties of a tuned pulse and constructed objective functions that will identify the tuned properties. Collecting information for the objective functions requires high frequency sampling of simulation and this data must be gathered within the simulation rather than post-processing a completed simulation. To enable introspective simulations, we add a parallel Python interpreter to Hydra. From these pieces, we constructed a program that tunes a pulse without human intervention. The net result is a significant time savings over manual tuning. Where a typical manual tuning takes several days of attention, an automated tuning takes around 4 hours to execute the same number of simulations.

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

REFERENCES

[Tirosh2008] O. Tirosh, *Pickle the interactive interpreter state (Python recipe)*, http://code.activestate.com/recipes/572213-pickle-the-interactive-interpreter-state/, 2008.

[SILO2011] https://wci.llnl.gov/codes/silo/.

[Betti2007] Betti, R, et al. 2007. Shock Ignition of Thermonuclear Fuel with High Areal Density. *Phys. Rev. Lett.* 98, 155001.

[Munro2001] Munro, David H, et al. 2001. Shock timing technique for the National Ignition Facility. *The 42nd annual meeting of the division of plasma physics of the American Physical Society and the 10th international congress on plasma physics* 8, 2245-2250.

[Fraley1974] Fraley, G S, et al. 1974. Thermonuclear burn characteristics of compressed deuterium-tritium microspheres. *Physics of Fluids* 17, 474-489.

[Marinak1996] Marinak, M M, et al. 1996. Three-dimensional simulations of Nova high growth factor capsule implosion experiments. *Physics of Plasmas* 3, 2070-2076.

[Beazley2003] Beazley,. 2003. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.* 19, 599--609.

Hurricane Prediction with Python

Minwoo Lee[§*], Charles W. Anderson[§], Mark DeMaria[‡]

Abstract—The National Centers for Environmental Prediction (NCEP) Global Forecast System (GFS) is a global spectral model used for aviation weather forecast. It produces forecasts of wind speed and direction, temperature, humidity and precipitation out to 192 hr every 6 hours over the entire globe. The horizontal resolution in operational version of the GFS is about 25 km. Much longer integration of similar global models are run for climate applications but with much lower horizontal resolution. Although not specifically designed for tropical cyclones, the model solutions contain smoothed representations of these storms. One of the challenges in using global atmospheric model for hurricane applications is objectively determining what is a tropical cyclone, given the three dimensional solutions of atmospheric variables. This is especially difficult in the lower resolution climate models. To address this issue, without manually selecting features of interests, the initial conditions from a low resolution version of the GFS (2 degree latitude-longitude grid) are examined at 6 hour periods and compared with the known positions of tropical cyclones. Several Python modules are used to build a prototype model quickly, and the prototype model shows fast and accurate prediction with the low resolution GFS data.

Index Terms—hurricane, prediction, GFS, SVM

Introduction

The devastating effects from tropical storms, hurricanes[1] , and typhoons on life and property places great importance on forecasting and warning systems [CAM02]. To minimize the possible damages from hurricanes, we need fast and accurate forecasts as early as possible. Even with the significance of predicting a hurricane, the procedure for identifying the initial position and intensity of tropical cyclones is not fully automated. From direct measures from aircraft, ships and surface stations and remote sensing observations, including satellite imagery and Doppler radar that is collected over time, meteorologists identify a storm, and it is cumbersome process.

Numerical models are used to forecast the future position, intensity and structure of hurricanes. For climate applications, the resolution of these models is marginal for representation of hurricanes. Different schemes are proposed to detect tropical cyclone-like vortices (TCLVs) in general circulation model (GCM) simulations, which rely on threshold values of observed characteristics of actual tropical cyclones. However, it is ad-hoc to use a different threshold for hurricane prediction [WAL04]. Although there is some research ongoing to improve the reanalysis approach that determines the threshold, currently there is no good representation of actual hurricane structure for reanalysis. Furthermore, considering potential changes of hurricane intensity [WAL04], faster and simple approaches are required for practical use.

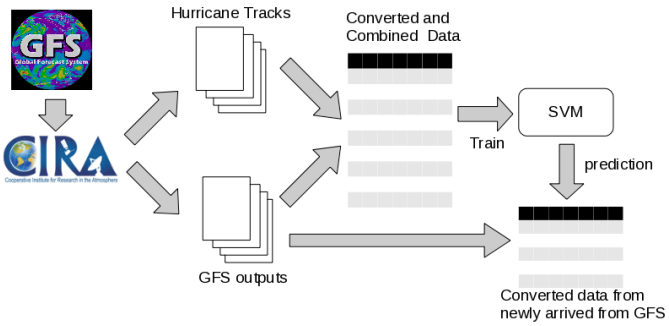

Fig. 1: *Hurricane prediction procedure.*

This paper examines a method to automate the objective identification of hurricanes in global model forecast fields by using a machine learning approach, support vector machines (SVM), based on Global Forecast System (GFS) analyses. The outputs from GFS [SAH06, EMC03] that produce forecasts of wind speed and direction, humidity, and temperature are used as source for hurricane prediction without any filtering based on previous knowledge. From these features at each grid point, SVMs can be trained to make an accurate prediction of hurricane occurrence.

Figure 1 shows the sequential procedure for hurricane prediction from data conversion to final hurricane prediction. Python provides useful packages to reduce the time for prototyping this hurricane prediction procedure. Using Numpy, the basic data matrices for meteorological features in each grid are stored and manipulated. Matplotlib is used to analyze the patterns of the data features, and the data is trained and classified by using PyML SVM. PyGTK with Glade and Basemap generate the graphical user interface to connect the sequential process of preparing data, training a classifier, predicting hurricanes and presenting prediction results to users.

Global Forecast System and Hurricane Tracks

To predict hurricanes, the first step is to access the weather data. In this paper, we choose the output from the U.S. National Centers for Environmental Prediction (NCEP) Global Forecast System (GFS). For this initial prototype, low resolution GFS analysis fields are

* Corresponding author: lemin@cs.colostate.edu
§ Colorado State University
‡ NOAA/NESDIS/STAR

1. In this paper, the term hurricane is used generically to represent tropical cyclones of all intensities, even though, technically speaking, a tropical cyclone must have winds greater than 63 kt to be classified as a hurricane.

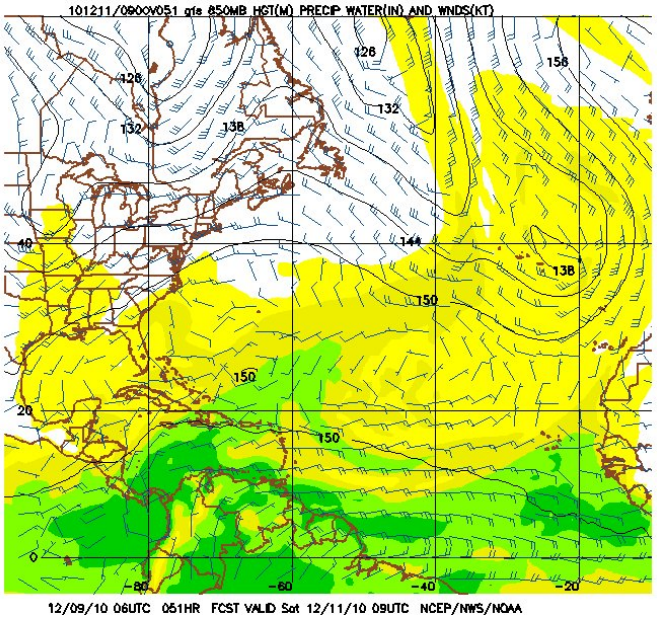

Fig. 2: *The wind component and humidity 2-D plot at a fixed vertical pressure level. (http://mag.ncep.noaa.gov/NCOMAGWEB/appcontroller)*

used, rather than the GFS model forecasts. Similar GFS analysis data are available in real time from NCEP (http://www.nco.ncep.noaa.gov/pmb/products/gfs/) along with the forecast fields.

The GFS data set contains wind speed and direction, temperature, geopotential height deviation and relative humidity in a meteorological 3-D grid along with the year, month-day, time, longitude, and latitude. The vertical coordinate of the 3-D grid represents pressure level, where 100 hPa is near the top of the atmosphere and 1000 hPa is near the surface. Figure 2 shows example wind vectors at fixed vertical level (850 hPa of pressure).

This paper uses a low resolution GFS data with the longitude and latitude intervals of 2 degrees and recording interval of 6 hours (0, 6, 12 or 18 UTC). This is similar to what might be obtained from a long-term climate simulation. Along with GFS outputs, hurricane tracks are used as labels for hurricane locations. Hurricane tracks contain storm number, year, month, day, time, and storm information such as latitude, longitude, maximum winds, minimum pressure at the storm center, storm type, and basin. For this research, each storm location and time information is extracted to use them as labels for hurricane prediction training.

Data Preprocessing for Hurricane Detection

Raw GFS data and hurricane tracks cannot be used directly; data preprocessing is necessary for efficient hurricane prediction. Since the goal of the research is predicting the longitudinal and latitudinal location of hurricanes, all the vertical coordinates can be combined at each grid point. Each location on the earth, specified by its latitude and longitude, is covered by a 3-D grid cell of GFS data. We chose to combine the GFS data from the four grid cell corners at all 11 heights by concatenating them into one vector, as illustrated in Figure 3. The presence or absence of a hurricane at each location is indicated by a 1 or -1, respectively, as the first element of the vector. Thus, each sample contains $1 + 11 \times 4 \times 8 = 353$ values.

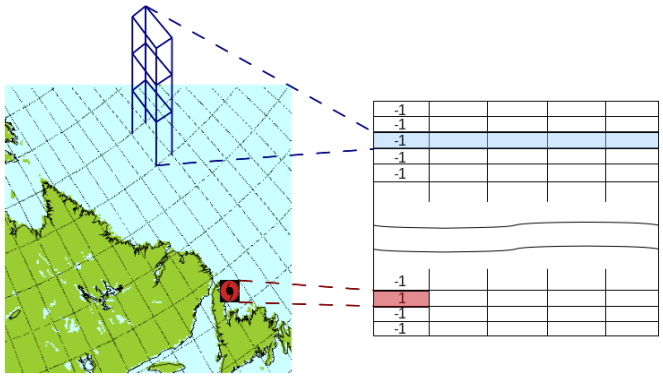

Fig. 3: *Converting GFS data and hurricane tracks for hurricane prediction.*

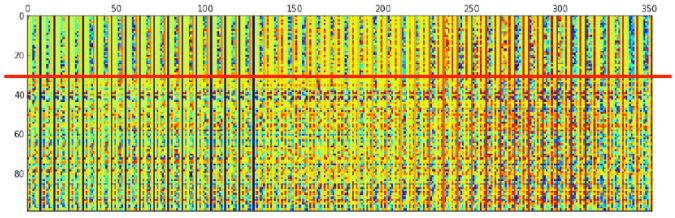

Fig. 4: *Image map for the converted data. First 33 rows are the grids that have hurricanes, and the rest rows are randomly sampled grids.*

From July 1st, 2008 through July 4th, 2008, there are 194,400 sample grid cells, and only 33 of them contains hurricanes. To examine the difference between hurricane cells and the others, the preprocessed data representation can be visualized by combining samples as the rows of a Numpy array and displayed as an image using Matplotlib. In Figure 4, the first 33 rows represent locations with hurricanes during the time period, and the other rows are randomly selected locations that do not have hurricanes. The image shows that the data patterns are significantly different between hurricane locations and the other samples. There is less variation in some columns in the first 33 rows, the locations contain hurricanes, than in the last 66 rows, locations without hurricanes.

Numpy and Matplotlib for Data Preprocessing and Analysis

Numpy is the fundamental package that is used as a multi-dimensional container. In this research, Numpy provides the basic data structure for converted data representations and operations. It includes various tools for data handling such as reading and storing files, linear algebra, and matrix manipulation. Data conversion in Figure 3 is easily implemented by using Numpy functions and array object. The matshow() function in Matplotlib is used to generate the image in Figure 4 and gives a first look at the data pattern. Although eye observation of data is not always successful for the general machine learning approaches, it is useful for the GFS data.

Support Vector Machine

Support Vector Machine (SVM) [ASA08,BIS06] is a popular tool for classification, regression, and novelty detection. An important property of support vector machines is that the determination of the model parameters corresponds to a convex optimization

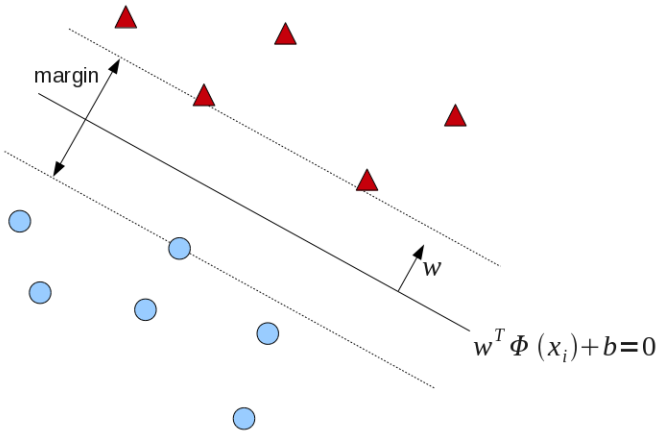

Fig. 5: *Support Vector Machine.*

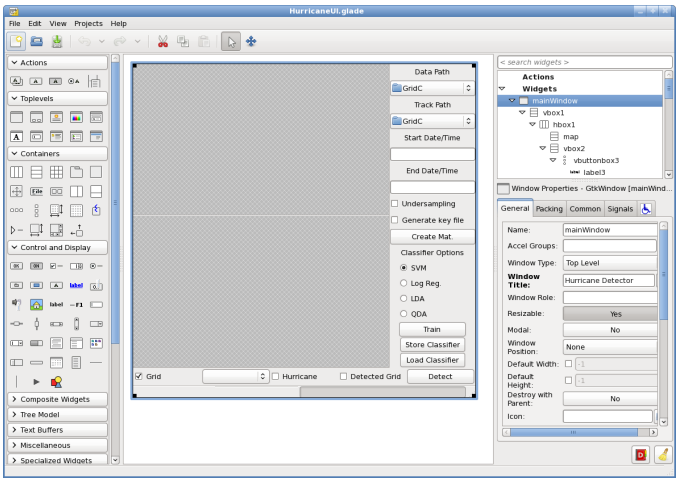

Fig. 6: *Glade-3 for creating the GUI for hurricane prediction.*

problem, thus a local solution is a global optimum [BIS06]. Figure 5 shows the binary SVM margin maximizer for classification and is explained below.

Support vector machine is characterized by its margin: it looks for the hyperplane that separates data into two classes with maximum margin. Let the training data be (x_i, y_i) with m input vectors x_i and target values $y_i \in -1, 1$. The hyperplane can be defined

$$f(x) = w^T \Phi(x) + b \tag{1}$$

where $\Phi(x)$ denotes a nonlinear function. The w is the weight vector, and scalar b is the bias. Thus, the margin separation into two half spaces can be defined

$$\begin{cases} y_i = -1 & \text{if } w^T \Phi(x_i) + b \leq -1 \\ y_i = 1 & \text{if } w^T \Phi(x_i) + b \geq 1 \end{cases}$$

If the data is linearly separable, we can find a hyperplane such that

$$y_i f(x_i) \geq 1$$

If we scale the hyperplane in Equation 1, we get the margin that is $\frac{2}{\|w\|}$. Since maximizing $\frac{2}{\|w\|}$ is equivalent to minimizing $\frac{\|w\|}{2}$, the hard margin SVM that seeks a maximum margin can be written as a linear program:

$$\begin{aligned} &\text{minimize} && \frac{\|w\|}{2} \\ &\text{subject to} && y_i(w^T \Phi(x) + b) \geq 1 \end{aligned}$$

In practice, the data is not always linearly separable. In such data, by allowing some misclassified points, we can get larger margins. Some previous theoretical and experimental study shows that larger margin will generally perform better than hard margin SVM [ASA08]. We can define the slack variables $\xi_i > 0$ to allow errors.

$$y_i(w^T \Phi(x) + b) \geq 1 - \xi_i$$

Now, adding control parameter C, we can rewrite the previous linear program:

$$\begin{aligned} &\text{minimize} && \frac{\|w\|}{2} + C\sum_{i=1}^{m} \xi_i \\ &\text{subject to} && y_i(w^T \Phi(x) + b) \geq 1 - \xi_i \\ & && \xi_i > 0 \end{aligned}$$

where m is the number of points. C controls the conflicting objectives, maximizing the margin and minimizing the sum of errors. When C is large, a large penalty is given to errors, it reduces the margin that minimizes the error term. When C is small, it allows more errors resulting in margin increase.

PyML

PyML is a machine learning library that focuses on SVM and kernel methods. As other python packages such as scikit-learn, shogun, orange, and mlpy, PyML efficiently wraps the state of the art SVM library, libsvm. PyML provides several dataset containers that hold class labels and a collection of data patterns. The Numpy array object concatenating our hurricane data can be easily converted to VectorDataSet in PyML. Since we have observed the significant difference between hurricane and non-hurricane data patterns, we apply a simple linear kernel for classification. Based on the dataset and linear kernel, the SVM is trained for hurricane prediction.

PyGTK and Glade for User Interface

For converting the raw data, training SVM, and finally predicting hurricanes, a simple interface prototype can be easily constructed by using PyGTK and Glade-3. Glade is a rapid application development tool to enable fast user interface design. Glade-3 tool in Figure 6 makes it easy to create the base UI for hurricane prediction. Instead of writing the codes for the placement, color, or type of each widget, the UI created in Glade-3 is stored in XML, and the XML file is loaded in the python program with PyGTK. This saves a fair amount of time for creating the GUI. The user interface is composed of right side inputs and buttons for GFS data and tracks file selection and converting with some options and for training a classifier and saving or loading the trained classifier. When a trained classifier is ready, the bottom interface is used to predict hurricanes after selecting the GFS data to apply to the classifier. The major part of the UI plots prediction results on a map by using Basemap. Check buttons on the bottom menu are for plotting options. The following code snippet shows the simple usage to load the glade UI (the prototype codes will be available on http://www.cs.colostate.edu/~lemin/hurricane/):

```
import gtk, gobject, cairo
import gtk.glade

gladefile = "HurricaneUI.glade"
builder = gtk.Builder()
```

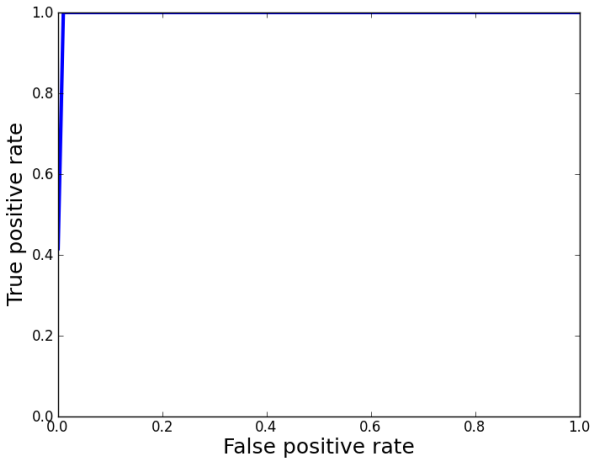

Fig. 7: *ROC curve for the support vector machine.*

```
builder.add_from_file(gladefile)
self.window = builder.get_object("mainWindow")
builder.connect_signals(self)
```

Basemap to locate hurricane and prediction

Basemap is an add-on toolkit for Matplotlib that enables plotting data over map projections. Coastlines, political boundaries, longitude and latitude grid lines are available in several different resolutions. Provided map projection coordinates and plotting functions make it easy to visualize predicted locations and actual hurricanes on the globe. Figure 8 shows the GUI for hurricane prediction. Orthogonal Basemap for the globe is projected in the middle of the interface and when the trained SVM is applied to the test data, it can show the hurricane locations as well as the predicted hurricane locations depending on the display options. Basemap on the interface can be loaded as below:

```
from mpl_toolkits.basemap import Basemap

self.map = Basemap(projection='ortho',
                   lat_0 = lat, lon_0 = lon,
                   resolution = 'l',
                   area_thresh = 1000., ax=ax)

self.map.drawcoastlines(ax=self.ax)
self.map.drawcountries(ax=self.ax)
self.map.drawlsmask(land_color='yellowgreen',
                    ocean_color='#CCFFFF',
                    lakes=True, ax=self.ax)
```

Hurricane Prediction

Using 2008 GFS data and hurricane tracks, we ran a simple experiment for hurricane prediction. Running the codes below for 5-fold cross-validation achieves 0.9998 of success rate (0.8458 balanced success rate). The almost square ROC curve (Figure 7) shows the accuracy of the proposed framework. The computed ROC/ROC_50 scores are 0.999808 and 0.916524 respectively.

```
import PyML as pyml

data = pyml.VectorDataSet(filen, labelsColumn=0)
s = pyml.SVM()
result = s.cv(data)
```

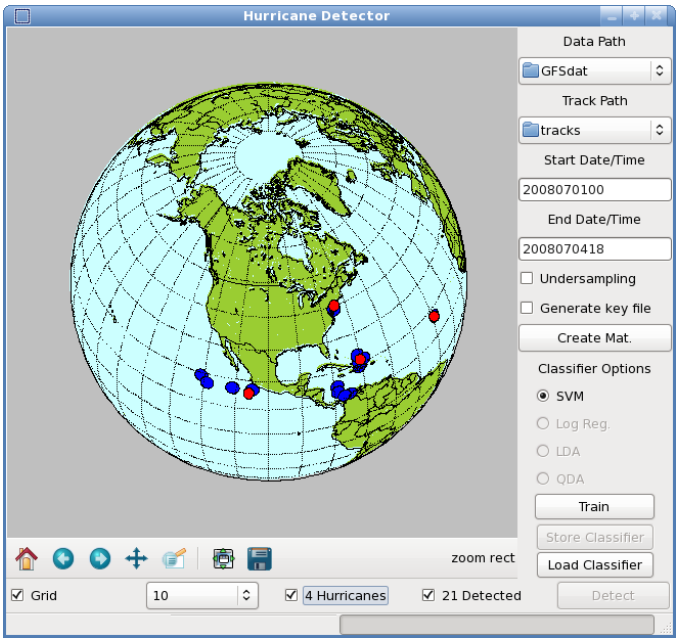

Fig. 8: *Hurricane prediction and actual hurricanes. Blue circles indicate predicted hurricane locations, and red circles indicate the actual hurricane locations.*

Now, we train SVM with four days of GFS data and hurricane tracks from July 1st to July 4th in 2008. The trained SVM predicts hurricane locations of one and half months later. It is tested on the data for August 29th when Hurricane Gustav neared the west side of Cuba, and it predict the actual hurricane or near hurricane locations successfully. Even with a short period time for training samples, it found all hurricane locations without an error in testing data: the prediction picks 21 grid cells including all four hurricane locations. Figure 8 shows that even with over estimation of hurricane locations, it predicts all the hurricanes. Furthermore, the false positives are neighboring locations that can be the area that hurricanes affect the atmospheric conditions close to the data pattern of true hurricane locations. Training and prediction is done simply by reading data files and calling train() and test() functions:

```
import PyML as pyml

data = pyml.VectorDataSet(filen, labelsColumn=0)
s = pyml.SVM()
s.train(data) # training
test_data = pyml.VectorDataSet(testfn, labelsColumn=0)
result = s.test(test_data) # prediction
```

Conclusion

In summary, we presented the hurricane prediction problem, how it can be tackled objectively with a machine learning approach, and how python packages are applied to prototype the hurricane prediction. For the proposed approached, meteorologists do not need to select features of interests anymore. To show this, various python packages are used for fast and efficient prototyping that solves the hurricane prediction problem: Numpy for converting GFS data and hurricane tracks, Matplotlib for analyzing the data patterns, PyML for binary classification of hurricanes, and PyGTK, Glade, and Basemap for the graphical user interface.

This machine learning approach will be able to contribute developing fast and objective adaptation model for hurricane

prediction without manual feature selection. Although the connection between global warming and hurricanes is not clear, some research such as [WAL04] points out that changes in the number of storms and the maximum intensities are likely to happen as climate changes. Considering the hurricane changes over time, online adaptation models for hurricane prediction needs to be investigated. The various python packages will be an excellent choice for use in future research as well.

REFERENCES

[SAH06] Saha S, Nadiga S, Thiaw C, Wang J, Wang W, Zhang Q, Van den Dool HM, Pan H-L, Moorthi S, Behringer D, Stokes D, Pena M, Lord S, White G, Ebisuzaki W, Peng P, Xie P. *The NCEP climate forecast system*, J Clim 19(15):3483–3517. doi: 10.1175/JCLI3812.1, 2006.

[EMC03] Environmental Modeling Center. *The GFS Atmospheric Model*, NOAA/NCEP/Environmental Modeling Center Office Note 442, 14 pp. 2003 [Available online at http://www.emc.ncep.noaa.gov/officenotes/FullTOC.html.].

[CAM02] Camargo SJ, Zebiak SE. *Improving the detection and tracking of tropical cyclones in atmospheric general circulation models*, Technical Report No. 02–02. International Research Institute for Climate Prediction, Palisades, NY, 2002.

[WAL04] K. Walsh. *Tropical cyclones and climate change: Unresolved issues*, Climate Res., 27, 78–83, 2004.

[HOU01] J.T. Houghton, Y. Ding, D.J. Griggs, M. Noguer, P.J. van der Linden, X. Dai, K. Maskell, C.A. Johnson, *Climate Change 2001: The Scientific Basis*, Contribution of Working Group I to the Third Assessment Report of the Intergovernmental Panel on Climate Change (IPCC). Cambridge University Press, Cambridge and New York, 2001.

[ASA08] A. Ben-Hur, C.S. Ong, S. Sonnenburg, B. Sch"{o}lkopf, and G. R"{a}tsch. *Support vector machines and kernels for computational biology*, PLoS Comput Biol, 4(10):e1000173, 2008.

[BIS06] C.M. Bishop. *Pattern recognition and machine learning*, volume 4. Springer New York, 2006.

IMUSim - Simulating inertial and magnetic sensor systems in Python

Martin J. Ling‡*, Alex D. Young‡

Abstract—IMUSim is a new simulation package developed in Python to model Inertial Measurement Units, i.e. devices which include accelerometers, gyroscopes and magnetometers. It was developed in the course of our research into algorithms for IMU-based motion capture, and has now been released under the GPL for the benefit of other researchers and users. The software generates realistic sensor readings based on trajectory, environment, sensor and system models. It includes implementaions of various relevant processing algorithms and mathematical utilities, some of which may be useful elsewhere. The simulator makes extensive use of NumPy, SciPy, SimPy, Cython, Matplotlib and Mayavi. The rapid development enabled by these tools allowed the project to be completed as a side project by two researchers. Careful design of an object-oriented API for the various models involved in the simulation allows the software to remain flexible and extensible while requiring users to write a minimum amount of code to use it.

Index Terms—simulation, IMU, accelerometer, gyroscope, magnetometer

Introduction

Inertial sensors—accelerometers and gyroscopes—are becoming increasingly ubiquitous in a wide range of devices and applications. In particular, they are often used to automatically find and track the orientation of a device. For this role they may be combined with a magnetometer, which can sense the direction of the Earth's magnetic field. Such a combination can determine the device's full 3D orientation, including compass heading. Devices designed specifically around these sensors are called Inertial Measurement Units (IMUs), though readers may be more familiar with them in modern smartphones, tablets and gaming controllers, which use the orientation and movements of the device as a user input mechanism.

Outside of consumer devices inertial and magnetic sensors find a wide range of uses. They are used for attitude tracking in aircraft, spacecraft, in many types of robotic systems, and in stabilised platforms for cameras and weapons. In engineering and industry they are used to detect and monitor vibrations, impacts, collisions and other events. They have also been widely used in healthcare, to monitor and classify the activities of a patient and to detect events such as falls. In biology and agriculture, they have been used to provide the same capabilities on animals. The list of applications of these sensors continues to grow, with more uses being found as their cost, size and power requirements decrease. However, as newer and more ambitious applications push towards the limits of sensor capabilities, development becomes harder.

Our own research over the past few years has focused on motion capture of the human body using networks of wearable IMUs. Most motion capture methods are based on cameras, and consequently have limited tracking areas and problems with occlusion. By instead tracking movements using IMUs on the body, motion capture can be freed from these limitations. However, achieving accurate tracking with this approach is a difficult problem and remains an active topic of research.

During our work in this area we invested a large amount of effort in designing, building and debugging both hardware and software to test our ideas. Some other research groups did similar work, on their own platforms, whilst further researchers developed algorithms which were tested only in their own simulations. We were not readily able to compare our methods in controlled experiments, and new researchers could not easily enter the field without investing significant time and money in the necessary infrastructure.

In our view, a significant obstacle for the development of advanced inertial/magnetic sensing applications has been a lack of useful simulation tools to allow a sensor system to be designed, modelled and tested before expensive and time-consuming hardware work is required. Ad-hoc simulations are sometimes developed for individual applications or hardware but due to their very specific nature these are rarely shared, and even if they are, hard to reuse. As a result, most simulations are created from scratch and tend towards being simplistic.

We therefore decided that a useful contribution we could make to this field would be an openly available simulation framework, that could continuously evolve to support state-of-the-art work with the best available methods and models. We hence wanted to keep the design as flexible, extensible and general purpose as possible.

Although most other researchers in this area were using MATLAB, we decided that Python would be the best platform for developing our simulator. Ease and speed of development were essential since we would be undertaking the project with just two people, and alongside our main research. We also felt that a popular general purpose language with a supporting open source ecosystem, rather than a proprietary tool, was the appropriate choice given the project's goals.

After several months we released our first version under the GPL in April 2011, and presented a paper [Young2011] that discussed the work from a primarily scientific perspective. In

* Corresponding author: m.j.ling@ed.ac.uk
‡ University of Edinburgh

contrast, this paper focuses on the Python implementation and our experiences during its development.

Overview

The goal of IMUSim is to allow inertial/magnetic sensor systems to be tested in simulations that are quick to develop yet as realistic as possible, reducing development time, cost, and risk in both academic research and commercial development.

The key function of the software is hence to generate realistic readings for sensors in simulated scenarios. The readings obtained from an inertial or magnetic sensor at a given instant depend on several factors:

- The trajectory followed by the sensor through space: its position and rotation at a given moment, and derivatives of these—in particular angular velocity and linear acceleration.
- The surrounding environment: in particular the gravitational and magnetic fields present.
- The nature of the sensor itself: its sensitivity, measurement range, bias, etc.
- The analogue-to-digital converter (ADC) used to sample the sensor output: its range, resolution, linearity, etc.
- Random noise associated with either the sensor or ADC.

We simulate all of these factors, taking an object-oriented approach. For each factor involved—e.g. trajectory, sensor, magnetic field—we provide an abstract class defining a model interface, and classes implementing specific models. All the models involved in the simulation can thus be easily interchanged, and extended or replaced as required.

In addition to just the model classes required to obtain simulated sensor readings, the IMUSim package also includes:

- A basic framework of model classes for simulating multi-device wireless systems with distributed processing.
- Implementations of existing processing algorithms for inertial and magnetic sensor data, including methods for sensor calibration, orientation estimation, body posture reconstruction and position estimation.
- General purpose mathematical utilities useful for implementing models and processing algorithms.
- 2D and animated 3D visualisation tools.

Rather than developing a specific UI for the simulator which would inevitably be restrictive, we designed the package to be easily used interactively via the [IPython] shell, or by scripting. A tutorial [Ling2011] has been written which aims to quickly introduce the use of the simulator through interactive examples with IPython, assuming some knowledge of the field but no previous Python experience. This tutorial accompanies the full API reference, which is generated using Epydoc from comprehensive docstrings included in the code.

The implementation makes extensive use of [NumPy], [SciPy], [SimPy], [Matplotlib], [MayaVi] and [Cython], and in general aims to use existing libraries wherever possible. In a few cases we have implemented limited amounts of functionality that could have been reused from elsewhere. Reasons for doing this have included performance (e.g. our fast Cython quaternion math implementation), maintaining ease of use and consistency of the API, or limiting the installation prerequisites to the common and well-supported libraries included in the main scientific Python distributions.

A quick example

In this section we look briefly at the IMUSim software starting from the user's perspective, and then at some aspects of the implementation. We begin by looking at a simple example script, which simulates an idealised IMU following a randomly generated trajectory, sampling its sensors at 100Hz:

```
# Import all public symbols from IMUSim
from imusim.all import *

# Create a new simulation
sim = Simulation()

# Create a randomly defined trajectory
trajectory = RandomTrajectory()

# Create an instance of an ideal IMU
imu = IdealIMU(simulation=sim, trajectory=trajectory)

# Define a sampling period
dt = 0.01

# Set up a behaviour that runs on the simulated IMU
behaviour = BasicIMUBehaviour(platform=imu,
    samplingPeriod=dt)

# Set the time inside the simulation
sim.time = trajectory.startTime

# Run the simulation till the desired end time
sim.run(trajectory.endTime)
```

The package has been designed to make simple tasks like this quick to write, and to only require lengthy setup code for a simulation when unusual and complex things are required. The `imusim.all` package automatically imports all public symbols from the various subpackages of `imusim`. The `Simulation` object wraps up the three things required for an individual simulation run: simulation engine, environment model, and random number generator (RNG). Unless told otherwise, it includes a randomly seeded RNG and a default environment model with nominal values for Earth's gravity and magnetic field. The `IdealIMU` class models a complete IMU device with accelerometer, magnetometer, gyroscope and supporting hardware components, all using ideal models. `BasicIMUBehaviour` implements the most common software functionality required on an IMU—sampling all its sensors at regular intervals, storing the resulting values and, if specified in options to its constructor, passing them on to processing algorithms.

The behavioural code accesses the simulated hardware it has been given through a defined API, allowing it to be written in straightforward Python code as if running on real hardware. The simulated hardware components then post events to the SimPy simulation engine as necessary to model their functionality. In this case, the main events will be the samples requested from the sensors via the ADC. At the moments these samples are taken, the sensor models will request information from the trajectory and environment models to which they are attached, as needed to compute their outputs. The ADC model will in turn process each value, and generate a final reading. After each event is simulated the simulation time advances directly to the next requested event. Depending on the user's computer and the complexity of the simulation, time may pass from a little faster to very much slower, compared to real time.

We display some progress output to keep the user informed. In the simple case above the simulation is quick:

```
Simulating...
```

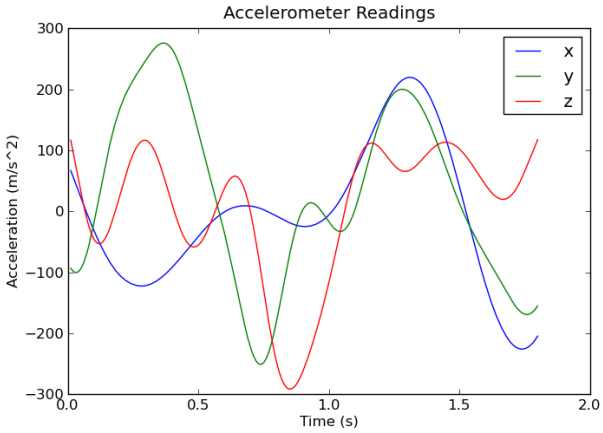

Fig. 1: *Accelerometer readings for an ideal accelerometer following a randomly curving trajectory.*

```
Simulated 0.1s of 1.8s (  5%).
Estimated time remaining 0.4s
...
Simulation complete.
Simulated 1.8 seconds in 0.4 seconds.
```

The user can now interactively explore the results via the same objects that were used in the simulation. For example, plotting the accelerometer samples from the IMU:

```
>>> plot(imu.accelerometer.rawMeasurements)
```

plus appropriate labels, gives the graph shown in Figure 1. Plotting uses the normal facilities of Matplotlib, but IMUSim provides its own `plot` function. This adds special support for its own data types whilst retaining backward compatibility.

Data types

The parameter passed to `plot` above was a `TimeSeries` object, one of the basic data types we developed for IMUSim. It represents timestamped scalar, vector or quaternion values with optional uncertainty information. We developed the `TimeSeries` class initially as a simple container, because we found that when plotting or otherwise passing around such data, it was often difficult or awkward to keep track of the correct combinations. We later included support for adding data sequentially, which is useful for storing data as it is generated by the simulation. New data points are appended to a list internally, with contiguous NumPy array versions generated only when required.

A `TimeSeries` thus provides two essential attributes, `timestamps` and `values`. The `timestamps` attribute is an array of time values in ascending order:

```
>>> imu.accelerometer.rawMeasurements.timestamps
array([ 0.01,  0.02, ...,  1.79,  1.8 ])
```

These are times at which the samples were taken. In this case they are uniformly distributed but any sequence of times may be represented. The sample values themselves are found in the `values` attribute:

```
>>> imu.accelerometer.rawMeasurements.values
array([[  66.705814  , ..., -204.6486176 ],
       [ -93.40026896, ..., -155.16993659],
       [ 116.56420017, ...,  117.56964057]])
```

Note the shape of this array, which is 3xN where N is the number of timestamps. IMUSim uses column vectors, in order to work correctly with matrix multiplication and other operations. Arrays of vector data are therefore indexed first by component and then by sample number. A single vector would be represented as a 3x1 array. IMUSim provides a `vector` function to concisely construct these:

```
>>> vector(1,2,3)
array([[ 1.],
       [ 2.],
       [ 3.]])
```

The other important data type is the quaternion, which is a mathematical construct with four components that can be used to represent a rotation in 3D space; see [Kuipers2002] for an in-depth treatment. Quaternions offer a more compact and usually more computationally efficient representation than rotation matrices, while avoiding the discontinuities and singularities associated with Euler angle sequences. IMUSim provides its own `Quaternion` class. Although a number of quaternion math implementations in Python already exist, we developed our own in Cython for performance reasons, due to the large number of quaternion operations used in the simulator. We hope this component will prove to be usefully reusable.

Quaternions can be constructed directly, converted to and from from other rotation representations such as Euler angle sequences and rotation matrices, used in mathematical expressions, and applied to perform specific operations on vectors:

```
>>> q1 = Quaternion(0, 1, 0, 0)
>>> q1.toMatrix()
matrix([[ 1.,  0.,  0.],
        [ 0., -1.,  0.],
        [ 0.,  0., -1.]])
>>> q2 = Quaternion.fromEuler((45, 10, 30), order='zyx')
>>> q1 * q2
Quaternion(-0.2059911, 0.8976356, -0.3473967, 0.176446)
>>> q2.rotateVector(vector(1,2,3))
array([[ 0.97407942],
       [ 1.30224882],
       [ 3.36976517]])
```

As mentioned, the `TimeSeries` class can also be used with quaternion values. The rotations of the random trajectory used in the previous example simulation were generated from a time series of quaternion key frames:

```
>>> trajectory.rotationKeyFrames.values
QuaternionArray(
    array([[-0.04667, -0.82763,  0.29852, -0.47300],
           [-0.10730, -0.81727,  0.33822, -0.45402],
           ...,
           [ 0.40666, -0.04250,  0.80062,  0.43796],
           [ 0.42667, -0.01498,  0.82309,  0.37449]]))
```

Arrays of quaternions are represented using the special `QuaternionArray` class, also implemented in Cython, which wraps an Nx4 NumPy array of the component values. Quaternion arrays provide support for applying quaternion math operations efficiently over the whole array.

Trajectory models

The data types we have just introduced form the basis for our trajectory model interface. A trajectory defines the path of an object through space, and also its changing rotation, over time. To allow simulating inertial and magnetic sensors, a trajectory

needs to provide position and rotation, and their first and second derivatives, at any given time. A trajectory must also give the start and end of the period for which it is defined. In this case we will look at a trajectory's parameters at its starting time, which is a scalar in seconds:

```
>>> t = trajectory.startTime
>>> t
3.8146809461460811
```

The position, velocity and acceleration methods of a trajectory provide vector values, in SI units, at given times:

```
>>> trajectory.position(t) # m
array([[-10.36337587],
       [  4.63926506],
       [ -0.17801693]])
>>> trajectory.velocity(t) # m/s
array([[ 30.79525389],
       [-20.9180481 ],
       [  2.68236355]])
>>> trajectory.acceleration(t) # m/s^2
array([[ 178.30674569],
       [ -15.11472827],
       [  15.54901256]])
```

The rotation at time t is a quaternion, but its derivatives—angular velocity and acceleration—are vectors:

```
>>> trajectory.rotation(t)
Quaternion(-0.046679, -0.82763, 0.29852, -0.47300)
>>> trajectory.rotationalVelocity(t) # rad/s
array([[-2.97192064],
       [ 2.97060751],
       [-7.32688967]])
>>> trajectory.rotationalAcceleration(t) # rad/s^2
array([[ -8.46813312],
       [ 19.43475152],
       [-31.28760834]])
```

Note that angular accelerations may be required, even when only angular velocity sensors (gyroscopes) and linear accelerometers are simulated. This is because sensors may be placed at offsets from a trajectory, e.g. on the surface of a rigid body is whose centre is following the trajectory. In the equation for linear acceleration at an offset from a centre of rotation, an angular acceleration term is present.

Any object which implements the methods above at can be used as a trajectory model by IMUSim. The trajectory can be defined in advance, or may be defined as a simulation progresses, e.g. by simulating the effect of some control system. The simulator will only call the trajectory methods for a time when all events prior to that time have been simulated.

Since defining realistic trajectory models is one of the most difficult aspects of IMU simulation, much of the code in IMUSim is devoted to assisting with this. In particular, we provide tools for defining trajectories from existing motion capture data in various formats. Using such data requires the creation of continuous time trajectories, with realistic derivatives, from discrete time position and/or rotation information.

From sampled position data, interpolated values and derivatives can be obtained by fitting three independent cubic spline functions to the x, y, and z components of the data, using the `splrep` and `splev` functions from `scipy.interpolate`. Obtaining usable rotational derivatives from sampled rotations is more complicated. The most common forms of quaternion interpolation, the SLERP [Shoemake1985] and SQUAD [Shoemake1991] algorithms, are continuous only in rotation and angular velocity

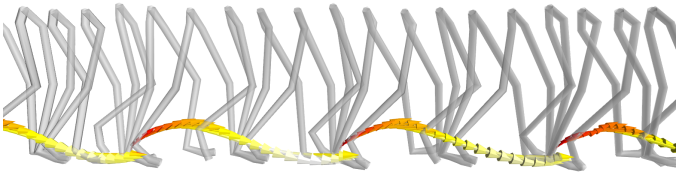

Fig. 2: *Interpolated trajectories from motion capture data, for the lower body of a walking human. The source data was in BVH format at 120 Hz. The model posture is displayed at 5 Hz, and the velocity vector obtained for the right foot is displayed at 50 Hz.*

respectively, and hence cannot provide a continuous angular acceleration. We developed a Cython implementation of the quaternion B-spline algorithm of [Kim1995], which provides the necessary continuity.

For both position and rotation data, it is usually necessary to use smoothing splines to avoid overfitting to noisy capture data, if realistic derivative values are to be obtained. Appropriate smoothing can be achieved by providing expected standard deviations of the input data. Our code then provides the appropriate parameters to `splrep`.

In many applications sensors are used to measure the movements of jointed but otherwise rigid structures, such as the human skeleton or a jointed robotic arm. We therefore provide specific trajectory classes for modelling articulated rigid-body systems, that obey their kinematic constraints. In particular, these classes are useful to work with human motion capture data, which is often pre-processed to fit this type of model and stored accordingly, in formats such as BVH and ASF/AMC. We provide loaders for these file formats, and splining wrapper classes that make it a simple to obtain physically consistent trajectories from such data. Figure 2 illustrates model trajectories and a derivative obtained in this manner, rendered using IMUSim's 3D visualisation tools, which are based on MayaVi.

Environment models

The second factor affecting sensor readings is the environment. Accelerometers sense gravity, and magnetometers sense magnetic field, both of which can vary with position and time. We may also want to simulate radio transmissions from a wireless IMU, the propagation of which will depend on its surroundings. All of these considerations are described by an `Environment` object, to which we assign models for each aspect of the environment relevant to the simulation.

If not otherwise specified, each `Simulation` is created with a default environment, including simple models of the gravitational and magnetic fields at the Earth's surface. Both are subclass instances of the abstract `VectorField` class, which defines an interface for time-varying vector fields. Field values can be obtained by calling the models with a position vector and time:

```
>>> p = trajectory.position(t)
>>> sim.environment.gravitationalField(p, t) # m/s^2
array([[ 0.  ],
       [ 0.  ],
       [ 9.81]])
>>> sim.environment.magneticField(p, t) # in Tesla
array([[  1.71010072e-05],
       [  0.00000000e+00],
       [  4.69846310e-05]])
```

Fig. 3: *Unstructured measurements of magnetic field distortion used to initialise an interpolated field model.*

On Earth, and within a small area, it is generally sufficient to model gravity as a constant field. For Earth's magnetic field, approximate values for a given location can be obtained from the International Geomagnetic Reference Field model [IGRF] and passed to the `EarthMagneticField` constructor. However, local distortions can be very significant, so we provide means for modelling varying fields. The `SolenoidMagneticField` class simulates the magnetic field around a single ideal solenoid, using the equations of [Derby2010]. More complex fields can be modelled by superposition of multiple solenoids. Alternatively, known field values at certain positions can be used to create an interpolating field model. This requires an $\mathbb{R}^3 \rightarrow \mathbb{R}^3$ interpolation on an unstructured grid, for which we use the Natural Neighbour algorithm described in [Hemsley2009]. Our code provides a wrapper for the C implementation of this algorithm [interpolate3d]. Figure 3 illustrates a real set of field measurements around the floor of a steel-framed building. The code allows detailed measurements such as these to be employed in simulations.

Sensor and device models

Real sensors suffer from noise, bias, misalignment, cross-axis sensitivity and many other undesirable effects. To acheive a realistic simulation we need to model these. IMUSim includes generic parametric models for imperfect sensors, and also specific models of some real sensor components, with parameters derived from measurements and datasheet information. All sensor models implement the interface of the abstract `Sensor` class. This defines three methods to be implemented, each of which is a function of time:

- `trueValues` returns a vector of values, one for each axis, that would be measured by an ideal sensor of this type. The units of these values are those of the sensed quantity (e.g. acceleration or angular rate).
- `sensedVoltages` returns the vector of analogue output voltages of the sensor at a given time. This method will internally call `trueValues`, and transform the result via some model of the sensor's transfer function. The result should include deterministic effects, but exclude random noise; i.e. it should be an ensemble mean of the voltages the sensor might actually output at that moment.
- `noiseVoltages` returns randomly generated noise that is additionally measured by the sensor, following an appropriate distribution. Noise values are taken from an individual RNG for the sensor, that is by default seeded from the main simulation RNG, but can be instead seeded explicitly. Running a new simulation with the same initial seed value for the sensor RNG will generate the same noise for that sensor, allowing repeatability with fine-grained control.

One reason for keeping these functions separate is to simplify the composition of different classes to create a sensor model. Usually `trueValues` is inherited from an abstract superclass such as `Magnetometer`, while `sensedVoltages` may be inherited from another class implementing the transfer function, and `noiseVoltages` may come from yet another class. Additionally, having true and noiseless values independently accessible is helpful for comparison and testing.

The final simulated voltage output is the sum of `sensedVoltages` and `noiseVoltages`. In reality, the output voltage is then converted to a digital value by an ADC, which has limited range and resolution and thus clips and quantises the values, as well as adding its own noise. Although somtimes sensor devices have an ADC combined on the same chip, others are interchangeable, and we therefore model ADCs with their own classes separately from sensors.

Another issue in real hardware is that samples are never taken at the exact times requested, because of the inevitable inaccuracy of the IMU's hardware timers. For this reason we also support modelling of imperfect hardware timers.

All of these components can be brought together to create a model of a specific device. The `IdealIMU` we used earlier is an example, with ideal models for all the components of a standard IMU. IMUSim also includes a model, produced from measured parameters, of the real *Orient-3* IMU we developed during our research at Edinburgh [Orient]. This allows users to test algorithms with a realistic model of a complete IMU device 'out of the box'.

The component-based API, including various parametric models and abstract classes implementing common functionality, is designed to make it easy to model a new type of device with a minimum of code. This is the same philosophy we have taken with all parts of the simulator design. For the simulator to be relevant to a wide range of users, and thereby gain an active user base who will contribute to its development, its design must be adaptable enough to support any usage and users must be able to develop new models with minimal difficulty.

A more realistic simulation

Our first example script showed a very unrealistic simulation, with an idealised device following a simple random trajectory. We will now show a brief example of how using IMUSim, much more realistic simulations can be produced with still very little code. This script simulates an *Orient-3* IMU attached to the right foot of a walking human:

```
# Import symbols from IMUSim
from imusim.all import *

# Define a sampling period
dt = 0.01

# Create an instance of a realistic IMU model
```

```
imu = Orient3IMU()

# Create a new environment
env = Environment()

# Define a procedure for calibrating an IMU in our
# selected environment
calibrator = ScaleAndOffsetCalibrator(
    environment=env, samples=1000,
    samplingPeriod=dt, rotationalVelocity=20)

# Calibrate the IMU
cal = calibrator.calibrate(imu)

# Import motion capture data of a human
sampledBody = loadBVHFile('walk.bvh',
    CM_TO_M_CONVERSION)

# Convert to continuous time trajectories
splinedBody = SplinedBodyModel(sampledBody)

# Create a new simulation
sim = Simulation(environment=env)

# Assign the IMU to the simulation
imu.simulation = sim

# Attach the IMU to the subject's right foot
imu.trajectory = splinedBody.getJoint('rfoot')

# Set the starting time of the simulation
sim.time = splinedModel.startTime

# Set up the behaviour to run on the IMU
BasicIMUBehaviour(platform=imu, samplingPeriod=dt,
    calibration=cal, initialTime=sim.time)

# Run the simulation
sim.run(splinedModel.endTime)
```

At 16 lines of code, this is only twice the length of the previous example, but is based on:

- a real human motion, imported from motion capture data and transformed to usable trajectories.
- an empirically obtained model of a real IMU design, including noise and other imperfections.
- a simulation of a real calibration procedure.

Further information on the new steps appearing in this example—including IMU calibration, and more on the use of motion capture data, of which much is freely available—can be found in the IMUSim tutorial [Ling2011].

Plotting the measurements of the accelerometer in this simulation, using the calibration obtained for the IMU, results in Figure 4. Compare the appearance of this data to that from the previous, more simplistic simulation in Figure 1.

Data processing algorithms

Obtaining realistic sensor data in simulations is one of IMUSim's key goals, but the package is also intended to support the comparison, development and selection of algorithms for processing this data. Implementations are included for a number of existing published algorithms. These may be useful as-is in some applications. They may also be used to compare new methods. We encourage users publishing new methods to contribute implementations of their algorithms themselves, and publish the scripts used for their experiments. This allows their results to be reproduced, and

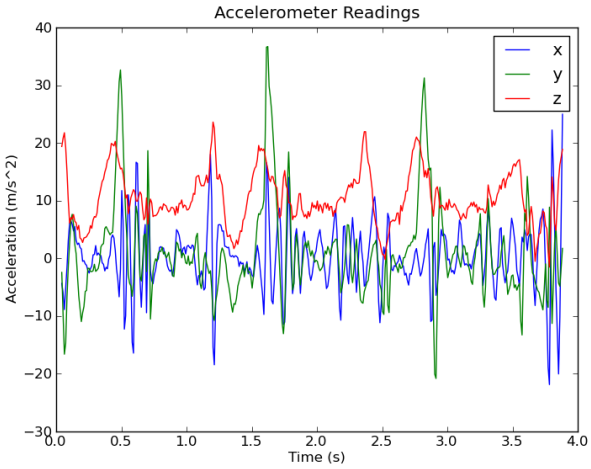

Fig. 4: *Simulated accelerometer readings for an* Orient-3 *IMU attached to the right foot of a walking human.*

reduces the risk that their work will be misrepresented by an incorrect reimplementation by another researcher.

In addition to the library of existing published methods, we have tried to provide some generally useful tools for working with sensor data. In particular, we include generic implementations of the standard linear Kalman filter, the Unscented Transform, and the Unscented Kalman Filter. These are widely useful state estimation and nonlinear system tools, and could be usefully transferred to SciPy or another library.

Validation and testing

In order to test the accuracy of our simulations, we have conducted some experiments to directly compare our simulated sensor values with those measured by real IMUs. To achieve this, we used an optical motion capture system to capture the movements of a subject who was also wearing wireless IMUs. In addition to the normal markers on the subject, the positions and rotations of the IMUs themselves were tracked using three markers attached to each IMU. From the optical capture data we produced a rigid body model of the subject, which was used via the methods we have described to obtain simulated sensor data. We also sampled the magnetic field in the capture area, using the magnetometer of an IMU swept around the capture volume whilst being tracked by the optical system. These measurements, seen in Figure 3, were used to generate an interpolated field model of the capture area which was also used in the simulation.

In our experiments we obtained correlations of $r^2 > 0.95$ between simulated and measured values for all three types of sensors—accelerometers, gyroscopes and magnetometers. More detail on these experiments and results can be found in [Young2011].

The software is accompanied by test scripts designed to be used with the `nosetests` tool. In total the current version runs over 30,000 test cases, which aim to verify the correct behaviour of the code. The tests include checking simulated sensor values against real ones obtained in the experiments described above, to ensure that after any code change the simulator still meets its published claims of accuracy.

We also generate code coverage reports from the tests and use these to identify untested code paths. Unfortunately at present it is not straightforward to obtain test coverage for the Cython parts

of the code; some unofficial code to do this is in circulation, but official future support for this in the `coverage` module would be helpful.

Conclusion

We have presented IMUSim, a simulation framework for inertial and magnetic sensor systems, and looked at some of the details of its Python implementation. The package has been designed to meet the simultaneous goals of:

- enabling accurate simulations,
- remaining as flexible and extensible as possible,
- minimising the amount of code that users must write.

This is achieved by careful design of an object-oriented API for the various models required in the simulation.

The project was completed in a matter of months by two researchers alongside other work. We believe this demonstrates well the rapid development enabled by Python and its increasing range of scientific libraries. In the process of development we created some contributions which may be of wider use, and could be moved to more general purpose libraries. These include:

- fast Cython classes for quaternion mathematics, including efficient quaternion arrays and B-spline fitting of quaternion values.
- generic implementations of the Kalman Filter, Unscented Transform, and Unscented Kalman Filter.
- a `TimeSeries` class for representing extendable time series of scalars, vectors or quaternions with covariance information, and an enhanced `plot` command that accepts these.
- 3D vector field interpolation from unstructured field samples, based on a wrapping of an existing C library for natural neighbour interpolation.

The IMUSim source code is available from the project website at http://www.imusim.org/, under the GPLv3 license. The software is supported by a tutorial, API reference, users mailing list, and test suite.

Acknowldgements

Development of the simulator was in part supported by the UK Engineering and Physical Sciences Research Council under the Basic Technology Research Programme, Grant C523881.

References

[Young2011] A D Young, M J Ling and D K Arvind, *IMUSim: A Simulation Environment for Inertial Sensing Algorithm Design and Evaluation*, in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pp. 199-210, ACM, April 2011.

[IPython] F Perez and B E Granger, *IPython: A System for Interactive Scientific Computing*, in *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21-29, May/June 2007.

[NumPy] T Oliphant, *Guide to Numpy*, 2006. Available at http://www.tramy.us/.

[SciPy] E Jones, T Oliphant, P Peterson and others, *SciPy: Open Source Scientific Tools for Python*. Available at http://www.scipy.org.

[SimPy] K Müller and T Vignaux, *SimPy: Simulating Systems in Python*, 2003. Available at http://onlamp.com/pub/a/python/2003/02/27/simpy.html.

[Matplotlib] J D Hunter, *Matplotlib: A 2D Graphics Environment*, in *Computing in Science & Engineering*, vol. 9, no. 3. pp. 90-95, 2007.

[MayaVi] P Ramachandran and G Varoquaux, *Mayavi: 3D Visualization of Scientific Data*, in *IEEE Computing in Science & Engineering*, vol. 13, no. 2, pp. 40-51, 2011.

[Cython] R Bradshaw, S Behnel, D S Seljebotn, G Ewing and others, *The Cython compiler*. Available at http://cython.org.

[Ling2011] M J Ling, *IMUSim Tutorial*, Version 0.2, May 2011. Available at http://www.imusim.org/docs/tutorial.html.

[Kuipers2002] J B Kuipers, *Quaternions and Rotation Sequences*, 5th Edition, Princeton University Press, 2002.

[Shoemake1985] K Shoemake, *Animating Rotation with Quaternion Curves*, in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'85)*, pp. 245-254, ACM, 1985.

[Shoemake1991] K Shoemake, *Quaternion Calculus for Animation*, in *Math for SIGGRAPH (ACM SIGGRAPH'91 Course Notes #2)*, 1991.

[Kim1995] M-J Kim, M-S Kim and S Y Shin, *A General Construction Scheme for Unit Quaternion Curves with Simple High Order Derivatives*, in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'95)*, pp. 369-376, ACM, 1995.

[IGRF] National Oceanic and Atmospheric Administration, *Geomagnetic Online Calculator*. Available at http://www.ngdc.noaa.gov/geomagmodels/IGRFWMM.jsp.

[Derby2010] N Derby and S Olbert, *Cylindrical Magnets and Ideal Solenoids*, in *American Journal of Physics*, vol. 78, no. 3, pp. 229-235, March 2010.

[Hemsley2009] R Hemsley, *Interpolation on a Magnetic Field*, Technical Report, Bristol University, September 2009. Available at http://interpolate3d.googlecode.com/files/Report.pdf.

[interpolate3d] R Hemsley, *A Natural Neighbour Interpolation program for 3D data*. Available at http://code.google.com/p/interpolate3d/.

[Orient] A D Young, *Orient Motion Capture*, Available at http://homepages.inf.ed.ac.uk/ayoung9/orient.html

Using Python to Construct a Scalable Parallel Nonlinear Wave Solver

Kyle T. Mandli[¶*], Amal Alghamdi[‡], Aron Ahmadia[‡], David I. Ketcheson[‡], William Scullin[§]

Abstract—Computational scientists seek to provide efficient, easy-to-use tools and frameworks that enable application scientists within a specific discipline to build and/or apply numerical models with up-to-date computing technologies that can be executed on all available computing systems. Although many tools could be useful for groups beyond a specific application, it is often difficult and time consuming to combine existing software, or to adapt it for a more general purpose. Python enables a high-level approach where a general framework can be supplemented with tools written for different fields and in different languages. This is particularly important when a large number of tools are necessary, as is the case for high performance scientific codes. This motivated our development of PetClaw, a scalable distributed-memory solver for time-dependent nonlinear wave propagation, as a case-study for how Python can be used as a high-level framework leveraging a multitude of codes, efficient both in the reuse of code and programmer productivity. We present scaling results for computations on up to four racks of Shaheen, an IBM BlueGene/P supercomputer at King Abdullah University of Science and Technology. One particularly important issue that PetClaw has faced is the overhead associated with dynamic loading leading to catastrophic scaling. We use the walla library to solve the issue which does so by supplanting high-cost filesystem calls with MPI operations at a low enough level that developers may avoid any changes to their codes.

Index Terms—parallel, scaling, finite volume, nonlinear waves, PyClaw, PetClaw, Walla

Introduction

Nowadays, highly efficient, robust, and reliable open source implementations of many numerical algorithms are available to computational scientists. However, different tools needed for a single project may not be available in a single package and may even be implemented in different programming languages. In this case, a common solution is to re-implement the various tools required in yet another software package. This approach is quite expensive in terms of effort, since a completely new code must be written, tested, and debugged. An alternative approach is to bring together the existing software tools by wrapping them in a small code based on abstractions compatible with all of them and able to interface with each programming language involved. The latter approach has the advantage that only a small amount of relatively high-level code needs to be written and debugged; the bulk of the work will still be done by the reliable, tested packages. In this paper, PyClaw and PetClaw are presented as examples of the philosophy that bridging well-established codes with high-level maintainable code is an alternative approach that leads to advantages in usability, extensibility, and maintainability when compared to completely custom built scientific software.

PyClaw and PetClaw are implemented in the Python programming language, and this choice of language has been essential to their success, for multiple reasons. Python is an interpreted scripting language that has become recognized in the scientific computing community as a viable alternative to Matlab, Octave, and other languages that are specialized for scientific work [cai2005]. For instance, Python (with the numpy package) possesses a natural and inuitive syntax for mathematical operations, has a built-in user-friendly interactive debugger, and allows simulation and visualization to be integrated into a single environment. At the same time, Python is a powerful, elegant, and flexible language. Furthermore, there exist many Python packages that make it simple to incorporate code written in C, C++, and Fortran into Python programs. Python has been suggested as particularly useful in enabling reproducible computational research [leveque2009].

PyClaw and PetClaw Design and Implementation

PyClaw and PetClaw are designed to facilitate the implementation of new algorithms and methods in the existing framework established in the well-known software package Clawpack [clawpack]. Clawpack is used to solve linear and nonlinear hyperbolic systems of partial differential equations using a Godunov type method with limiters and is written primarily in Fortran. It has been freely available since 1994 and has more than 7,000 registered users in a large variety of applications. The goal in the design of PyClaw and PetClaw is to provide interfaces to Clawpack that will facilitate the use of advanced parallel strategies, algorithm improvements, and other possible enhancements that may be field specific to the original algorithms available in Clawpack.

PyClaw

PyClaw is based on the principles of abstraction and careful definition of interfaces. These principles have allowed a broad range of extensions to be developed for PyClaw in a short period of time with wide success. The basic building blocks of PyClaw involve the separation of the data structures representing the gridded data and the domain and the solution operators that advance the gridded data to the end-time requested. Both of these abstraction layers also implement accessors that can be overridden to provide advance functionality, a feature used in PetClaw.

* *Corresponding author: mandli@amath.washington.edu*
¶ *University of Washington*
‡ *King Abdullah University of Science and Technology*
§ *Argonne National Labs*

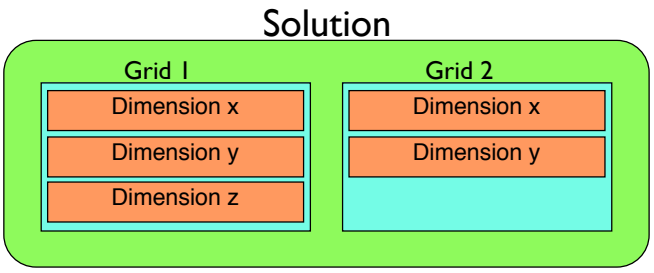

Fig. 1: *PyClaw Solution structure.*

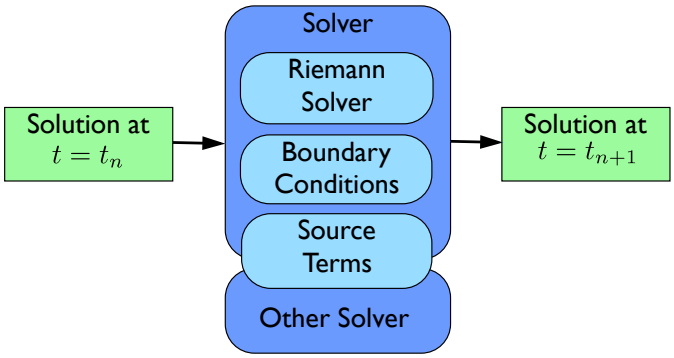

Fig. 2: *PyClaw architecture flow with solver structures.*

The *Solution* class represents a snap-shot of the gridded data at a single instance in time. The class acts as a container object with possibly multiple *Grid* objects in the case of adaptive mesh refinement or nested grids, both of which the Clawpack algorithms are capable of. Furthermore, the *Grid* objects also contain a set of *Dimension* objects that define the domain that each *Grid* is defined on. Using this hierarchical class structure allows the gridded data in PyClaw to not only represent relatively arbitrarily complex gridded data but also allows individual components of the data structures to be sub-classed without the knowledge of the rest of the data structure. This is why the implementation of a package like PetClaw is as transparent as it is to the end-user. An example of a *Solution* object can be seen in figure 1.

The *Solver* class is the numerical realization of a solution operator mapping the initial condition, represented by a *Solution* object, to a later time. The base *Solver* class defines a basic set of interfaces that must be implemented in order for the infrastructure included in PyClaw to evolve the *Solution* object forward in time. For instance, the routine:

```
evolve_to_time(solution,t)
```

will operate on the *Solution* object *solution* and do the necessary operations to evolve it forward in time. This is accomplished through appropriate time stepping in the base *Solver* object and the definition of a *step()* routine that the particular sub-class of *Solver* has implemented. This basic algorithm can be seen in figure 2.

We expect the PyClaw code to be more easily maintainable and extensible than Clawpack, for reasons based on the difference between the Fortran 77 and Python languages [logg2010]. Fortran 77 codes generally require very long lists of arguments to subroutines, because of the lack of dynamic memory allocation and structured data types. This often leads to bugs when a subroutine interface is changed, because it is challenging to ensure that the function call is modified correctly throughout the program. In contrast, Python allows the use of extremely simple argument lists, since it has dynamic memory allocation, is object-oriented, and allows for default argument values to be pre-specified. This difference has already allowed the simple integration of different algorithms into a single framework (PyClaw). The Fortran versions of these programs share only a small fraction of code and require significantly different setup by the user, but in PyClaw switching between them is trivial.

The solvers currently available are the 2nd-order algorithms of Clawpack and the high order algorithms found in SharpClaw [sharpclaw]. Clawpack is based on a Lax-Wendroff approach plus TVD limiters, while SharpClaw is based on a method of lines approach using weighted essentially non-oscillatory (WENO) reconstruction and high order Runge-Kutta methods. The abstract Solver class has been carefully designed to allow these solvers to be swapped trivially, i.e. by using either:

```
solver = pyclaw.ClawSolver2D()
```

for Clawpack, or:

```
solver = pyclaw.SharpClawSolver2D()
```

for SharpClaw. This allows the user to easily compare the performance of different methods.

Another very useful abstraction managed by PyClaw is that of the implementation language. The 1D PyClaw solvers contain a complete implementation of both the Clawpack and SharpClaw algorithms, written entirely in Python. This is useful for rapidly prototyping, debugging, and testing modifications or new options, since new algorithms for hyperbolic PDEs are typically developed in a 1D setting. Since this code is written using *numpy* and vectorization, it is tolerably fast, but still significantly slower than compiled C or Fortran (vectorized *numpy* code is similar in speed to vectorized MATLAB code). For production runs, the user can easily switch to the more efficient wrapped Fortran codes. This is handled simply by setting the *kernel_language* attribute of the *Solver* object to "Python" or "Fortran" (the latter being the default). Even more efficient CUDA implementations of these kernels are in preparation. The benefit of this design is that the user does not need to know multiple programming languages in order to take advantage of different implementations.

PetClaw

Nilsen et. al. have suggested Python as a good high-level language for use in parallelization of scientific codes because it allows for extensive reuse of serial code and little effort (related to parallelism) from the end user [nilsen2010].

PetClaw is designed to use PETSc to add parallel functionality to PyClaw with both of these objectives in mind. This means that the (serial) PyClaw code should not need modification to accommodate PetClaw extensions and that within PetClaw all parallel operations should be handled by PETSc data structures in a way that is transparent to the user. Python makes both of these goals achievable within an elegant framework.

By implementing all necessary parallel code in Python, Nilsen demonstrated approximately 90% parallel efficiency for various applications on up to 50 processors. Because we need to go three orders of magnitude further in parallel scaling, PetClaw design goes beyond the approach suggested in [nilsen2010] and related works, by handing off all parallel operations to a widely used, robust library (PETSc) written in a compiled language. Because

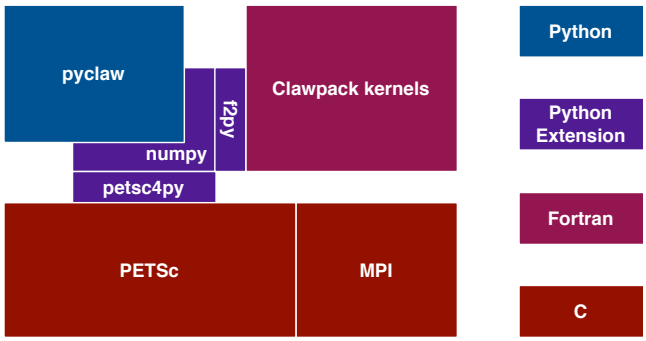

Fig. 3: *Modular structure of the PetClaw code, with a focus on the orthogonality of the Fortran kernels from the parallel decomposition through PETSc.*

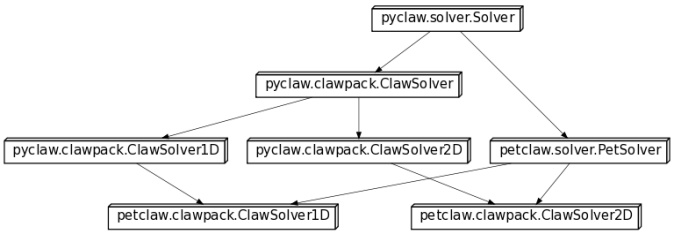

Fig. 4: *Class inheritance diagram for PetClaw solvers.*

PETSc is very actively developed and maintained for use in many scientific codes and on many hardware platforms, this also means that PetClaw developers don't have to worry about portability or maintenance of the parallel routines and can instead focus on the numerical algorithms that are particular to PetClaw.

An even more significant advantage gained by interfacing with PETSc that may be leveraged in the future is access to a variety of efficient parallel solvers.

Python language features and multiple inheritance have been used to make running parallel simulations with PetClaw very simple. The only difference between a serial PyClaw script and a PetClaw script that runs on tens of thousands of cores is exchanging:

```
import pyclaw
```

for:

```
import petclaw as pyclaw
```

Arrays for the solution and for coefficients that vary in space are represented by numpy arrays in PyClaw but by a custom distributed *Vec* class in PETSc. Using the *property* Python language feature, this difference is completely transparent to the user. Parallel solver classes are implemented via multiple inheritance; in most cases, a parallel solver is created merely by subclassing the corresponding serial solver as well as a base parallel solver class *PetSolver*; no further attributes or methods need to be implemented. As a result, the entire PetClaw extension consists of less than 300 lines of code.

Figre 4 shows how serial and parallel functionality, as well as algorithmic and dimensional differences, are implemented in an orthogonal way using class inheritance.

Software Engineering

One of the potential indirect benefits of developing a code in Python is exposure to the generally high level of software engineering practices maintained by the Python community. Primarily as a result of this exposure, PyClaw includes a suite of regression tests that currently cover 57% of the code and are being expanded. The Python package *nose* is used to easily run the tests or any desired subset of them. Code development is coordinated using the distributed version control software Git and the code hosting website Github. The project has an active issue tracker where bugs are reported and new features are suggested, as well as an online forum (petclaw-dev@googlegroups.com) where more detailed discussions take place. Finally, online documentation including both reference material and tutorials is maintained using the Python package *Sphinx*, which allows, among other things, for mathematical expressions to be included in inline code documentation and automatically rendered using *LaTeX* when viewed online. While many of these practices and features would be taken for granted in industrial codes, they are not standard in academic scientific codes [wilson2006].

2D Performance Results

For PetClaw performance assessment with 2D problems, we have conducted on-core serial experiments to compare the performance of PetClaw code with the corresponding pure Fortran code, Clawpack. We have also performed weak scaling experiments to study the scalability of PetClaw on up to four racks of the Shaheen system. Corresponding results for PetClaw simulations in 1D may be found in [petclaw11].

On-Core Performance

We consider two systems of equations in our serial performance tests. The first is the system of 2D linear acoustics and the second is the 2D shallow water (SW) equations. The acoustics test involves a very simple Riemann solver and is intended to highlight any performance difficulties arising from the Python code overhead. The shallow water test involves a more typical, costly Riemann solver (specifically, a Roe solver with entropy fix) and should be considered as more representative of realistic nonlinear application problems.

Table 1 shows an on-core serial comparison between the Fortran-only Clawpack code and the corresponding hybrid PetClaw implementation for two systems of equations in two different platforms. Both codes rely on similar Fortran kernels that differ only in the array layout. The tests on the first platform were compiled for the x86_64 instruction set using gfortran 4.5.1 (4.5.1 20100506 (prerelease)). Each result was timed on a single core of a Quad-Core Intel Xeon 2.66GhZ Mac Pro workstation equipped with 8x2 GB 1066MHz DDR3 RAM. The same tests were conducted on Shaheen, on a single core of a Quad-Core PowerPC 450 processor with 4GB of available RAM. IBM XLF 11.1 Fortran compiler was used to produce a PowerPC 450d binray code in the latter platform. On both platforms, the compiler optimization flag -O3 was set. Because most of the computational cost is in executing the low-level Fortran kernels, the difference in performance is relatively minor with the difference owing primarily to the Python overhead in PetClaw. Interestingly, while the relative acoustics performance between the two codes was similar for both versions of gfortran, a significant difference was observed in the relative performance of the codes on the shallow water example, depending on the compiler version.

	Processor	Clawpack	PetClaw	Ratio
Acoustics	Intel Xeon	28s	41s	1.5
Shallow Water	Intel Xeon	79s	99s	1.3
Acoustics	PowerPC 450	192s	316s	1.6
Shallow Water	PowerPC 450	714s	800s	1.1

TABLE 1: *Timing results in seconds for on-core serial experiment of an acoustics and shallow water problems implemented in both Clawpack and PetClaw for Intel Xeon and PowerPC 450 machines.*

Parallel Performance

In our parallel performance tests, we consider the same acoustics 2D linear system used in the serial runs to represent an application where the communication over computation ratio can be relatively high due to the simplicity of its Riemann solver. We also tested 2D Euler equations of compressible fluid dynamics as a more realistic nonlinear application problem that has a relatively expensive Riemann solver.

Table 2 shows the execution time for both experiments as the number of cores increases from one core up to 16 thousand cores (four racks of BlueGene/P), with the ratio of work per core fixed. The acoustics problem used involves 178 time steps on a square grid with 160,000 (400x400) grid cells per core. The Euler problem used involves 67 time steps on a grid also with 160,000 grid cells per core. The first column for each test indicates the simulation time excluding the load time required to import Python modules. The second column indicates the total simulation time, including Python module imports.

Excellent scaling is observed for both tests, apart from the dynamic loading. Profiling of the acoustics example shows that the small loss of efficiency is primarily due to the communication of the CFL number, which requires a max global reduce operation that is done each time step, and also partly due to the communication of ghost cell values between adjacent domains at each time step.

In contrast, the total job time reveals the very poor scaling of the dynamic loading time. For the largest jobs considered, this load time is roughly one hour, which is significant though generally not excessive relative to typical simulation times, since the CFL condition means that large simulations of hyperbolic problems necessarily require long run times in order for waves to propagate across the full domain. Nevertheless, this inefficiency remains as a disadvantage for high performance Python codes. Although much longer simulations can to some extent justify the start up time required for dynamic loading of Python, this loading time severely impacts parallel scaling, motivating the development of Walla to address this challenge.

Addressing the Catastrophic Loading Problem with Walla

Catastrophic scaling has been observed in applications written in all languages when they perform dynamic linking and loading on large distributed systems. Python applications are particularly prone to poor scaling due to systems issues as they tend to strongly exercise dynamic linking and loading. At the same time, Python applications provide excellent models for examining possible solutions to catastrophic dynamic link and load times [pynamic2007].

Python applications are particularly prone to poor scaling due to system overheads. They generally exercise the sort of dynamic

	Acoustics		Euler	
Cores No.	Evolve Solution	Total	Evolve Solution	Total
1	76.7	154	98.9	124
4	69	152	101.1	123
16	71.7	164	103.2	142
64	73.7	217	103.0	184
256	74	407	103.4	465
1024	75	480	103.9	473
4096	76.6	898	104.9	953
16384	79.6	3707	112.9	3616

TABLE 2: *Timing results in seconds from scaling comparisons of the acoustics and Euler test problems for the time required for evolving the solution and the communication between processes. The total time includes the overhead due to the dynamic loading in Python and reveals the catastrophic dynamic loading problem.*

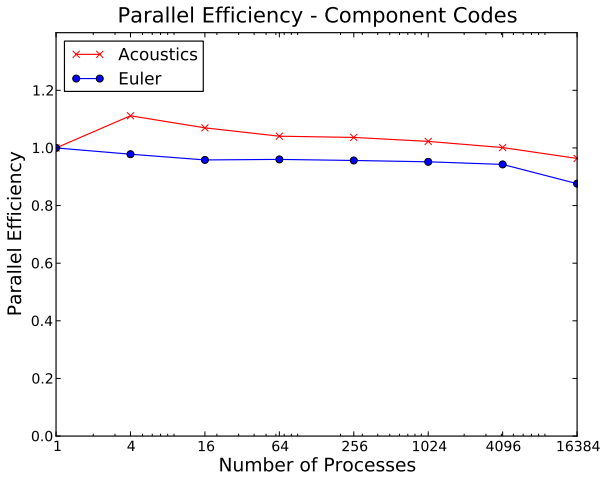

Fig. 5: *Parallel efficiency results of a 2D acoustics problem and a 2D Euler problem for evolving the solution to the final time. These times does not include the dynamic load time of Python.*

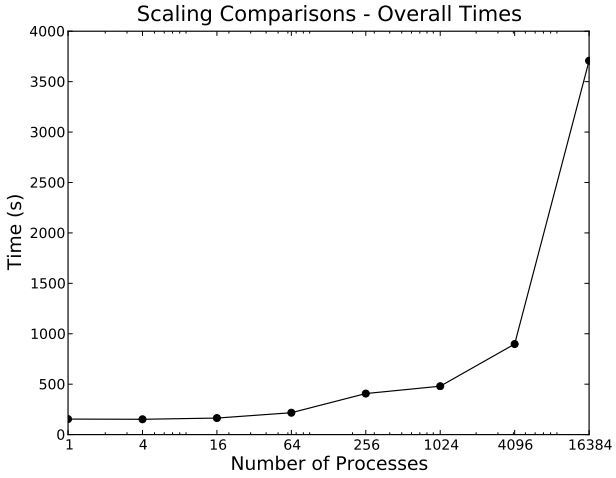

Fig. 6: *Weak scaling results of the previous acoustics run from figure 5 but including the entire time to completion.*

linking and loading that creates contention for file data and metadata. In general, the farther you scale, the worse the impact on application load times becomes. This problem is well understood and benchmarks, such as in Lawrence Livermore National Laboratory's Pynamic, which help to describe and understand the extent to which an application may be impacted on a particular system [pynamic2007]. Conversely, Python applications can highlight the deficits and make it an apt platform to explore solutions.

The CPython interpreter's process for importing modules is very I/O and metadata intensive. If dynamically linked, the overhead of loading a module is further increased as Python must work through the operating system software stack before the interpreter may continue. This process is generally ignored by Python developers as single system file I/O performance is reasonable compared to the costs of computation. In large distributed systems used for scientific computation, the problem is turned on its head with file I/O, unless parallel file I/O is available, presenting a fairly substantial bottleneck. Even where parallel I/O is available, the emphasis has been on the reading and writing of application data in a way that optimizes for file system bandwidth, generally favoring large reads and writes.

Walla's Approach

The Walla project attempts to take advantage of the high speed interconnects normally used for interprocess communication to speed dynamic loading without alteration of user codes. The project originated on IBM's Blue Gene/P platform where load times at 8,192 nodes exceeded 45 minutes for a large Python code called GPAW. Initial efforts were focused on using the low-level interface to the Blue Gene/P's high performance networks with the goal of being able to use Walla to speed all aspects of loading by coming in before the loading of MPI libraries. Due to community interest and feedback, the original codebase was abandoned in favor of using MPI for all communications ensuring portability between systems and eliminating any licensing restrictions created by use of vendor code.

In the Walla design, the CPython importer and the glibc libdl are replaced with versions that have been modified such that only a single rank performs metadata intensive tasks and file system I/O. Modifications are generally kept to a minimum with *fopen* and *stat* being replaced with versions that rely on MPI rank 0 to perform the actual *fopen* and *stat* calls, then broadcast the result to all other nodes. While wasteful of memory, the glibc *fmemopen* function is used to produce a file handle returned by the *fopen* replacement. At no time do nodes other than MPI rank 0 access Python modules or libraries via the filesystem, eliminating much of the overhead and contention that is caused by large number of ranks attempting to perform loads simultaneously.

There are a handful of caveats to using Walla. First, users must be in a situation where I/O is more expensive than broad- cast operations. While initial numbers show no significant performance hit from using Walla at small node counts, this is not guaranteed. Second, MPI_Init must already be called at the time Walla is first invoked. As Walla relies on MPI, it cannot be used to load MPI itself. The file handle generated by *fmemopen* does not contain and cannot be used to generate a file descriptor as the file handle is created in user space and file descriptors require the allocation of resources by the kernel. While the the handle is sufficient for use with most codes, this does create compatibility issues when an application contains calls expecting a file descriptor. Finally, some thought has to be given to the bandwidth available through I/O networks versus the MPI broadcast otherwise it becomes easy to replace one slow loading interface with another.

Despite the need for substantial reengineering of the CPython importer internals, almost all changes should eventually be transparent to end users and require no changes to user Python codes. The runtime environment requires changes to the *site.py* to ensure the loading of MPI and replace the native importer with the Walla importer. For compatibility reasons, *libdl* is not completely replaced; users should link *libwalla* before the glibc *libdl* to ensure that the symbols for *dlopen*, *dlsym*, and *dlclose* resolve back to *libwalla* rather than *libdl*.

Blue Gene/P Implementation

The Blue Gene/P platform presents additional difficulties due to I/O shipping since Blue Gene/P nodes have no local storage. At boot, operating system images get broadcasted directly into a node's memory with I/O nodes receiving a lightweight version of Linux that mounts remote file systems over a 10 Gigabit Ethernet link to a site's storage infrastructure. All Blue Gene/P nodes have three bidirectional 850MBps connections to a collective network designed for one-to-all high-bandwidth communications. When a compute node performs an I/O function, the operation is shipped to the I/O node via a collective network link, then processed on the I/O node, and the result returned to the compute node.

While metadata operations are easily reduced and eliminated with Walla on the Blue Gene, developers need to watch for calls that would remain local under Linux, but will be shipped on the Blue Gene such as read, seek, and close operations despite being pointed at a local memory buffer. This leads to the need to eliminate or replace read and write calls in code loading libraries and importing modules with code that directly maps or executes the contents of the broadcasted buffers. Eliminating any trace of function shipping has been a major focus of reworking the CPython importer on the Blue Gene/P platform.

Conclusions and Future Directions

We have described and demonstrated an example of using Python as the means to bind and extend the well-established hyperbolic PDE code Clawpack. The serial and parallel performance of the resulting codes are remarkable given the relatively small amount of coding (300 lines) required to turn a serial Fortran code into a scalable parallel one. This is much preferable to the alternative, more traditional approach of ex- tending legacy codes directly for high-performance computing applications using hand-coded APIs, which would be more time-consuming and more difficult to maintain.

One of the drawbacks to the approach proposed is the contention that can be caused by dynamic loading stresses on many high-performance systems. The approach introduced by Walla is a promising answer to this problem and preliminary results suggest that it may be a solution for Python codes suffering from poor scalability on distributed systems.

REFERENCES

[petclaw11] Amal Alghamdi, Aron Ahmadia, David I. Ketcheson, Matthew G. Knepley, Kyle T. Mandli, and Lisandro Dalcin. *PetClaw: A scalable parallel nonlinear wave propagation solver for Python* Proceedings of the High Performance Computing Symposium 2011 (2011).

[clawpack] Randall J. LeVeque, Marsha J. Berger, et. al., Clawpack Software 4.6.1, www.clawpack.org, 15 June, 2011.

[sharpclaw] David I. Ketcheson, Matteo Parsani, and Randall J. LeVeque. *High-order wave propagation algorithms for general hyperbolic systems* (submitted).

[nilsen2010] J. K. Nilsen, X. Cai, B. Hoyland, and H. P. Langtangen (2010). *Simplifying the parallelization of scientific codes by a function-centric approach in Python*. Computational Science & Discovery, 3, 015003.

[cai2005] Xing Cai, H. P. Langtangen, and H. Moe. *On the performance of the Python programming language for serial and parallel scientific computations* Scientific Programming, 13(1):31-56 (2005).

[leveque2009] R. J. LeVeque, *Python Tools for Reproducible Research on Hyperbolic Problems* Computing in Science & Engineering, 11(1): 19-27 (2009).

[wilson2006] G. Wilson, *Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive* Computing in Science & Engineering, 2006;8(6):66-69.

[logg2010] A. Logg, H. P. Langtangen, and X. Cai, *Past and Future Perspectives on Scientific Software.* In: Tveito A, Bruaset AM, Lysne O, eds. Simula Research Laboratory. Springer Berlin Heidelberg; 2010:321-362.

[pynamic2007] Gregory L. Lee, Dong H. Ahn, Bronis R. de Supinski, John Gyllenhaal, Patrick Miller, *Pynamic: the Python Dynamic Benchmark* IEEE Workload Characterization Symposium, pp. 101-106, 2007 IEEE 10th International Symposium on Workload Characterization, 2007.

Building a Framework for Predictive Science

Michael M. McKerns[‡*], Leif Strand[‡], Tim Sullivan[‡], Alta Fang[‡], Michael A.G. Aivazis[‡]

Abstract—Key questions that scientists and engineers typically want to address can be formulated in terms of predictive science. Questions such as: "How well does my computational model represent reality?", "What are the most important parameters in the problem?", and "What is the best next experiment to perform?" are fundamental in solving scientific problems. `mystic` is a framework for massively-parallel optimization and rigorous sensitivity analysis that enables these motivating questions to be addressed quantitatively as global optimization problems. Often realistic physics, engineering, and materials models may have hundreds of input parameters, hundreds of constraints, and may require execution times of seconds or longer. In more extreme cases, realistic models may be multi-scale, and require the use of high-performance computing clusters for their evaluation. Predictive calculations, formulated as a global optimization over a potential surface in design parameter space, may require an already prohibitively large simulation to be performed hundreds, if not thousands, of times. The need to prepare, schedule, and monitor thousands of model evaluations, and dynamically explore and analyze results, is a challenging problem that requires a software infrastructure capable of distributing and managing computations on large-scale heterogeneous resources. In this paper, we present the design behind an optimization framework, and also a framework for heterogeneous computing, that when utilized together, can make computationally intractable sensitivity and optimization problems much more tractable. The optimization framework provides global search algorithms that have been extended to parallel, where evaluations of the model can be distributed to appropriate large-scale resources, while the optimizer centrally manages their interactions and navigates the objective function. New methods have been developed for imposing and solving constraints that aid in reducing the size and complexity of the optimization problem. Additionally, new algorithms have been developed that launch multiple optimizers in parallel, thus allowing highly efficient local search algorithms to provide fast global optimization. In this way, parallelism in optimization also can allow us to not only find global minima, but to simultaneously find all local minima and transition points -- thus providing a much more efficient means of mapping out a potential energy surface.

Index Terms—predictive science, optimization, uncertainty quantification, verification, validation, sensitivity analysis, parallel computing, distributed computing, heterogeneous computing

Introduction

Recently, a unified mathematical framework for the rigorous construction and solution of uncertainty quantification (UQ) problems was formulated [OSS11]. This framework, called Optimal Uncertainty Quantification (OUQ), is based on the observation that, given a set of assumptions and information about the problem, there exist optimal bounds on the uncertainties. These bounds are obtained as extreme values of well-defined optimization problems that correspond to extremizing probabilities of failure subject to the constraints imposed by scenarios compatible with the information set.

An accompanying software framework that implements these rigorous UQ/OUQ methods is now posed.

A rigorous quantification of uncertainty can easily require several thousands of model evaluations $f(x)$. For all but the smallest of models, this requires significant clock time -- a model requiring 1 minute of clock time evaluated 10,000 times in a global optimization will take 10,000 minutes ($\sim$ 7 days) with a standard optimizer. Furthermore, realistic models are often high-dimensional, highly-constrained, and may require several hours to days even when run on a parallel computer cluster. For studies of this size or larger to be feasible, a fundamental shift in how we build optimization algorithms is required. The need to provide support for parallel and distributed computing at the lowest level -- within the optimization algorithm -- is clear. Standard optimization algorithms must be extended to parallel. The need for new massively-parallel optimization algorithms is also clear. If these parallel optimizers are not also seamlessly extensible to distributed and heterogeneous computing, then the scope of problems that can be addressed will be severely limited.

While several robust optimization packages exist [JOP01], [KROOO], there are very few that provide massively-parallel optimization [BMM10], [EKL02], [MAT09] -- the most notable effort being DAKOTA [DAKOT], which also includes methods for uncertainty quantification [DAKUQ]. A rethinking of optimization algorithms, from the ground up, is required to dramatically lower the barrier to massively-parallel optimization and rigorous uncertainty quantification. The construction and tight integration of a framework for heterogeneous parallel computing is required to support such optimizations on realistic models. The goal should be to enable widespread availablility of these tools to scientists and engineers in all fields.

Several of the component pieces of such a framework for predictive science already exist, while a few key pieces must be constructed -- furthermore, these packages must then be assembled and integrated. Python [GVRPY] is a natural integration environment, and is one that readily supports the dynamic nature of working across heterogeneous resources. By requiring this framework be pure-Python, many of the barriers to running on a new platform are removed. `multiprocessing` [MPROC], `mpi4py` [MPI4P], and `pp` [VVPPP] are selected for communication mechanisms, both due to their high level of feature coverage and their relative ease of installation. NumPy [NUMPY] is used for algorithmic efficiency, and SymPy [SYM11] is used to provide an alternate interface for building constraints. Many of

* *Corresponding author: mmckerns@caltech.edu*
‡ *California Institute of Technology*

the optimization algorithms leverage SciPy [JOP01]; however like the use of Matplotlib [MATPL] for plotting, SciPy is an optional dependency.

This paper will discuss the modifications to the `mystic` [MHA09] optimization framework required to provide a simple interface to massively parallel optimization, and also to the `pathos` [MBA10] framework for staging and launching optimizations on heterogeneous resources. These efforts leverage `pyre` [MAGA1] -- an component integration framework for parallel computing, which has recently been extended to distributed communication and management with `hydra` (part of this development effort). This paper will also overview a new mathematical framework [OSS11], [ALL11], [KLL11], [LOO08] for the quantification of uncertainties, which provides a formulation of UQ problems as global optimization problems.

Rigorous Uncertainty Quantification

Following [LOO08], we specifically take a *certification* point of view of uncertainty quantification. For definiteness, we consider systems whose operation can be described in terms of N scalar performance measures $(Y_1,\ldots,Y_N) = Y \in \mathbb{R}^N$. The response of the system is taken as *stochastic* due to the intristic randomness of the system, or randomness in the input parameters defining the operation of the system, or both. Suppose that the outcome $Y \in A$ constitutes a satisfactory outcome for the system of interest, for some prescribed measureable *admissible* set $A \subseteq \mathbb{R}^N$. Hence, we are interested in determining the *probability of failure* (PoF) $\mathbb{P}[Y \in A^c]$.

Evidently, for an upper bound to be useful, it must also be *tight* (i.e. it must be close to the actual PoF of the system) and accessible by some combination of laboratory and computational means. In [ALL11], [KLL11], a methodology for a rigorous determination of tight upper bounds on the probability of failure for complex systems is presented, and is summarized below.

We consider a response function $Y = F(X)$ that maps controllable system inputs X to performance measures Y, and relies on a probability of failure (PoF) upper bounds of the concentration of measure (CoM) type [BBL04], [LED01], [MCD89]. If McDiarmid's inequality [MCD89] (i.e. the bounded differences inequality) is used to bound PoF, the system may then be certified on the sole knowledge of ranges of its input parameters -- without *a priori* knowledge of their probability distributions, its mean performance $\mathbb{E}[Y] = M$ and a certain measure $D_G = U$ of the spread of the response, known as *system diameter*, which provides a rigorous quantitative measure of the uncertainty in the response of the system.

A model is regarded as $Y = F(X)$ that approximates the response $Y = G(X)$ of the system. An upper bound on the system diameter -- and thus on the uncertainty in the response of the system -- then follows from the triangle inequality $D_G \leq D_F + D_{G-F}$, and $U = D_F + D_{G-F}$ can be taken as a new -- and conservative -- measure of system uncertainty. In this approach, the total uncertainty of the system is the sum of the *predicted uncertainty* (i.e. the variability in performance predicted by the model as quantified by the *model diameter* D_F), and the *modeling-error uncertainty* (i.e. the discrepancy between model prediction and experiment as quantified by the *modeling-error diameter* D_{G-F}.

In [LOO08], PoF upper bounds of the CoM type were formulated by recourse to McDiarmid's inequality. In its simplest version, this inequality pertains to a system characterized by N real random inputs $X = (X_1,\ldots,X_N) \in E \subseteq \mathbb{R}^N$ and a single real performance measure $Y \in \mathbb{R}$. Suppose that the function $G : \mathbb{R}^N \to \mathbb{R}$ describes the response function of the system. Suppose that the system fails when $Y \leq a$, where a is a threshold for the safe operation of the system. Then, a direct application of McDiarmid's inequality gives the following upper bound on the PoF of the system:

$$\mathbb{P}[G \leq a] \leq \exp\left(-2\frac{M^2}{U^2}\right) \tag{1}$$

where

$$M = (\mathbb{E}[G] - a)_+ \tag{2}$$

is the *design margin* and

$$U = D_G \tag{3}$$

is the *system uncertainty*. In (3), D_G is the diameter of the response function. From (1) it follows that the system is certified if

$$\exp\left(-2\frac{M^2}{U^2}\right) \leq \varepsilon$$

where ε is the PoF tolerance, or, equivalently, if

$$\mathrm{CF} = \frac{M}{U} \geq \sqrt{\log\sqrt{\frac{1}{\varepsilon}}} \tag{4}$$

where CF is the *confidence factor*. In writing (2) and subsequently, we use the function $x_+ := \max(0,x)$. We see from the preceding expressions that McDiarmid's inequality supplies rigorous quantitative definitions of design margin and system uncertainty. In particular, the latter is measured by *system diameter* D_G, which measures the largest deviation in performance resulting from arbitrarily large perturbations of one input parameter at a time. Within this simple framework, rigorous certification is achieved by the determination of two--and only two--quantities: the *mean performance* $\mathbb{E}[G]$ and the *system diameter* D_G.

McDiarmid's inequality is a result in probability theory that provides an upper bound on the probability that the value of a function depending on multiple independent random variables deviates from its expected value. A central device in McDiarmid's inequality is the *diameter* of a function. We begin by recalling that the *oscillation* $\mathrm{osc}(f,E)$ of a real function $f : E \to \mathbb{R}$ over a set $E \in R$ is

$$\mathrm{osc}(f,E) = \sup\{|f(y) - f(x)| : x,y \in E\} \tag{5}$$

Thus, $\mathrm{osc}(f,E)$ measures the spread of values of f that may be obtained by allowing the independent variables to range over its entire domain of definition. For functions $f : E \subset \mathbb{R}^N \to \mathbb{R}$ of several real values, component-wise *suboscillations* can be defined as

$$\mathrm{osc}_i(f,E) = \sup\{|f(y) - f(x)| : x,y \in E,\ x_j = y_j \text{ for } j \neq i\} \tag{6}$$

Thus $\mathrm{osc}_i(f,E)$ measures the maximum oscillation among all one-dimensional fibers in the direction of the ith coordinate. The *diameter* $D(f,E)$ of the function $f : E \to \mathbb{R}$ is obtained as the root-mean square of its component-wise suboscillations:

$$D(f,E) = \left(\sum_{i=1}^{n} \mathrm{osc}_i^2(f,E)\right)^{1/2} \tag{7}$$

and it provides a measure of the spread of the range of the function. Thus (6) also us to regard $\mathrm{osc}_i(f,E)$ as a *subdiameter* of the system corresponding to variable X_i, where the subdiameter can be

regarded as a measure of uncertainty contributed by the variable X_i to the total uncertainty of the system.

The attractiveness of the McDiarmid CoM approach to UQ relies on the requirement of tractable information on response functions (sub-diameters) and measures (independence and mean response). Above, it is described how to "plug" this information into McDiarmid's concentration inequality to obtain an upper bound on probabilies of deviation. One may wonder if it is possible to obtain an "optimal" concentration inequality, especially when the available information may not necessarily be sub-diameters and mean values. A general mathematical framework for optimally quantifying uncertainties based only on available information has been proposed [OSS11], and will be summarized here. Assume, for instance, that one wants to certify that

$$\mathbb{P}[G \geq a] \leq \varepsilon \tag{8}$$

based on the information that $\mathrm{osc}_i(G,E) \leq D_i$, $X = (X_1, \ldots, X_N)$, $\mathbb{E}[G] \leq 0$ and that the inputs X_i are independent under $\mathbb{P}$. In this situation, the optimal upper bound $\mathscr{U}(\mathscr{A}_{MD})$ on the PoF $\mathbb{P}[G \geq a]$ is the solution of the following optimization problem

$$\mathscr{U}(\mathscr{A}_{MD}) = \sup_{(f,\mu)\in\mathscr{A}_{MD}} \mu[f(X) \geq a] \tag{9}$$

subject to constraints provied by the information set

$$\mathscr{A}_{MD} = \left\{ (f,\mu) \;\middle|\; \begin{array}{c} f : E_1 \times \cdots \times E_N \to \mathbb{R}, \\ \mu \in \mathscr{M}(E_1) \otimes \cdots \otimes \mathscr{M}(E_N), \\ \mathbb{E}_\mu[f] \leq 0, \\ \mathrm{osc}_i(f,E) \leq D_i \end{array} \right\} \tag{10}$$

where $\mathscr{M}(E_k)$ denotes the set of measures of probability on E_k. Hence, McDiarmid's inequality is the statement that

$$\mathscr{U}(\mathscr{A}_{MD}) \leq \exp\left(-2\frac{a^2}{\sum_{i=1}^N D_i^2}\right) \tag{11}$$

Similarly, for any other set of information $\mathscr{A}$, we have an optimal (i.e.) least upper bound on the probability of deviation

$$\mathscr{U}(\mathscr{A}) = \sup_{(f,\mu)\in\mathscr{A}} \mu[f(X) \geq a] \tag{12}$$

The idea is that in practical applications, the available information does not determine $(G,\mathbb{P})$ uniquely, but does determine a set $\mathscr{A}$ such that $(G,\mathbb{P}) \in \mathscr{A}$ and such that any $(f,\mu) \in \mathscr{A}$ could *a priori* be $(G,\mathbb{P})$. This mathematical framework, called optimal uncertainty quantification (OUQ), is based on the observation that, given a set of assumptions and information about the problem, there exist optimal bounds on uncertainties; these are obtained as extreme values of well-defined optimization problems corresponding to extremizing probabilities of failure, or of deviations, over the feasible set $\mathscr{A}$. Observe that this framework does not implicitly impose inappropriate assumptions, nor does it repudiate relevant information. Indeed, as demonstrated in (10 and 11) for the CoM approach, OUQ can pose a problem that incorporates the assumptions utilized in other common UQ methods (such as Bayesian inference [LJH99]) and provide a rigorous optimal bound on the uncertainties.

Although some OUQ problems can be solved analytically, most must be solved numerically. To that end, the reduction theorems of [OSS11] reduce the infinite-dimensional feasible set $\mathscr{A}$ to a finite-dimensional subset $\mathscr{A}_\Delta$ that has the key property that the objective function (PoF) has the same lower and upper extreme values over $\mathscr{A}_\Delta$ as over $\mathscr{A}$.

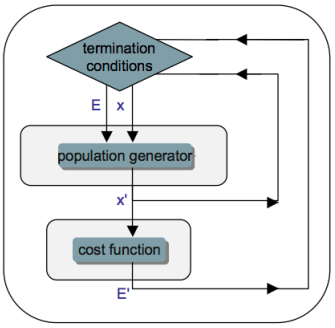

Fig. 1: *Conceptual diagram for an optimizer. The cost function provides a difference metric that accepts input parameters x and produces a cost E.*

For example, the reduction for $\mathscr{A}_{MD}$ in (10) is to pass to measures $\mu = \mu_1 \otimes \cdots \otimes \mu_N$ such that each marginal measure μ_i is supported on at most two points of the parameter space E_i, i.e. μ_i is a convex combination of two Dirac measures (point masses). Having reduced the set of feasible measures μ, the set of feasible response functions f is also reduced, since we only care about the values of f on the finite support of μ and nowhere else.

We refer the reader to [OSS11] for the more general reduction theorems. The essential point is that if the information/constraints take the form of n_i inequalities of the form $\mathbb{E}_{\mu_i}[\phi_j] \leq 0$ (for some test functions ϕ_j) and n' inequalities of the form $\mathbb{E}_\mu[\phi_j] \leq 0$, then it is enough to consider μ_i with support on $1 + n_i + n'$ points of E_i.

The reduction theorems leave us with a finite-dimensional optimization problem in which the optimization variables are suitable parametrizations of the *reduced* feasible scenarios (f,μ).

A Highly-Configurable Optimization Framework

We have built a robust optimization framework (`mystic`) [MHA09] that incorporates the mathematical framework described in [OSS11], and have provided an interface to prediction, certification, and validation as a framework service. The `mystic` framework provides a collection of optimization algorithms and tools that lowers the barrier to solving complex optimization problems. `mystic` provides a selection of optimizers, both global and local, including several gradient solvers. A unique and powerful feature of the framework is the ability to apply and configure solver-independent termination conditions --- a capability that greatly increases the flexibility for numerically solving problems with non-standard convergence profiles. All of `mystic`'s solvers conform to a solver API, thus also have common method calls to configure and launch an optimization job. This allows any of `mystic`'s solvers to be easily swapped without the user having to write any new code.

The minimal solver interface:

```
# the function to be minimized and the initial values
from mystic.models import rosen as my_model
x0 = [0.8, 1.2, 0.7]

# configure the solver and obtain the solution
from mystic.solvers import fmin
solution = fmin(my_model, x0)
```

The criteria for when and how an optimization terminates are of paramount importance in traversing a function's potential well. Standard optimization packages provide a single convergence condition for each optimizer. `mystic` provides a set of fully

customizable termination conditions, allowing the user to discover how to better navigate the optimizer through difficult terrain. Optimizers can be further configured through several methods for choosing the `InitialPoints`.

The expanded solver interface:

```
# the function to be minimized and initial values
from mystic.models import rosen as my_model
x0 = [0.8, 1.2, 0.7]

# get monitor and termination condition objects
from mystic.monitors import Monitor, VerboseMonitor
stepmon = VerboseMonitor(5)
evalmon = Monitor()
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# instantiate and configure the solver
from mystic.solvers import NelderMeadSimplexSolver
solver = NelderMeadSimplexSolver(len(x0))
solver.SetInitialPoints(x0)
solver.SetGenerationMonitor(stepmon)
solver.SetEvaluationMonitor(evalmon)
solver.Solve(my_model, COG)

# obtain the solution
solution = solver.bestSolution

# obtain diagnostic information
function_evals = solver.evaluations
iterations = solver.generations
cost = solver.bestEnergy

# modify the solver configuration, and continue
COG = ChangeOverGeneration(tolerance=1e-8)
solver.Solve(my_model, COG)

# obtain the new solution
solution = solver.bestSolution
```

`mystic` provides progress monitors that can be attached to an optimizer to track progress of the fitted parameters and the value of the cost function. Additionally, monitors can be customized to track the function gradient or other progress metrics. Monitors can also be configured to record either function evaluations or optimization iterations (i.e. *generations*). For example, using `VerboseMonitor(5)` in the `SetGenerationMonitor` method will print the `bestEnergy` to `stdout` every five generations.

Constraints Toolkit

`mystic` provides a method to constrain optimization to be within an N-dimensional box on input space, and also a method to impose user-defined parameter constraint functions on any cost function. Thus, both *bounds constraints* and *parameter constraints* can be generically applied to any of `mystic`'s unconstrained optimization algorithms. Traditionally, constrained optimization problems tend to be solved iteratively, where a penalty is applied to candidate solutions that violate the constraints. Decoupling the solving of constraints from the optimization problem can greatly increase the efficiency in solving highly-constrained nonlinear problems -- effectively, the optimization algorithm only selects points that satisfy the constraints. Constraints can be solved numerically or algebraically, where the solving of constraints can itself be cast as an optimization problem. Constraints can also be dynamically applied, thus altering an optimization in progress.

Penalty-based methods indirectly modify the candidate solution by applying a change in energy $\Delta E = k \cdot p(\vec{x})$ to the

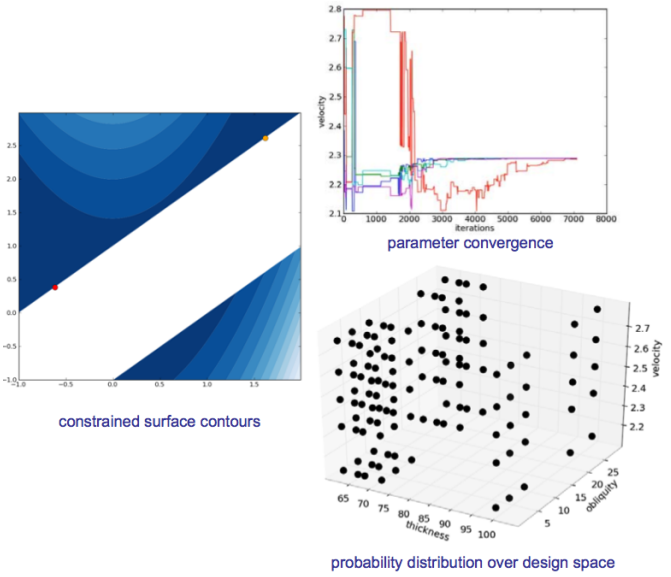

Fig. 2: *Optimization analysis viewers available in* `mystic`.

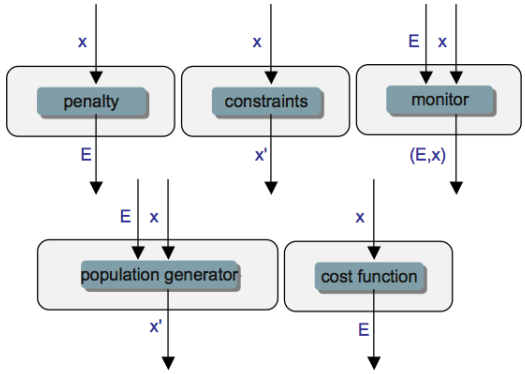

Fig. 3: *Basic components provided in the optimizer toolkit. Several wrapper classes are also provided for binding components, while factory classes are provided for generating components.*

unconstrained cost function $f(\vec{x})$ when the constraints are violated. The modified cost function ϕ is thus written as:

$$\phi(\vec{x}) = f(\vec{x}) + k \cdot p(\vec{x}) \tag{13}$$

Set-based methods directly modify the candidate solution by applying a constraints solver c that ensures the optimizer will always select from a set of candidates that satisfy the constraints. The constraints solver has an interface $\vec{x}' = c(\vec{x})$, and the cost function becomes:

$$\phi(\vec{x}) = f(c(\vec{x})) \tag{14}$$

Adding parameter constraints to a solver is as simple as building a constraints function, and using the `SetConstraints` method. Additionally, simple bounds constraints can also be applied through the `SetStrictRanges` method:

```
# a user-provided constraints function
def constrain(x):
  x[1] = x[0]
  return x

# the function to be minimized and the bounds
from mystic.models import rosen as my_model
lb = [0.0, 0.0, 0.0]
```

```
ub = [2.0, 2.0, 2.0]

# get termination condition object
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# instantiate and configure the solver
from mystic.solvers import NelderMeadSimplexSolver
solver = NelderMeadSimplexSolver(len(x0))
solver.SetRandomInitialPoints(lb, ub)
solver.SetStrictRanges(lb, ub)
solver.SetConstraints(constrain)
solver.Solve(my_model, COG)

# obtain the solution
solution = solver.bestSolution
```

`mystic` provides a simple interface to a lot of underlying complexity -- thus allowing a non-specialist user to easily access optimizer configurability and high-performance computing without a steep learning curve. This feature must also be applied to the application of constraints on a function or measure. The natural syntax for a constraint is one of symbolic math, hence `mystic` leverages SymPy [SYM11] to construct a symbolic math parser for the translation of the user's input into functioning constraint code objects:

```
# a user-provided constraints function
constraints = """
x2 = x1
"""
from mystic.constraints import parse
constrain = parse(constraints)
```

The constraints parser is a constraints factory method that can parse multiple and nonlinear constraints, hard or soft (i.e. "$\sim$") constraints, and equality or inequality (i.e. ">") constraints.

Similar tools exist for creating penalty functions, including a `SetPenalty` method for solvers. Available penalty methods include the exterior penalty function method [VEN09], the augmented Lagrange multiplier method [KSK94], and the logarithmic barrier method [JJB03]. At the low-level, penalty functions are bound to the cost function using `mystic`'s `functionWrapper` method.

It is worth noting that the use of a constraints solver c does not require the constraints be bound to the cost function. The evaluation of the constraints are decoupled from the evaluation of the cost function -- hence, with `mystic`, highly-constrained optimization decomposes to the solving of K independent constraints, followed by an unconstrained optimization over only the set of valid points. This method has been shown effective for solving optimization problems where $K \approx 200$ [OSS11].

Seamless Migration to Parallel Computing

`mystic` is built from the ground up to utilize parallel and distributed computing. The decomposition of optimization algorithms into their component parts allow this decomposition to not only be in an abstraction layer, but across process-space. `mystic` provides a `modelFactory` method that convers a user's model to a *service*. We define a service to be an entity that is callable by globally unique identifier. Services can also be called by proxy. In `mystic`, services also include infrastructure for monitoring and handling events. An optimization is then composed as a network of interacting services, with the most common being the user's model or cost function being mapped over parallel resources.

`mystic` provides several stock models and model factories that are useful for testing:

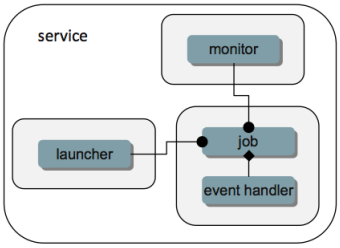

Fig. 4: *Conceptual diagram for a service-based model. Here, the job is the fundamental commodity of work, and is the object on which the service is based -- in `mystic`, this is typically the user's model or a cost function. Services have a global unique identifier, and thus can easily be called by proxy. Note that services may not be located on the machine that requested the service be spawned. Services also can be imbued with infrastructure for monitoring and handling events. Monitors write to a stream that can be piped into another object, such as a logger or one of `mystic`'s viewers.*

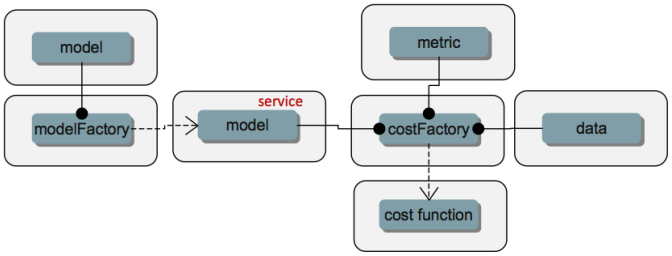

Fig. 5: *Use of a `modelFactory` to cast a user's model $F(x)$ as a service. The model and experimental data G are then bound with a `costFactory` to produce a cost function. A `costFactory` can accept a raw user's model, a model proxy, or a model service (as shown here). A typical metric is $|F(x) - G|^2$.*

```
# generate a model from a stock 'model factory'
from mystic.models.lorentzian import Lorentzian
lorentz = Lorentzian(coeffs)

# evaluate the model
y = lorentz(x)
```

Model factory methods insert `pathos` infrastructure, thus casting a model as a callable *service* that has been imbued with `pathos` infrastructure as shown in Figure (4). The default `launcher` and `map` included in `mystic` are functionally equivalent to execution and `map` within the standard Python distribution. Any user-provided function can be cast as a service through the use of a `modelFactory`:

```
# a user-provided model function
def identify(x)
  return x

# add pathos infrastructure (included in mystic)
from mystic.tools import modelFactory, Monitor
evalmon = Monitor()
my_model = modelFactory(identify, monitor=evalmon)

# evaluate the model
y = my_model(x)

# evaluate the model with a map function
from mystic.tools import PythonMap
my_map = PythonMap()
z = my_map(my_model, range(10))
```

A Framework for Heterogeneous Computing

We have developed a computational job management framework (`pathos`) [MBA10] that offers a simple, efficient, and consistent user experience in a variety of heterogeneous environments from multi-core workstations to networks of large-scale computer clusters. `pathos` provides a single environment for developing and testing algorithms locally -- and enables the user to execute the algorithms on remote clusters, while providing the user with full access to their job history. `pathos` primarily provides the communication mechanisms for configuring and launching parallel computations across heterogenous resources. `pathos` provides stagers and launchers for parallel and distributed computing, where each launcher contains the syntactic logic to configure and launch jobs in an execution environment. Some examples of included launchers are: a queue-less MPI-based launcher, a SSH-based launcher, and a `multiprocessing` launcher. `pathos` also provides a map-reduce algorithm for each of the available launchers, thus greatly lowering the barrier for users to extend their code to parallel and distributed resources. `pathos` provides the ability to interact with batch schedulers and queuing systems, thus allowing large computations to be easily launched on high-performance computing resources. One of the most powerful features of `pathos` is `sshTunnel`, which enables a user to automatically wrap any distributed service calls within an SSH tunnel.

`pathos` is divided into four subpackages: `dill` (a utility for serialization of Python objects), `pox` (utilities for filesystem exploration and automated builds), `pyina` (a MPI-based parallel mapper and launcher), and `pathos` (distributed parallel map-reduce and SSH communication).

`pathos` utilizes `pyre`, which provides tools for connecting components and managing their interactions. The core component used by `pathos` is a service -- a callable object with a configurable connection mechanism. A service can utilize `Launcher` and `Monitor` objects (which provide abstraction to execution and logging, respectively), as well as `Strategy` objects (which provide abstraction patterns for coupling services). A standard interface for services enables massively parallel applications that utilize distributed resources to be constructed from a few simple building blocks. A `Launcher` contains the logic required to initiate execution on the current execution environment. The selection of launcher will determine if the code is submitted to a batch queue, run across SSH tunneled RPC connections, or run with MPI on a multiprocessor. A `Strategy` provides an algorithm to distribute the workload among available resources. Strategies can be static or dynamic. Examples of static strategies include the `equalportion` strategy and the `carddealer` strategy. Dynamic strategies are based on the concept of a worker `pool`, where there are several workload balancing options to choose from. Strategies and launchers can be coupled together to provide higher-level batch and parallel-map algorithms. A `Map` interface allows batch processing to be decoupled from code execution details on the selected platforms, thus enabling the same application to be utilized for sequential, parallel, and distributed parallel calculations.

Globally Unique Message Passing

We must design for the case where an optimizer's calculation spans multiple clusters, with a longevity that may exceed the uptime of any single cluster or node. `hydra` enables any Python object to obtain a network address. After obtaining an address, an object can asynchronously exchange messages with other objects on the network. Through the use of proxy objects, sending messages to remote objects is easy as calling an instance method on a local object. A call to a proxy transparently pickles the function name along with the arguments, packages the message as a datagram, and sends it over the network to the remote object represented by the proxy. On the recieving end, there is a mechanism for responding to the sender of the current message. Since message sending is asynchronous, an object responds to a message by sending another message.

The `modelFactory` method essentially provides `mystic` with a high-level interface for a `pathos` server, with an option to bind a monitor directly to the service. The lower-level construction of a distributed service, using SSH-based communication, is as follows:

```
# a user-provided model function
def identify(x)
  return x

# cast the model as a distributed service
from pathos.servers import sshServer
id = 'foo.caltech.edu:50000:spike42'
my_proxy = sshServer(identify, server=id)

# evaluate the model via proxy
y = my_proxy(x)
```

Parallel map functions are built around available launchers, providing a high-level interface to launching several copies of a model in parallel. The creation of a parallel map that will draw from a pool of two local workers and all available IPC servers at `'foo.caltech.edu'` is shown below:

```
# a user-provided model function
def identify(x)
  return x

# select and configure a parallel map
from pathos.maps import ipcPool
my_map = ipcPool(2, servers=['foo.caltech.edu'])

# evaluate the model in parallel
z = my_map(identify, range(10))
```

Serialization

`dill` extends Python's `pickle` module for serializing and de-serializing Python objects to the majority of the built-in Python and NumPy types. Serialization is the process of converting an object to a byte stream, the inverse of which is converting a byte stream back to a Python object hierarchy.

`dill` provides the user the same interface as the `pickle` module, and also includes some additional features. In addition to pickling Python objects, `dill` provides the ability to save the state of an interpreter session in a single command. Hence, it would be feasible to save a interpreter session, close the interpreter, ship the pickled file to another computer, open a new interpreter, unpickle the session and thus continue from the "saved" state of the original interpreter session.

Filesystem Interaction

`pox` provides a collection of utilities for navigating and manipulating filesystems. This module is designed to facilitate some of the low level operating system interactions that are useful when exploring a filesystem on a remote host, where queries such as

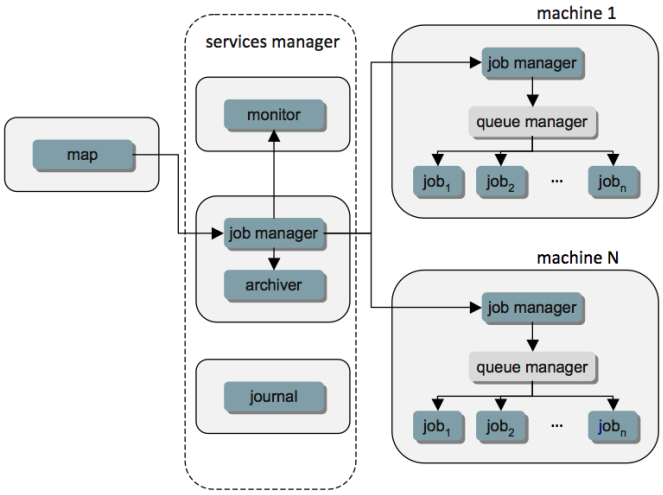

Fig. 6: *Conceptual diagram for heterogeneous job management. A distributed parallel map function is used to copy a service n times on N machines. If the object being mapped is not a service, then the services manager is omitted from the diagram -- the jobs still undergo a distributed launch, but are managed at the machine level.*

"what is the root of the filesystem?", "what is the user's name?", and "what login shell is preferred?" become essential in allowing a remote user to function as if they were logged in locally. While `pox` is in the same vein of both the `os` and `shutil` built-in modules, the majority of its functionality is unique and compliments these two modules.

`pox` provides Python equivalents of several unix shell commands such as "which" and "find". These commands allow automated discovery of what has been installed on an operating system, and where the essential tools are located. This capability is useful not only for exploring remote hosts, but also locally as a helper utility for automated build and installation.

Several high-level operations on files and filesystems are also provided. Examples of which are: finding the location of an installed Python package, determining if and where the source code resides on the filesystem, and determining what version the installed package is.

`pox` also provides utilities to enable the abstraction of commands sent to a remote filesystem. In conjunction with a registry of environment variables and installed utilites, `pox` enables the user to interact with a remote filesystem as if they were logged in locally.

Distributed Staging and Launching

`pathos` provides methods for configuring, launching, monitoring, and controlling a service on a remote host. One of the most basic features of `pathos` is the ability to configure and launch a IPC-based service on a remote host. `pathos` seeds the remote host with a small `portpicker` script, which allows the remote host to inform the localhost of a port that is available for communication.

Beyond the ability to establish a IPC service, and then post requests, is the ability to launch code in parallel. Unlike parallel computing performed at the node level (typically with MPI), `pathos` enables the user to launch jobs in parallel across heterogeneous distributed resources. `pathos` provides a distributed map-reduce algorithm, where a mix of local processors and distributed IPC services can be selected. `pathos` also provides a very basic automated load balancing service, as well as the ability for the user to directly select the resources.

A high-level interface is provided which yields a map-reduce implementation that hides the IPC internals from the user. For example, with `ipcPool`, the user can launch their code as a distributed parallel service, using standard Python and without writing a line of server or parallel batch code. `pathos` also provides tools to build a custom `Map`. In following code, the map is configured to `'autodetect'` the number of processors, and only run on the localhost:

```
# configure and build map
from pathos.launchers import ipc
from pathos.strategies import pool
from pathos.tools import mapFactory
my_map = mapFactory(launcher=ipc, strategy=pool)
```

IPC servers and communication in general is known to be insecure. However, instead of attempting to make the IPC communication itself secure, `pathos` provides the ability to automatically wrap any distributes service or communication in an SSH tunnel. SSH is a universally trusted method. Using `sshTunnel`, `pathos` has launched several distributed calculations on clusters at National Laboratories, and to date has performed test calculations that utilize node-to-node communication between two national lab clusters and a user's laptop. `pathos` allows the user to configure and launch at a very atomistic level, through raw access to `ssh` and `scp`. Any distributed service can be tunneled, therefore less-secure methods of communication can be provided with secure authentication:

```
# establish a tunnel
from pathos.tunnel import sshTunnel
uid = 'foo.caltech.edu:12345:tunnel69'
tunnel_proxy = sshTunnel(uid)

# inspect the ports
localport = tunnel_proxy.lport
remoteport = tunnel_proxy.rport

# a user-provided model function
def identify(x)
  return x

# cast the model as a distributed service
from pathos.servers import ipcServer
id = 'localhost:%s:bug01' % localport
my_proxy = ipcServer(identify, server=id)

# evaluate the model via tunneled proxy
y = my_proxy(x)

# disconnect the tunnel
tunnel_proxy.disconnect()
```

Parallel Staging and Launching

The `pyina` package provides several basic tools to make MPI-based high-performance computing more accessable to the end user. The goal of `pyina` is to allow the user to extend their own code to MPI-based high-performance computing with minimal refactoring.

The central element of `pyina` is the parallel map-reduce algorithm. `pyina` currently provides two strategies for executing the parallel-map, where a strategy is the algorithm for distributing the work list of jobs across the availble nodes. These strategies can be used "in-the-raw" (i.e. directly) to provide map-reduce to a user's own MPI-aware code. Further, `pyina` provides several

map-reduce implementations that hide the MPI internals from the user. With these `Map` objects, the user can launch their code in parallel batch mode -- using standard Python and without ever having to write a line of Parallel Python or MPI code.

There are several ways that a user would typically launch their code in parallel -- directly with `mpirun` or `mpiexec`, or through the use of a scheduler such as torque or slurm. `pyina` encapsulates several of these launching mechanisms as `Launchers`, and provides a common interface to the different methods of launching a MPI job. In the following code, a custom `Map` is built to execute MPI locally (i.e. not to a scheduler) using the `carddealer` strategy:

```
# configure and build map
from pyina.launchers import mpirun
from pyina.strategies import carddealer as card
from pyina.tools import mapFactory
my_map = mapFactory(4, launcher=mpirun, strategy=card)
```

New Massively-Parallel Optimization Algorithms

In `mystic`, optimizers have been extended to parallel whenever possible. To have an optimizer execute in parallel, the user only needs to provide the solver with a parallel map. For example, extending the Differential Evolution [SKP95] solver to parallel is involves passing a `Map` to the `SetEvaluationMap` method. In the example below, each generation has 20 candidates, and will execute in parallel using MPI with 4 workers:

```
# the function to be minimized and the bounds
from mystic.models import rosen as my_model
lb = [0.0, 0.0, 0.0]
ub = [2.0, 2.0, 2.0]

# get termination condition object
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# select the parallel launch configuration
from pyina.maps import MpirunCarddealer
my_map = MpirunCarddealer(4)

# instantiate and configure the solver
from mystic.solvers import DifferentialEvolutionSolver
solver = DifferentialEvolutionSolver(len(lb), 20)
solver.SetRandomInitialPoints(lb, ub)
solver.SetStrictRanges(lb, ub)
solver.SetEvaluationMap(my_map)
solver.Solve(my_model, COG)

# obtain the solution
solution = solver.bestSolution
```

Another type of new parallel solver utilizes the `SetNestedSolver` method to stage a parallel launch of N optimizers, each with different initial conditions. The following code shows the `BuckshotSolver` scheduling a launch of $N = 20$ optimizers in parallel to the default queue, where 5 nodes each with 4 processors have been requested:

```
# the function to be minimized and the bounds
from mystic.models import rosen as my_model
lb = [0.0, 0.0, 0.0]
ub = [2.0, 2.0, 2.0]

# get monitor and termination condition objects
from mystic.monitors import LoggingMonitor
stepmon = LoggingMonitor(1, 'log.txt')
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# select the parallel launch configuration
from pyina.maps import TorqueMpirunCarddealer
my_map = TorqueMpirunCarddealer('5:ppn=4')

# instantiate and configure the nested solver
from mystic.solvers import PowellDirectionalSolver
my_solver = PowellDirectionalSolver(len(lb))
my_solver.SetStrictRanges(lb, ub)
my_solver.SetEvaluationLimits(50)

# instantiate and configure the outer solver
from mystic.solvers import BuckshotSolver
solver = BuckshotSolver(len(lb), 20)
solver.SetRandomInitialPoints(lb, ub)
solver.SetGenerationMonitor(stepmon)
solver.SetNestedSolver(my_solver)
solver.SetSolverMap(my_map)
solver.Solve(my_model, COG)

# obtain the solution
solution = solver.bestSolution
```

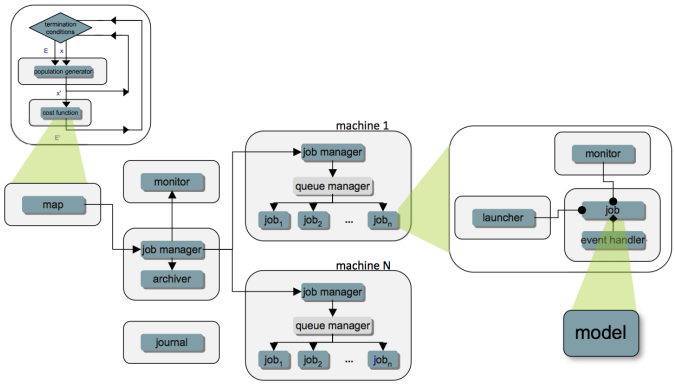

Fig. 7: Conceptual diagram for a `carddealer-DE` optimizer. The optimizer contains a map function that stages n copies of the user's model $F(x)$ in parallel across distributed resources.

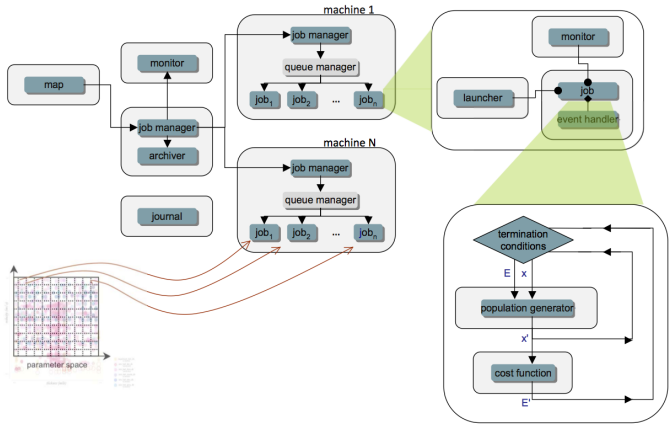

Fig. 8: Conceptual diagram for a `lattice-Powell` optimizer. N Powell's local-search optimizers are launched in parallel, with each optimizer starting from the center of a different lattice cuboid in parameter space. A `buckshot-Powell` optimizer is similar; however, instead utilizes a uniform random distribution of initial values.

Probability and Uncertainty Tooklit

The software framework presented in this paper was designed to solve UQ problems. Calculation of the upper and lower bounds for probability of failure is provided as a framework service. The McDiarmid subdiameter is a model-based measure of sensitivity,

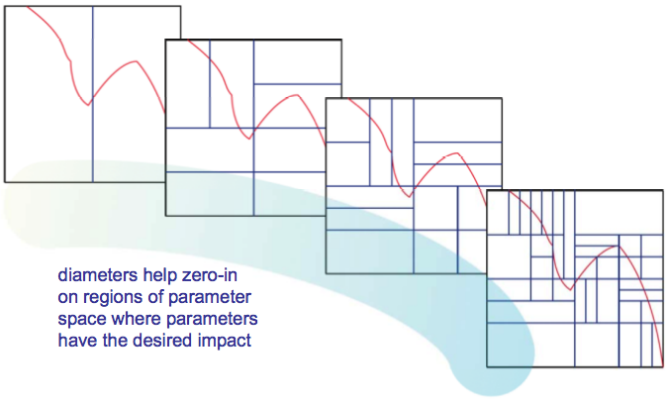

Fig. 9: *Coupling an iterative partitioning algorithm with a sensitivity calculation enables the discovery of critical regions in parameter space.*

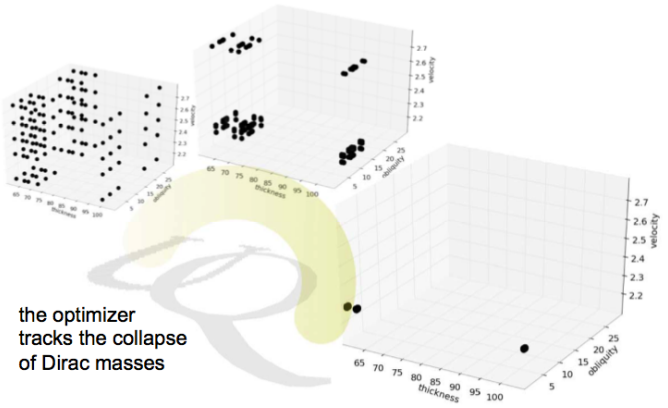

Fig. 10: *Optimal uncertainty quantification is an optimization of probability measures over design parameter space. Collapse of probability masses corresponds to the determination of the critical design parameters.*

and is cast within `mystic` as a global optimization. Diameter calculations can be coupled with partitioning algorithms, and used to discover regions of critical behavior. Optimization over probability measures is also available as a framework service, and is utilized in (OUQ) calculations of optimal bounds.

The minimization or maximization of a cost function is the basis for performing most calculations in `mystic`. The optimizer generates new trial parameters, which are evaluated in a user-provided model function against a user-provided metric. Two simple difference metrics provided are: $metric = |F(x) - G|^2$, where F is the model function evaluated at some trial set of fit parameters $\mathscr{P}$, and G is the corresponding experimental data -- and $metric = |F(x) - F(y)|^2$, where x and y are two slightly different sets of input parameters (6).

`mystic` provides factory methods to automate the generation of a cost function from a user's model. Conceptually, a `costFactory` is as follows:

```
# prepare a (F(X) - G)**2 a metric
def costFactory(my_model, my_data):
  def cost(param):

    # compute the cost
    return ( my_model(param) - my_data )**2

  return cost
```

Suboscillations (6), used in calculations of rigorous sensitivity (such as D_i/D), can also be cast as a cost metric:

```
# prepare a (F(X) - F(X'))**2 cost metric
def suboscillationFactory(my_model, i):

  def cost(param):

    # get X and X' (Xi' is appended to X at param[-1])
    x       = param[:-1]
    x_prime = param[:i] + param[-1:] + param[i+1:-1]

    # compute the suboscillation
    return -( my_model(x) - my_model(x_prime) )**2

  return cost
```

The diameter D (7) is the root-mean square of its component-wise suboscillations. The calculation of the diameter is performed as a nested optimization, as shown above for the `BuckshotSolver`. Each inner optimization is a calculation of a component suboscillation, using the a global optimizer (such as `DifferentialEvolutionSolver`) and the cost metric shown above.

The optimization algorithm takes a set of model parameters $\mathscr{P}$ and the current measure of oscillation $O(\mathscr{P})$ as inputs, and produces an updated $\mathscr{P}$. The optimization loop iterates until the termination conditions are satisfied.

When the global optimization terminates the condition $O(\mathscr{P}) < -(osc_i^2 + \varepsilon)$ is satisfied, and the final set $\mathscr{P}$ is composed of X and X'.

OUQ problems can be thought of optimization problems where the goal is to find the global maximum of a probability function $\mu[H \leq 0]$, where $H \leq 0$ is a failure criterion for the model response function H. Additional conditions in an OUQ problem are provided as constraints on the information set. Typically, a condition such as a mean constraint on H, $m_1 \leq \mathbb{E}_\mu[H] \leq m_2$, will be imposed on the maximization. After casting the OUQ problem in terms of optimization and constraints, we can plug these terms into the infrastructure provided by `mystic`.

Optimal uncertainty quantification (OUQ) is maximization over a probability distribution, and not over a standard difference metric. Therefore the fundamental data structure is not the user-provided model function, but is a user-configured probability measure. For example, a discrete measure is represented by a collection of support points, each with an accompanying weight. Measures come with built-in methods for calculating the mass, range, and mean of the measure, and also for imposing a mass, range, and mean on the measure. Measures also have some very basic operations, including point addition and subtraction, and the formation of product measures.

Global optimizations used in solving OUQ problems are composed in the same manner as shown above for the `DifferentialEvolutionSolver`. The cost function, however, is not formulated as in the examples above -- OUQ is an optimization over product measures, and thus uses `mystic`'s `product_measure` class as the target of the optimization. Also as shown above, the bounds constraints are imposed with the `SetStrictRanges` method, while parameter constraints (composed as below) are imposed with the `SetConstraints` method. The union set of these constraints defines the set $\mathscr{A}$.

So for example, let us define the feasable set

$$\mathscr{A} = \left\{ (f,\mu) \left| \begin{array}{c} f = \texttt{my\_model} : \prod_{i=1}^{3}[\texttt{lb}_\texttt{i},\texttt{ub}_\texttt{i}] \to \mathbb{R}, \\ \mu = \bigotimes_{i=1}^{3}\mu_i \in \bigotimes_{i=1}^{3}\mathscr{M}([\texttt{lb}_\texttt{i},\texttt{ub}_\texttt{i}]), \\ \texttt{m}_\texttt{lb} \le \mathbb{E}_\mu[\texttt{f}] \le \texttt{m}_\texttt{ub} \end{array} \right. \right\} \quad (15)$$

which reduces to the finite-dimensional subset

$$\mathscr{A}_\Delta = \left\{ (f,\mu) \in \mathscr{A} \left| \begin{array}{c} \text{for } \vec{x} \text{ and } \vec{y} \in \prod_{i=1}^{3}[\texttt{lb}_\texttt{i},\texttt{ub}_\texttt{i}], \\ \text{and } \vec{w} \in [0,1], \\ \mu_i = w_i\delta_{x_i} + (1-w_i)\delta_{y_i} \end{array} \right. \right\} \quad (16)$$

where $\vec{x}$ = some (x_1,x_2,x_3), $\vec{y}$ = some (y_1,y_2,y_3), and $\vec{w}$ = some (w_1,w_2,w_3).

To solve this OUQ problem, we first write the code for the bounds, cost function, and constraints -- then we plug this code into a global optimization script, as noted above.

OUQ requires the user provide a list of bounds that follow the formatting convention that `mystic`'s `product_measure.load` uses to build a product measure from a list of input parameters. This roughly follows the definition of a product measure as shown in equation (16), and also is detailed in the comment block below:

```
# OUQ requires bounds in a very specific form...
# param = [wxi]*nx + [xi]*nx \
#       + [wyi]*ny + [yi]*ny \
#       + [wzi]*nz + [zi]*nz
npts = (nx,ny,nz)
lb = (nx * w_lower) + (nx * x_lower) \
   + (ny * w_lower) + (ny * y_lower) \
   + (nz * w_lower) + (nz * z_lower)
ub = (nx * w_upper) + (nx * x_upper) \
   + (ny * w_upper) + (ny * y_upper) \
   + (nz * w_upper) + (nz * z_upper)
```

The constraints function and the cost function typically require the use of measure mathematics. In the example below, the constraints check if `measure.mass` ≈ 1.0; if not, the the measure's mass is normalized to 1.0. The second block of constraints below check if $m_1 \le \mathbb{E}_\mu[H] \le m_2$, where $m_1 =$ `target_mean - error` and $m_2 =$ `target_mean + error`; if not, an optimization is performed to satisfy this mean constraint. The `product_measure` is built (with `load`) from the optimization parameters `param`, and after all the constraints are applied, `flatten` is used to extract the updated `param`:

```
from mystic.math.measures import split_param
from mystic.math.dirac_measure import product_measure
from mystic.math import almostEqual

# split bounds into weight-only & sample-only
w_lb, m_lb = split_param(lb, npts)
w_ub, m_ub = split_param(ub, npts)

# generate constraints function
def constraints(param):
  prodmeasure = product_measure()
  prodmeasure.load(param, npts)

  # impose norm on measures
  for measure in prodmeasure:
    if not almostEqual(float(measure.mass), 1.0):
      measure.normalize()

  # impose expectation on product measure
  E = float(prodmeasure.get_expect(my_model))
  if not (E <= float(target_mean + error)) \
  or not (float(target_mean - error) <= E):
    prodmeasure.set_expect((target_mean, error), \
                           my_model, (m_lb, m_ub))
  # extract weights and positions
  return prodmeasure.flatten()
```

The PoF is calculated in the cost function with the `pof` method:

```
# generate maximizing function
def cost(param):
  prodmeasure = product_measure()
  prodmeasure.load(param, npts)
  return MINMAX * prodmeasure.pof(my_model)
```

We find the *supremum* (as in 12) when `MINMAX=-1` and, upon solution, the function maximum is `-solver.bestEnergy`. We find the *infimum* when `MINMAX=1` and, upon solution, the function minimum is `solver.bestEnergy`.

Future Developments

Many of the features presented above are not currently in released versions of the code. Of primary importance is to migrate these features from development branches to a new release.

The next natural question beyond "what is the sensitivity of a model to an input parameter?" is "how does the correlation between input parameters affect the outcome of the model?". Methods for calculating parameter correlation will be very useful in analysis of results. Another natural question is how to handle uncertainty in the data.

New partitioning algorithms for the discovery of regions of critical behavior will be added to `mystic`. Currently the only partitioning rule drives the optimizer toward partitioning space such that the upper bounds of a "piecewise-McDiarmid" type are iteratively tightened [STM11]. We will extend the partitioning algorithm not to refine the diameter, but to discover regions where the diameters meet a set of criteria (such as: regions where there are two roughly equal subdiameters that account for 90% or more of the total diameter (i.e. automated discovery of regions where two parameters compete to govern the system behavior). `mystic` will also further expand its base of available statistical and measure methods, equation solvers, and also make available several more traditional uncertainty quantification methods. `mystic` will continue to expand its base of optimizers, with particular emphasis on new optimization algorithms that efficiently utilize parallel computing. `mystic` currently has a few simple parallel optimization algorithms, such as the `LatticeSolver` and `BuckshotSolver` solvers; however, algorithms that utilize a variant of game theory to do speculation about future iterations (i.e. break the paradigm of an iteration being a blocker to parallelism), or use parallelism and dynamic constraints to allow optimizers launched in parallel to avoid finding the same minimum twice, are planned. Parallelism in optimization also allows us to not only find the global minima, but to simultaneously find all local minima and transition points -- thus providing a much more efficient means of mapping out a potential energy surface. Solving uncertainty quantification problems requires a lot of computational resources and often must require a minimum of both model evaluations and accompanying experiments, so we also have to keep an eye on developing parallel algorithms for global optimization with overall computational efficiency.

`pathos` includes utilities for filesystem exploration and automated builds, and a utility for the serialization of Python objects, however these framework services will need to be made more robust as more platforms and more extensive objects and codes are tackled. Effort will continue on expanding the management and platform capabilities for `pathos`, unifying and hardening

the map interface and providing load balancing for all types of connections. The high-level interface to analysis circuits will be extended to encompass new types of logic for combining and nesting components (as nested optimizers are utilized in many materials theory codes). Monitoring and logging to files and databases across parallel and distributed resources will be migrated from `mystic` and added as `pathos` framework services.

Summary

A brief overview of the mathematical and software components used in building a software framework for predictive science is presented.

Acknowledgements

This material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number DE-FC52-08NA28613, and by the National Science Foundation under Award Number DMR-0520547.

REFERENCES

[MHA09] M. McKerns, P. Hung, M. Aivazis, *mystic: a simple model-independent inversion framework*, 2009, http://dev.danse.us/trac/mystic.

[MBA10] M. McKerns, M. Aivazis, *pathos: a framework for heterogeneous computing*, 2010, http://dev.danse.us/trac/pathos.

[LOO08] L. Lucas, H. Owhadi, M. Ortiz, *Rigorous verification, validation, uncertainty quantification and certification through concentration-of-measure ineqalities*, Computer Methods in Applied Mechanics and Engineering 197, 4591, 2008.

[MCD89] C. McDiarmid, *On the method of bounded differences*, In: Surveys in combinatorics, 1989, vol. 141 of London Math. Soc. Lecture Note Ser., Cambridge Univ. Press, Cambridge, 148.

[BBL04] S. Boucheron, O. Bousquet, G. Lugosi, *Concentration inequalities*, In: Advanced Lectures in Machine Learning, 2004, Springer, 208.

[LED01] M. Ledoux, *The concentration of measure phenomenon*, In: Mathematical Surveys and Monographs, 2001, vol 89, American Mathematical Society.

[ALL11] M. Adams, A. Lashgari, B. Li, M. McKerns, J. Mihaly, M. Ortiz, H. Owhadi, A. Rosakis, M. Stalzer, T. Sullivan, *Rigorous model-based uncertainty quantification with application to terminal ballistics, part II: systems with uncontrollable inputs and large scatter*, Journal of the Mechanics of Physics and Solids, (submitted).

[KLL11] A. Kidane, A. Lashgari, B. Li, M. McKerns, M. Ortiz, H. Owhadi, G. Ravichandran, M. Stalzer, T. Sullivan, *Rigorous model-based uncertainty quantification with application to terminal ballistics, part I: systems with controllable inputs and small scatter*, Journal of the Mechanics of Physics and Solids, (submitted).

[OSS11] H. Owhadi, C. Scovel, T. Sullivan, M. McKerns, M. Ortiz, *Optimal uncertainty quantification*, SIAM Review, (submitted).

[VEN09] P. Venkataraman. *Applied Optimization with MATLAB Programming*. Hoboken, NJ: John Wiley & Sons, 2009.

[KSK94] B. Kanna, S. Kramer, *An Augmented Lagrange Multiplier Based Method for Mixed Integer Discrete Continuous Optimization and Its Applications to Mechanical Design*. J. Mech. Des. June 1994. Volume 116, Issue 2, 405. DOI:10.1115/1.2919393.

[JJB03] P. Jensen, J. Bard, *Algorithms for Constrained Optimization*. Supplement to: Operations Research Models and Methods, 2003, http://www.me.utexas.edu/~jensen/ORMM/supplements/units/nlp_methods/const_opt.pdf.

[SYM11] O. Certik, et al, *SymPy: Python Library for Symbolic Mathematics*, http://code.google.com/p/sympy.

[SKP95] R. Storn and K. Price. *Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces*. TR-95-012. 1995.

[KROOO] D. Kroshko, et al, *OpenOpt*, http://openopt.org/

[JOP01] E. Jones, T. Oliphant, P. Peterson, et al, *SciPy: Open Source Scientific Tools for Python*, 2001, http://www.scipy.org/

[DAKOT] B. Adams, W. Bohnhoff, K. Dalbey, J. Eddy, M. Eldred, D. Gay, K. Haskell, P. Hough, L. Swiler, *DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.0 User's Manual*, Sandia Technical Report SAND2010-2183, December 2009.

[DAKUQ] M. Eldred, A. Giunta, B. van Bloemen Waanders, S. Wojtkiewicz, W. Hart, M. Alleva, *DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 3.0 Developers Manual*, Sandia Technical Report SAND2001-3514, April 2002.

[MAGA1] M. Aivazis, et al, *pyre: an integration framework for high performance computing*, http://danse.us/trac/pyre.

[GVRPY] G. van Rossum, et al, *Python Programming Language*, http://www.python.org/

[MPROC] R. Oudkerk, *multiprocessing*, http://pyprocessing.berlios.de/

[MPI4P] L. Dalcin, *mpi4py: MPI for Python*, http://mpi4py.googlecode.com/

[VVPPP] V. Vanovschi, *pp: Parallel Python Software*, http://www.parallelpython.com/

[NUMPY] T. Oliphant, et al, *NumPy*, http://www.numpy.org/

[MATPL] J. Hunter, *Matplotlib: A 2D Graphics Environment*, Journal of Computing in Science and Engineering, vol 9(3), 2007

[EKL02] I. Egorov, G. Kretinin, I. Leshchenko, S. Kuptzov, *IOSO Optimization Toolkit - Novel Software to Create Better Design*, 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, 04 - 06 Sep. 2002, Atlanta, Georgia. http://www.iosotech.com/publicat.htm

[BMM10] S. Benson, L. Curfman McInnes, J. More, T. Munson, J. Sarich, *TAO User Manual (Revision 1.10.1)*, 2010, Mathematics and Computer Science Division Argonne National Laboratory ANL/MCS-TM-242, http://www.mcs.anl.gov/tao

[STM11] T. Sullivan, U. Topcu, M. McKerns, H. Owhadi, *Uncertainty quantification via codimension-one partitioning*, International Journal for Numerical Methods in Engineering, 85, 1499 (2011).

[LJH99] T. Leonard, J. Hsu. *Bayesian methods*, In: volume 5 of Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge, 1999. An analysis for statisticians and interdisciplinary researchers.

[MAT09] The MathWorks Inc., Technical Report 91710v00, March 2009.

PyStream: Compiling Python onto the GPU

Nick Bray‡∗

Abstract—PyStream is a static compiler that can radically transform Python code and run it on a Graphics Processing Unit (GPU). Python compiled to run on the GPU is ~100,000x faster than when interpreted on the CPU. The PyStream compiler is specially designed to simplify the development of real-time rendering systems by allowing the entire rendering system to be written in a single, highly productive language. Without PyStream, GPU-accelerated real-time rendering systems must contain two separate code bases written in two separate languages: one for the CPU and one for the GPU. Functions and data structures are not shared between the code bases, and any common functionality must be redundantly written in both languages. PyStream unifies a rendering system into a single, Python code base, allowing functions and data structures to be transparently shared between the CPU and the GPU. A single, unified code base makes it easy to create, maintain, and evolve a high-performance GPU-accelerated application.

Index Terms—pystream, compiling python, gpu

Introduction

High-performance computer hardware can be difficult to program because ease of programming is often traded for raw performance. For example, graphics processing units (GPUs) are traditionally programmed in languages that either restrict memory use or explicitly expose the memory hierarchy to the programmer. The OpenGL Shading Language (GLSL) is an example of the first, and OpenCL is an example of the second. Neither type of language is particularly easy to use, rather they are designed to address a potential bottleneck for GPU architectures: memory bandwidth. GPUs pack enough functional units into a single chip that overall performance can easily be limited by the memory subsystem's ability to feed data to the functional units.

Ease of programming is not the only issue when using GPU-specific languages. These languages are specialized for performance-critical numeric computations and are not suitable for writing a complete application. For instance, these languages cannot load data from disk or provide a graphical user interface. Instead, GPU languages typically provide APIs to interoperate with a different, general-purpose language. Using these APIs results in an application with two code bases, each with distinct semantics. Additional *glue code* is also required overcome the impedance mismatch between the code bases. Glue code is used to remap and transfer data structures from one code base to another. Glue code is also used to invoke functions across the language boundary.

∗ *Corresponding author: ncbray@google.com*
‡ *Google*

Fig. 1: *Naïve background color.*

Fig. 2: *Background color calculation using shared code.*

Constructing a GPU-accelerated application with two code bases and glue code has a number of software engineering costs. For instance, any data transferred between the CPU and the GPU must have its structure defined in both languages and have glue code to remap and transfer the data. Any modification to such a data structure will require modifying all its definitions. Certain functions may also need to be duplicated so they can be used in each code base. For example, Figure 1 shows a rendering

system where the background color - calculated on the CPU - does not take into account the color processing performed on the GPU. Figure 2 shows the same rendering system, but with the color-processing code duplicated on the CPU and applied to the background color. Ultimately, applications that incorporate GPU-specific languages tend to resist change because data structures and functions in two code bases must be kept in sync. This complicates the process of maintaining and evolving such an application.

PyStream

PyStream [Bra10] is a static source-to-source compiler that translates Python code into GLSL for use in real-time graphical rendering. PyStream also generates the glue code necessary to seamlessly invoke the generated GLSL code from a Python application. PyStream allows applications incorporating GPU-accelerated real-time rendering to be written as a single, unified Python code base. This allows the high productivity of the Python language to be used while also gaining the performance of GPU acceleration. PyStream sidesteps the problems that arise from having two code bases, which would otherwise diminish the productivity gains of using Python.

Programming a GPU with Python allows the use of object-oriented programming, polymorphic functions, and other programming language features that are often not available in GPU-specific languages. Python also has the advantage of being more concise. In practice, Python code is roughly five to seven times more terse than the corresponding GLSL code.

Language Restrictions

PyStream can compile a restricted subset of Python onto the GPU. Restrictions are necessary to make the compilation process tractable. Restrictions are also necessary because of the fundamental limitations of modern GPU hardware. PyStream's restricted subset of Python provides, at minimum, the functionality of GLSL but with the syntax, semantics, and abstraction mechanisms of Python, as well as complete integration with Python applications. PyStream requires the following to translate code onto a GPU:

- A closed world.
- No global side effects.
- No recursive function calls.
- Bounded memory usage.

To statically compile a Python program, a closed world must be created. If a program can call a function that the compiler knows nothing about, then the compiler must assume that the function can have arbitrary side effects: rewriting globals, classes, and other data structures. In such a situation, a static compiler cannot prove anything about the program and therefore cannot transform it in any meaningful way. To prevent this situation, PyStream disallows the execution of unknown code. Dynamic code compilation and execution, such as through the use of `exec` and `eval`, is forbidden. In addition, modules are imported at compile time and assumed to never change thereafter.

GLSL has several restrictions, when compared to Python, and they are adopted by PyStream so that it can generate GLSL code. For instance, GLSL programs are constrained to have no global side effects. Code compiled by PyStream must behave the same. GLSL does not allow recursive function calls because it is designed to run on hardware without a call stack. This restriction is adopted by PyStream. Similarly, GLSL is designed to run in an environment where memory is statically allocated for each processor. PyStream in turn requires that the code it compiles have bounded memory usage, allowing the compiler to statically allocate memory.

Fig. 3: *An image produced by the example rendering system.*

In practice, the most significant of these restrictions appears to be the need for bounded memory usage. This restriction prevents the use of recursive data structures and most mutable `list`, `dict`, and `set` objects. For example, if a program appends to a list inside of a loop, the compiler will be unable to determine the maximum size of the list. Future improvements to the compiler may allow it to bound the number of loop iterations in some cases, this problem is equivalent to the halting problem in the general case.

Most of these restrictions are applied after the compiler optimizes a program. For example, a highly polymorphic function may initially appear to be recursive, but this recursion can disappear once the function has been duplicated and specialized for the different situations in which it is called. As will be discussed later, PyStream uses a novel approach for representing Python programs. This approach treats the Python interpreter as part of the program being compiled. There are often recursive calls through the interpreter, such as when the addition of a vector type is implemented in terms of the addition of its scalar elements. This pattern is so pervasive that disallowing recursive calls before optimizations are applied would disallow most Python programs. Problematically, disallowing recursive calls after compilation requires that a programmer must understand how the compiler behaves. Although this conceit is undesirable, it is necessary.

PyStream currently does not support a number of Python features, including exceptions and closures. These features will be supported in the future.

PyStream in Practice

A real-time rendering system was developed with the PyStream compiler to validate the design of the compiler. The example ren-

dering system implements the core algorithms used by the game Starcraft 2 [Fil08]. Rendering systems typically use many different algorithms to produce a final image. These algorithms are divided into *shader programs* that that are executed on batches of data sent to the GPU. The example rendering system contains 8 different shader programs. A shader programs is further subdivided into several individual *shaders* that process different kinds of data, such as vertices in a 3D model or pixels being written into an image. The code for one of the shader programs in the example rendering system is included below.

```
class AmbientPass(ShaderProgram):

  def shadeVertex(self, context, pos, texCoord):
    context.position = pos
    return texCoord,

  def shadeFragment(self, context, texCoord):
    # Sample the underlying geometry
    g = self.gbuffer.sample(texCoord)
    # Sample the ambient occlusion
    ao = self.ao.texture(texCoord).xyz
    # Calculate the lighting
    ambientLight = self.env.ambientColor(g.normal)*ao
    # Modulate the output
    color=vec4(g.diffuse*ambientLight, 1.0)
    context.colors = (color,)
```

This shader program performs a specific kind of lighting calculation for the example rendering system. PyStream's shader programs are a Pythonic version of GLSL's shader programs. The previous shader program is implemented as a class that contains two shader methods: a vertex shader and a fragment shader. The first two arguments for each shader are special. The `self` argument holds data that is constant during the execution of the shader. The `context` argument holds an object with shader-specific fields. For example, colors written to the `context.colors` field inside of a fragment shader will be written into the image(s) being rendered after the shader has been executed. All subsequent arguments correspond to streams of data being fed into the shader. Return values correspond to streams of data produced by the shader.

Python's abstraction mechanisms are used throughout the example rendering system. For instance, algorithms for calculating how light reflects off surfaces are encapsulated in polymorphic `Material` objects. This allows the appearance of a surface to be controlled by composing a shader object with different types of material objects. Rendering systems often contain custom code generators [Mit07] because the GPU-specific language they are using does not natively support polymorphism.

Compiling Python

PyStream takes a novel approach to compiling Python that is simpler and more flexible than previous approaches [Sal04], [Pyp11]. The key to PyStream's approach is that it keeps its internal representation of the program it is compiling as simple as possible. Compiling Python can be potentially complicated because the language is filled with numerous special cases. For example, adding two objects together can result in the `__add__` method being called on the first object, the `__radd__` method being called on the second object, or an exception being thrown. More precisely, the interpreter can do all of the above for a single operation if both methods exist but return `NotImplemented`. Calling either method can result in arbitrary code being executed and can have arbitrary side effects, so the precise definition of the addition operator is both complicated and ambiguous. Any relationship between Python's addition operator and the mathematical concept of addition is a convention and not an intrinsic part of the language. Virtually every Python operation can execute arbitrary code, even operations such as reading an attribute of an object.

Prior to PyStream, Python compilers attempted to embed extensive knowledge of Python's semantics into their algorithms. For example, every analysis algorithm and optimization would need to implicitly understand how the interpreter dispatched addition operations. Typically this knowledge was not precise, and did not cover every corner case. PyStream takes a different approach. Instead of trying to embed a complete knowledge of Python's semantics into its algorithms, it treats the interpreter as if it were a library being called by the Python program. This allows PyStream to easily and accurately analyze Python's complex semantics without complicating the compiler. The consequence of this approach is that PyStream appears to process three times as much code as other Python compilers. This extra code would need to be evaluated one way or the other, PyStream evaluates it explicitly as code rather than implicitly inside the compiler.

Because PyStream treats the interpreter as part of the program, standard optimizations such as dead code elimination and function inlining are extremely effective at eliminating Python's run time overhead. Interpreter functions are initially quite complicated, but they are typically optimized down to a single operation and later inlined. In addition to the standard optimizations, several Python-specific transformations are also performed. For example, method calls are optimized to eliminate the creation of bound method objects wherever possible.

Mapping Python onto the GPU

After analyzing and optimizing a program, PyStream then maps it onto the GPU. One of the biggest challenges in mapping a Python shader program onto the GPU is the presence of memory operations. GLSL does not support pointers in any form: the address of an objects cannot be taken, and function arguments are passed by value. Python, on the other hand, hold every object by reference. PyStream bridges this semantic gap by eliminating as many memory operations as possible and then emulating the rest.

Before even trying to map a program onto the GPU, PyStream aggressively eliminates as many memory operations as possible. If PyStream can eliminate every memory operation, translating the program into GLSL is trivial. The optimizations PyStream performs are a mixture of load/store elimination and shader-specific transformations such as flattening the input and output data structures for each shader into a list of local variables. In practice, these optimization eliminate almost all the memory operations in the example rendering system.

It is not always possible to eliminate every memory operation, however. PyStream uses two different strategies to emulate the remaining memory operations. If an object is never modified or is only held by a single reference at a time, PyStream copies the object as needed rather than treating it as a distinct memory location. If an object is held by multiple references and also modified, PyStream places it in an array of objects and uses an index into the array as a pointer to the object.

Performance

The example rendering system demonstrates that PyStream is quite effective at compiling Python shaders. A manual inspection

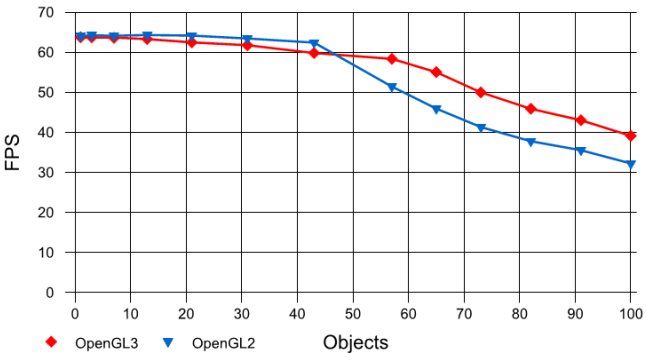

Fig. 4: *Performance of the example rendering system as the number of objects drawn is increased.*

of the generated GLSL code reveals that it is close to what would be written by hand. Quantitatively, PyStream provides a massive speedup for the compiled shaders. The following table shows the time taken to draw one million pixels with a Python shader program when it is interpreted on the CPU versus when it is compiled onto the GPU. Measurements were taken on a AMD Athlon 64 X2 3800+ CPU with a NVidia 9800 GT GPU running Windows XP and Python 2.6.4.

Shader	CPU	GPU	Speedup
material	220.5 s	5.62 ms	39,211x
skybox	35.5 s	0.81 ms	43,568x
ssao	444.9 s	1.44 ms	308,958x
bilateral	429.1 s	1.49 ms	288,956x
ambient	64.1 s	0.84 ms	76,310x
light	127.1 s	0.95 ms	133,789x
blur	74.2 s	0.54 ms	138,692x
post	442.6 s	9.57 ms	46,272x
average	180.8 s	1.23 ms	146,712x

On average, the shaders in the example rendering system run 146,712x faster when compiled onto the GPU than when interpreted on the CPU. The CPU timings are synthetic and only measure the execution time of the shader code and neglect the time required to sample textures and other functionality in the rendering system. The GPU timings take all costs into account, so the speedup is understated. Five orders of magnitude speedup is reasonable, however. Compiling an optimized Python program into C can provide two orders of magnitude speedup [Sal04]. For unoptimized programs taking full advantage of Python's abstraction mechanisms, an additional order of magnitude of speedup can be achieved because a static compiler can inline functions and globally optimize the program whereas an interpreter always pays the abstraction overhead. Switching from a CPU to a GPU can easily provide another two orders of magnitude speedup for real-time rendering, a task the GPU was designed for. Taken together, this easily explains the net speedup.

Figure 4 shows the performance of the example rendering system, in frames per second (FPS), as the number of objects drawn increases. Drawing more objects requires more computation, and will naturally reduce the rate at which images are produced. Rendering systems may be bottlenecked by factors other than computation; they can also be limited by the rate that glue code can transfer data to the GPU. PyStream can generate glue code for both OpenGL 2 and OpenGL 3. OpenGL 3 has features that let it transfer data more efficiently to the GPU. As seen in the figure, these features can offer a ~20% speed improvement when the rendering system is bottlenecked by its glue code. This demonstrates an interesting benefit of PyStream: future proofing. PyStream can take advantage of new features offered by GPUs and GPU APIs without requiring modifications to the rendering system.

Conclusion

PyStream takes a unique approach to high-performance high-level programming. The compiler can map a significant portion of a general-purpose language onto a GPU, and allow a complete GPU-accelerated application to be written with a single code base. This demonstrates that productive high-level languages and high performance are not mutually exclusive, even for critical computational kernels.

References

[Bra10] N. C. Bray. *PyStream: Python Shaders Running on the GPU*, PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign.

[Fil08] D. Filion and R. McNaughton, *Effects & techniques*, SIGGRAPH '08: ACM SIGGRAPH 2008 Classes, pp. 133-164.

[Mit07] M. Mittring, *Finding next gen: CryEngine 2*, SIGGRAPH '07: ACM SIGGRAPH 2007 Courses, 2007, pp. 97-121.

[Pyp11] Online: http://codespeak.net/pypy/dist/pypy/doc/

[Sal04] M. Salib, *Starkiller: A static type inferencer and compiler for Python*, M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Bringing Parallel Performance to Python with Domain-Specific Selective Embedded Just-in-Time Specialization

Shoaib Kamil[‡*], Derrick Coetzee[‡], Armando Fox[‡]

Abstract—Today's *productivity programmers*, such as scientists who need to write code to do science, are typically forced to choose between productive and maintainable code with modest performance (e.g. Python plus native libraries such as SciPy [SciPy]) or complex, brittle, hardware-specific code that entangles application logic with performance concerns but runs two to three orders of magnitude faster (e.g. C++ with OpenMP, CUDA, etc.). The dynamic features of modern productivity languages like Python enable an alternative approach that bridges the gap between productivity and performance. SEJITS (Selective, Embedded, Just-in-Time Specialization) embeds domain-specific languages (DSLs) in high-level languages like Python for popular computational *kernels* such as stencils, matrix algebra, and others. At runtime, the DSLs are "compiled" by combining expert-provided source code templates specific to each problem type, plus a strategy for optimizing an abstract syntax tree representing a domain-specific but language-independent representation of the problem instance. The result is efficiency-level (e.g. C, C++) code callable from Python whose performance equals or exceeds that of handcrafted code, plus performance portability by allowing multiple code generation strategies within the same specializer to target different hardware present at runtime, e.g. multicore CPUs vs. GPUs. Application writers never leave the Python world, and we do not assume any modification or support for parallelism in Python itself.

We present Asp ("Asp is SEJITS for Python") and initial results from several domains. We demonstrate that domain-specific specializers allow highly-productive Python code to obtain performance meeting or exceeding expert-crafted low-level code on parallel hardware, without sacrificing maintainability or portability.

Index Terms—parallel programming, specialization

Introduction

It has always been a challenge for productivity programmers, such as scientists who write code to support doing science, to get both good performance and ease of programming. This is attested by the proliferation of high-performance libraries such as BLAS, OSKI [OSKI] and FFTW [FFTW], by domain-specific languages like SPIRAL [SPIRAL], and by the popularity of the natively-compiled SciPy [SciPy] libraries among others. To make things worse, processor clock scaling has run into physical limits, so future performance increases will be the result of increasing hardware parallelism rather than single-core speedup, making programming even more complex. As a result, programmers must choose between productive and maintainable but slow-running code on the one hand, and performant but complex and hardware-specific code on the other hand.

The usual solution to bridging this gap is to provide compiled native libraries for certain functions, as the SciPy package does. However, in some cases libraries may be inadequate or insufficient. Various families of computational patterns share the property that while the *strategy* for mapping the computation onto a particular hardware family is common to all problem instances, the specifics of the problem are not. For example, consider a stencil computation, in which each point in an n-dimensional grid is updated with a new value that is some function of its neighbors' values. The general strategy for optimizing sequential or parallel code given a particular target platform (multicore, GPU, etc.) is independent of the specific function, but because that function is unique to each application, capturing the stencil abstraction in a traditional compiled library is awkward, especially in the efficiency level languages typically used for performant code (C, C++, etc.) that don't support higher-order functions gracefully.

Even if the function doesn't change much across applications, work on auto-tuning [ATLAS] has shown that for algorithms with tunable implementation parameters, the performance gain from fine-tuning these parameters compared to setting them naively can be up to 5×. [SC08] Indeed, the complex internal structure of auto-tuning libraries such as the Optimized Sparse Kernel Interface [OSKI] is driven by the fact that often runtime information is necessary to choose the best execution strategy or tuning-parameter values.

We therefore propose a new methodology to address this performance-productivity gap, called SEJITS (Selective Embedded Just-in-Time Specialization) [Cat09]. This methodology embeds domain-specific languages within high-level languages, and the embedded DSLs are specialized at runtime into high-performance, low-level code by leveraging metaprogramming and introspection features of the host languages, all invisibly to the application programmer. The result is performance-portable, highly-productive code whose performance rivals or exceeds that of implementations hand-written by experts.

The insight of our approach is that because each embedded DSL is specific to just one type of computational pattern (stencil, matrix multiplication, etc.), we can select an implementation

* *Corresponding author: skamil@cs.berkeley.edu*
‡ *University of California, Berkeley*

strategy and apply optimizatic
knowledge in generating the e
returning to the domain of ste
skewing [Wonn00] involves bl
repeatedly to the same grid. Th
unless we know the computatic
stencil's "footprint," so a gene
unable to identify the opportun

We therefore leverage the
guages like Python to defer
must do at compile time, and
knowledge than can be inferre

Asp: Approach and Mechanic

High-level productivity or sc
include sophisticated introspec
terface) capabilities. We lever
build domain- and machine-s
user-written code in a high-l
expose parallelism, and then ge
in a low-level language. Then,
executed. This entire process
the user, it appears that an inte

Asp (a recursive acronym
is a collection of libraries tha
Python, using Python both as
programmers write their code
guage in which transformatio
out (the *transformation langu*
and transformation languages
happily serves both purposes w

Specifically, Asp provides
classes (*specializers*), each of
putational pattern. Applicatior
specific problem instances. The specializer class's methods use a combination of pre-supplied low-level source code snippets (*templates*) and manipulation of the Python abstract syntax tree (AST, also known as a parse tree) to generate low-level source code in an efficiency-level language (ELL) such as C, C++ or CUDA.

For problems that call for passing in a function, such as the stencil example above, the application writer codes the function in Python (subject to some restrictions) and the specializer class iterates over the function's AST to lower it to the target ELL and inline it into the generated source code. Finally, the source code is compiled by an appropriate conventional compiler, the resulting object file is dynamically linked to the Python interpreter, and the method is called like a native library.

Python code in the application for which no specializer exists is executed by Python as usual. As we describe below, a recommended best practice for creating new specializers is that they include an API-compatible, pure-Python implementation of the kernel(s) they specialize in addition to providing a code-generation-based implementation, so that every valid program using Asp will also run in pure Python without Asp (modulo removing the import directives that refer to Asp). This allows the kernel to be executed and debugged using standard Python tools, and provides a reference implementation for isolating bugs in the specializer.

One of Asp's primary purposes is separating application and algorithmic logic from code required to make the application run

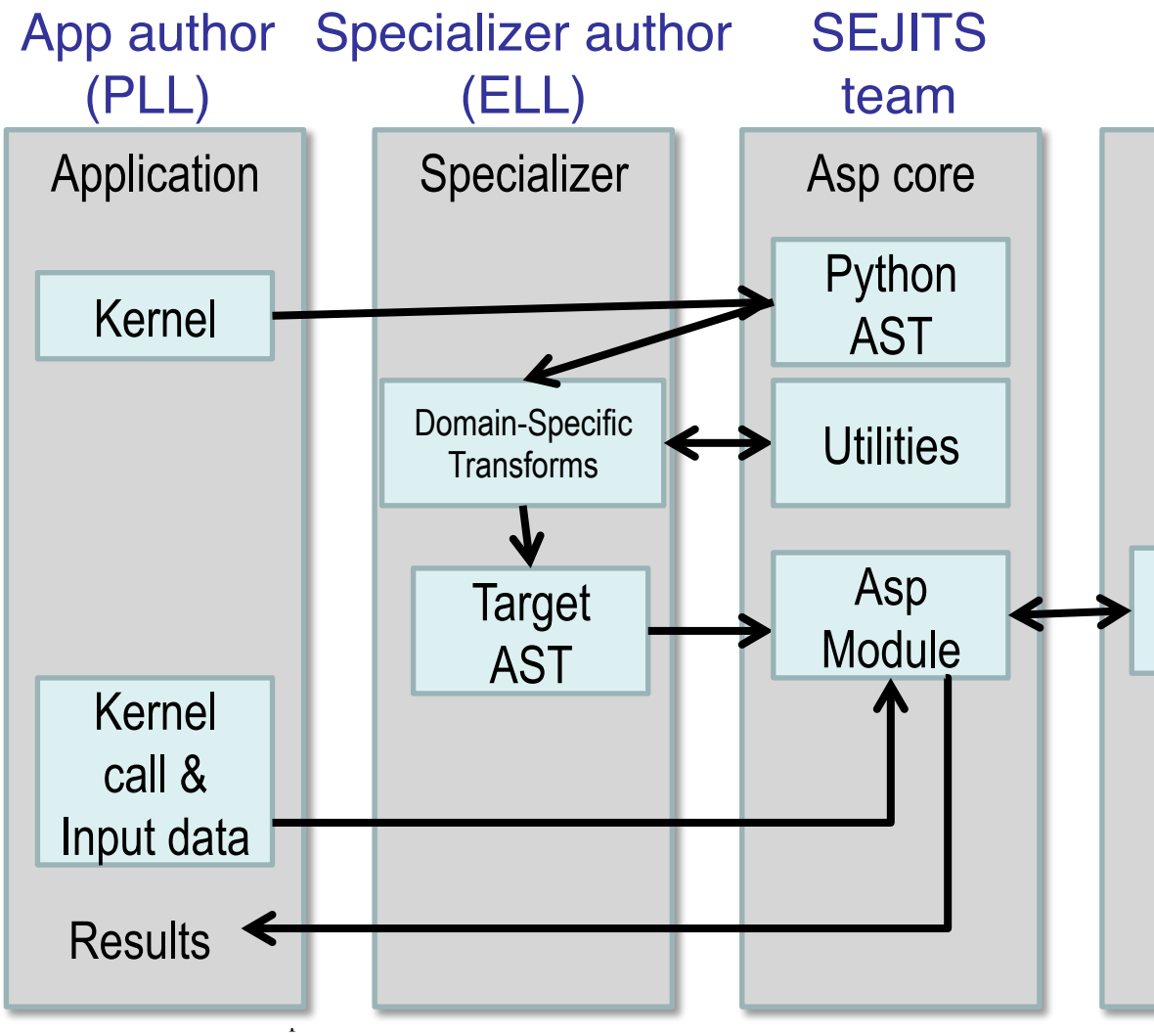

Fig. 1: *Separation of concerns in Asp. App authors write code that is transformed by specializers, using Asp infrastructure and third-party libraries.*

fast. Application writers need only program with high-level class-based constructs provided by specializer writers. It is the task of these specializer writers to ensure the constructs can be specialized into fast versions using infrastructure provided by the Asp team as well as third-party libraries. An overview of this separation is shown in Figure 1.

An overview of the specialization process is as follows. We intercept the first call to a specializable method, grab the AST of the Python code of the specializable method, and immediately transform it to a domain-specific AST, or DAST. That is, we immediately move the computation into a domain where problem-specific optimizations and knowledge can be applied, by applying transformations to the DAST. Returning once again to the stencil, the DAST might have nodes such as "iterate over neighbors" or "iterate over all stencil points." These abstract node types, which differ from one specializer to another, will eventually be used to generate ELL code according to the code generation strategy chosen; but at this level of representation, one can talk about optimizations that make sense *for stencils specifically* as opposed to those that make sense *for iteration generally*.

After any desired optimizations are applied to the domain-specific (but language- and platform-independent) representation

```
from stencil_kernel import *

class ExampleKernel(StencilKernel):
    def kernel(self, in_grid, out_grid):
        for x in out_grid.interior_points():
            for y in in_grid.neighbors(x, 1):
                out_grid[x] = out_grid[x] + in_grid[y]

in_grid = StencilGrid([5,5])
in_grid.data = numpy.ones([5,5])
out_grid = StencilGrid([5,5])
ExampleKernel().kernel(in_grid, out_grid)
```

```
1 1 1
1 1 1
1 1 1
      ↓
0 0 0
0 4 0
0 0 0
```

Fig. 2: *Example stencil application. Colored source lines match up to nodes of same color in Figure 4.*

of the problem, conversion of the DAST into ELL code is handled largely by CodePy [CodePy]. Finally, the generated source code is compiled by an appropriate downstream compiler into an object file that can be called from Python. Code caching strategies avoid the cost of code generation and compilation on subsequent calls.

In the rest of this section, we outline Asp from the point of view of application writers and specializer writers, and outline the mechanisms the Asp infrastructure provides.

Application Writers

From the point of view of application writers, using a specializer means installing it and using the domain-specific classes defined by the specializer, while following the conventions outlined in the specializer documentation. Thus, application writers never leave the Python world. As a concrete example of a non-trivial specializer, our structured grid (stencil) specializer provides a StencilKernel class and a StencilGrid class (the grid over which a stencil operates; it uses NumPy internally). An application writer subclasses the StencilKernel class and overrides the function `kernel()`, which operates on StencilGrid instances. If the defined kernel function is restricted to the class of stencils outlined in the documentation, it will be specialized; otherwise the program will still run in pure Python.

An example using our stencil specializer's constructs is shown in Figure 2.

Specializer Writers

Specializer writers often start with an existing implementation of a solution, written in an ELL, for a particular problem type on particular hardware. Such solutions are devised by human experts who may be different from the specializer writer, e.g. numerical-analysis researchers or auto-tuning researchers. Some parts of the solution which remain the same between problem instances, or the same with very small changes, can be converted into *templates*, which are simply ELL source code with a basic macro substitution facility, supplied by [Mako], for inserting values into fixed locations or "holes" at runtime.

Other parts of the ELL solution may vary widely or in a complex manner based on the problem instance. For these cases, a better approach is to provide a set of rules for transforming the DAST of this type of problem in order to realize the optimizations present in the original ELL code. Finally, the specializer writer provides high-level transformation code to drive the entire process.

Specializer writers use Asp infrastructure to build their domain-specific translators. In Asp, we provide two ways to generate low-level code: templates and abstract syntax tree (AST) transformation. For many kinds of computations, using templates is sufficient to translate from Python to C++, but for others, phased AST transformation allows application programmers to express arbitrary computations to specialize.

In a specializer, the user-defined kernel is first translated into a Python AST, and analyzed to see if the code supplied by the application writer adheres to the restrictions of the specializer. Only code adhering to a narrow subset of Python, characterizing the embedded domain-specific language, will be accepted. Since specializer writers frequently need to iterate over ASTs, the Asp infrastructure provides classes that implement a visitor pattern on these ASTs (similar to Python's `ast.NodeTransformer`) to implement their specialization phases. The final phase transforms the DAST into a target-specific AST (e.g, C++ with OpenMP extensions). The Example Walkthrough section below demonstrates these steps in the context of the stencil kernel specializer.

Specializer writers can then use the Asp infrastructure to automatically compile, link, and execute the code in the final AST. In many cases, the programmer will supply several code variants, each represented by a different ASTs, to the Asp infrastructure. Specializer-specific logic determines which variant to run; Asp provides functions to query the hardware features available (number of cores, GPU, etc.). Asp provides for capturing and storing performance data and caching compiled code across runs of the application.

For specializer writers, therefore, the bulk of the work consists of exposing an understandable abstraction for specializer users, ensuring programs execute whether specialized or not, writing test functions to determine specializability (and giving the user meaningful feedback if not), and expressing their translations as phased transforms.

Currently, specializers have several limitations. The most important current limitation is that specialized code cannot call back into the Python interpreter, largely because the interpreter is not thread safe. We are implementing functionality to allow serialized calls back into the interpreter from specialized code.

In the next section, we show an end-to-end walkthrough of an example using our stencil specializer.

Example Walkthrough

In this section we will walk through a complete example of a SEJITS translation and execution on a simple stencil example. We begin with the application source shown in Figure 2. This simple two-dimensional stencil walks over the interior points of a grid and for each point computes the sum of the four surrounding points.

This code is executable Python and can be run and debugged using standard Python tools, but is slow. By merely modifying ExampleKernel to inherit from the StencilKernel base class, we activate the stencil specializer. Now, the first time the `kernel()` function is called, the call is redirected to the stencil specializer, which will translate it to low-level C++ code, compile it, and then dynamically bind the machine code to the Python environment and invoke it.

The translation performed by any specializer consists of five main phases, as shown in Figure 3:

1) Front end: Translate the application source into a domain-specific AST (DAST)
2) Perform platform-independent optimizations on the DAST using domain knowledge.

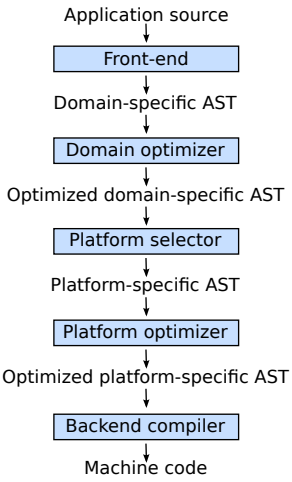

Fig. 3: *Pipeline architecture of a specializer.*

3) Select a platform and translate the DAST into a platform-specific AST (PAST).
4) Perform platform-specific optimizations using platform knowledge.
5) Back end: Generate low-level source code, compile, and dynamically bind to make available from the host language.

As with any pipeline architecture, each phase's component is reusable and can be easily replaced with another component, and each component can be tested independently. This supports porting to other application languages and other hardware platforms, and helps divide labor between domain experts and platform performance experts. These phases are similar to the phases of a typical optimizing compiler, but are dramatically less complex due to the domain-specific focus and the Asp framework, which provides utilities to support many common tasks, as discussed in the previous section.

In the stencil example, we begin by invoking the Python runtime to parse the `kernel()` function and produce the abstract syntax tree shown in Figure 4. The front end walks over this tree and matches certain patterns of nodes, replacing them with other nodes. For example, a call to the function `interior_points()` is replaced by a domain-specific StencilInterior node. If the walk encounters any pattern of Python nodes that it doesn't handle, for example a function call, the translation fails and produces an error message, and the application falls back on running the `kernel()` function as pure Python. In this case, the walk succeeds, resulting in the DAST shown in Figure 4. Asp provides utilities to facilitate visiting the nodes of a tree and tree pattern matching.

The second phase uses our knowledge of the stencil domain to perform platform-independent optimizations. For example, we know that a point in a two-dimensional grid has four neighbors with known relative locations, allowing us to unroll the innermost loop, an optimization that makes sense on all platforms.

The third phase selects a platform and translates to a platform-specific AST. In general, the platform selected will depend on available hardware, performance characteristics of the machine, and properties of the input (such as grid size). In this example we will target a multicore platform using the OpenMP framework. At this point the loop over the interior points is mapped down to nested parallel for loops, as shown in Figure 5. The Asp framework

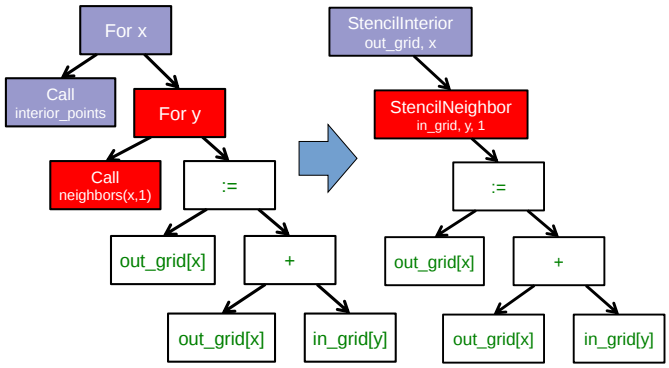

Fig. 4: *Left: Initial Python abstract syntax tree. Right: Domain-specific AST.*

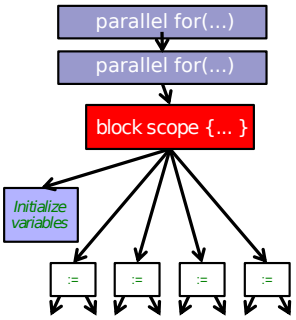

Fig. 5: *Platform-specific AST.*

provides general utilities for transforming arithmetic expressions and simple assignments from the high-level representation used in DASTs to the low-level platform-specific representation, which handles the body of the loop.

Because the specializer was invoked from the first call of the `kernel()` function, the arguments passed to that call are available. In particular, we know the dimensions of the input grid. By hardcoding these dimensions into the AST, we enable a wider variety of optimizations during all phases, particularly phases 4 and 5. For example, on a small grid such as the 8x8 blocks encountered in JPEG encoding, the loop over interior points may be fully unrolled.

The fourth phase performs platform-specific optimizations. For example, we may partially unroll the inner loop to reduce branch penalties. This phase may produce several ASTs to support run-time auto-tuning, which times several variants with different optimization parameters and selects the best one.

Finally, the fifth phase, the backend, is performed entirely by components in the Asp framework and the CodePy library. The PAST is transformed into source code, compiled, and dynamically bound to the Python environment, which then invokes it and returns the result to the application. Interoperation between Python and C++ uses the Boost.Python library, which handles marshalling and conversion of types.

The compiled `kernel()` function is cached so that if the function is called again later, it can be re-invoked directly without the overhead of specialization and compilation. If the input grid dimensions were used during optimization, the input dimensions must match on subsequent calls to reuse the cached version.

Results

SEJITS claims three benefits for productivity programmers. The first is *performance portability*. A single specializer can include code generation strategies for radically different platforms, and even multiple code variants using different strategies on the *same* platform depending on the problem parameters. The GMM specializer described below illustrates this advantage: a single specializer can produce code either for NVIDIA GPUs (in CUDA) or x86 multicore processors (targeting the Cilk Plus compiler), and the same Python application can run on either platform.

The second benefit is the ability to let application writers work with patterns requiring higher-order functions, something that is cumbersome to do in low-level languages. We can inline these functions into the emitted source code and let the low-level compiler optimize the solution using the maximum available information. Our stencil specializer, as described below, demonstrates this benefit; the performance of the generated code reaches 87% of the achievable memory bandwidth of the multicore machine on which it runs.

The third benefit is the ability to take advantage of auto-tuning or other runtime performance optimizations even for simple problems. Our matrix-powers specializer, which computes $\{x, Ax, A^2x, ..., A^kx\}$ for a sparse matrix A and vector x (an important computation in Krylov-subspace solvers), demonstrates this benefit. Its implementation uses a recently-developed *communication-avoiding* algorithm for matrix powers that runs about an order of magnitude faster than Python+SciPy (see performance details below) while remaining essentially API-compatible with SciPy. Beyond the inherent performance gains from communication-avoidance, a number of parameters in the implementation can be tuned based on the matrix structure in each individual problem instance; this is an example of an optimization that cannot easily be done in a library.

Stencil

To demonstrate the performance and productivity effectiveness of our stencil specializer, we implemented two different computational stencil kernels using our abstractions: a 3D laplacian operator, and a 3D divergence kernel. For both kernels, we run a simple benchmark that iteratively calls our specializer and measures the time for applying the operator (we ensure the cache is cleared in between calls). Both calculations are memory-bound; that is, they are limited by the available bandwidth from memory. Therefore, in accordance to the roofline model [SaWi09], we measure performance compared to measured memory bandwidth performance using the parallel STREAM [STREAM] benchmark.

Figure 6 shows the results of running our kernels for a 256^3 grid on a single-socket quad-core Intel Core i7-840 machine running at 2.93 GHz, using both the OpenMP and Cilk Plus backends. First-run time is not shown; the code generation and compilation takes tens of seconds (mostly due to the speed of the Intel compiler). In terms of performance, for the 3D laplacian, we obtain 87% of peak memory bandwidth, and 64% of peak bandwidth for the more cache-unfriendly divergence kernel, even though we have only implemented limited optimizations. From previous work [Kam10], we believe that, by adding only a few more tuning parameters, we can obtain over 95% of peak performance for these kernels. In contrast, pure Python execution is nearly three orders of magnitude slower.

In terms of productivity, it is interesting to note the difference in LoC between the stencils written in Python and the produced low-level code. Comparing the divergence kernel with its best-performing produced variant, we see an increase from five lines to over 700 lines--- an enormous difference. The Python version expresses the computation succinctly; using machine characteristics to express fast code requires expressing the stencil more verbosely in a low-level language. With our specialization infrastructure, programmers can continue to write succinct code and have platform-specific fast code generated for them.

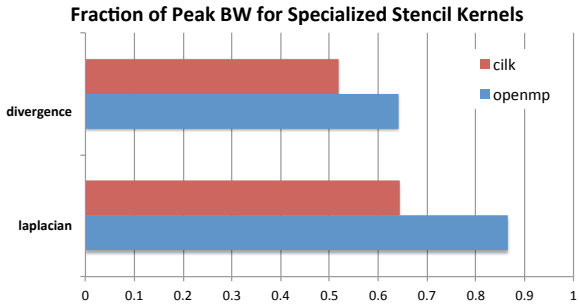

Fig. 6: *Performance as fraction of memory bandwidth peak for two specialized stencil kernels. All tests compiled using the Intel C++ compiler 12.0 on a Core i7-840.*

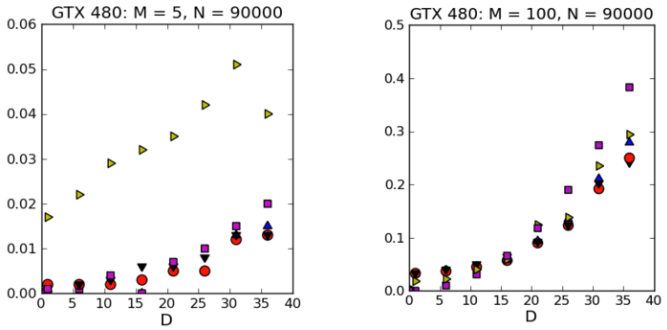

Fig. 7: *Runtimes of GMM variants as the D parameter is varied on an Nvidia Fermi GPU (lower is better). The specializer picks the best-performing variant to run.*

Gaussian Mixture Modeling

Gaussian Mixture Models (GMMs) are a class of statistical models used in a wide variety of applications, including image segmentation, speech recognition, document classification, and many other areas. Training such models is done using the Expectation Maximization (EM) algorithm, which is iterative and highly data parallel, making it amenable to execution on GPUs as well as modern multicore processors. However, writing high performance GMM training algorithms are difficult due to the fact that different code variants will perform better for different problem characteristics. This makes the problem of producing a library for high performance GMM training amenable to the SEJITS approach.

A specializer using the Asp infrastructure has been built by Cook and Gonina [Co10] that targets both CUDA-capable GPUs and Intel multicore processors (with Cilk Plus). The specializer implements four different parallelization strategies for the algorithm; depending on the sizes of the data structures used in GMM training, different strategies perform better. Figure 7 shows performance for different strategies for GMM training on an NVIDIA Fermi GPU as one of the GMM parameters are varied.

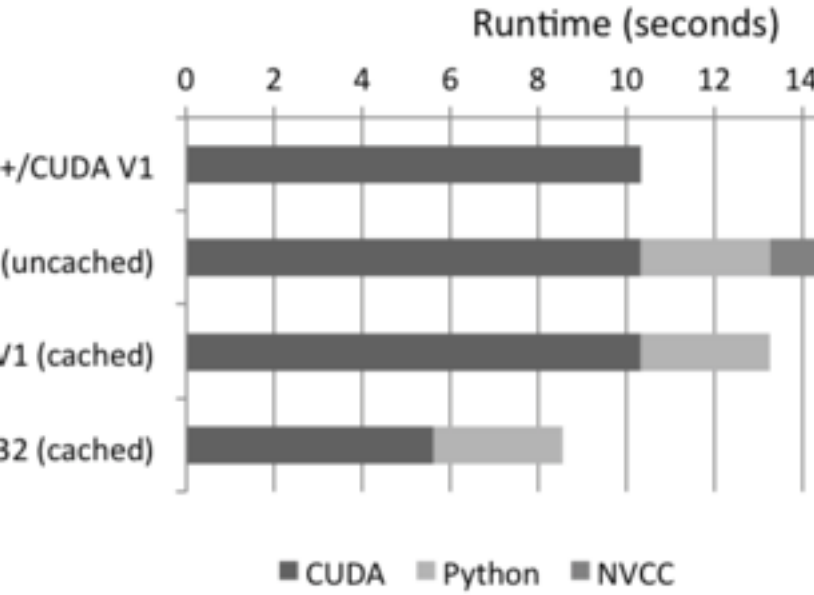

Fig. 8: *Overall performance of specialized GMM training versus original optimized CUDA algorithm. Even including specializer overhead, the specialized EM training outperforms the original CUDA implementation.*

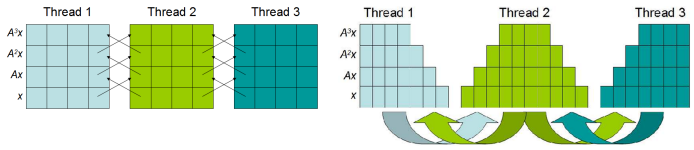

Fig. 9: *Left: Naive $A^k x$ computation. Communication required at each level. Right: Algorithm PA1 for communication-avoiding matrix powers. Communication occurs only after k levels of computation, at the cost of redundant computation.*

The specializer uses the best-performing variant (by using the different variants to do one iteration each, and selecting the best-performing one) for the majority of iterations. As a result, even if specialization overhead (code generation, compilation/linking, etc.) is included, the specialized GMM training algorithm outperforms the original, hand-tuned CUDA implementation on some classes of problems, as shown in Figure 8.

Matrix Powers

Recent developments in communication-avoiding algorithms [Bal09] (AF: need canonical citation here, as well as specific cite for Erin and Nick's CA-matrix powers presentation at Euro-SomethingOrOther) have shown that the performance of parallel implementations of several algorithms can be substantially improved by partitioning the problem so as to do redundant work in order to minimize inter-core communication. One example of an algorithm that admits a communication-avoiding implementation is matrix powers [Hoe10]: the computation $\{x, Ax, A^2x, ..., A^kx\}$ for a sparse matrix A and vector x, an important building block for communication-avoiding sparse Krylov solvers. A specializer currently under development enables efficient parallel computation of this set of vectors on multicore processors.

The specializer generates parallel communication avoiding code using the pthreads library that implements the PA1 [Hoe10] kernel to compute the vectors more efficiently than just repeatedly doing the multiplication $A \times x$. The naive algorithm, shown in Figure 9, requires communication at each level. However, for many matrices, we can restructure the computation such that communication only occurs every k steps, and before every superstep of k steps, all communication required is completed. At the cost of redundant computation, this reduces the number of communications required. Figure 9 shows the restructured algorithm.

The specializer implementation further optimizes the PA1 algorithm using traditional matrix optimization techniques such as cache and register blocking. Further optimization using vectorization is in progress.

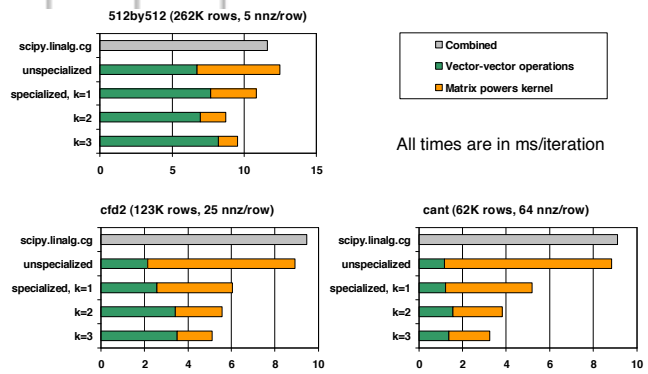

Fig. 10: *Results comparing communication-avoiding CG with our matrix powers specializer and SciPy's default solver, run on an Intel Nehalem machine.*

To see what kinds of performance improvements are possible using the specialized communication-avoiding matrix powers kernel, Morlan implemented a conjugate gradient (CG) solver in Python that uses the specializer. Figure 10 shows the results for three test matrices and compares performance against `scipy.linalg.solve` which calls the LAPACK `dgesv` routine. Even with just the matrix powers kernel specialized, the CA CG already outperforms the native solver routine used by SciPy.

Related Work

Allowing domain scientists to program in higher-level languages is the goal of a number of projects in Python, including SciPy [SciPy] which brings Matlab-like functionality for numeric computations into Python. In addition, domain-specific projects such as Biopython [Biopy] and the Python Imaging Library (PIL) [PIL] also attempt to hide complex operations and data structures behind Python infrastructure, making programming simpler for users.

Another approach, used by the Weave subpackage of SciPy, allows users to express C++ code that uses the Python C API as strings, inline with other Python code, that is then compiled and run. Cython [Cython] is an effort to write a compiler for a subset of Python, while also allowing users to write extension code in C. Another instance of the SEJITS approach is Copperhead [Cat09], which implements SEJITS targeting CUDA GPUs for data parallel operations.

The idea of using multiple code variants, with different optimizations applied to each variant, is a cornerstone of auto-tuning. Auto-tuning was first applied to dense matrix computations in the PHiPAC (Portable High Performance ANSI C) library [PHiPAC]. Using parametrized code generation scripts written in

Perl, PHiPAC generated variants of generalized matrix multiply (GEMM) with loop unrolling, cache blocking, and a number of other optimizations, plus a search engine, to, at install time, determine the best GEMM routine for the particular machine. After PHiPAC, auto-tuning has been applied to a number of domains including sparse matrix-vector multiplication (SpMV) [OSKI], Fast Fourier Transforms (FFTs) [SPIRAL], and multicore versions of stencils [KaDa09], [Kam10], [Tang11], showing large improvements in performance over simple implementations of these kernels.

Conclusion

We have presented a new approach to bridging the "productivity/efficiency gap": rather than relying solely on libraries to allow productivity programmers to remain in high-level languages, we package the expertise of human experts as a collection of code templates in a low-level language (C++/OpenMP, etc.) and a set of transformation rules to generate and optimize problem-specific ASTs at runtime. The resulting low-level code runs as fast or faster than the original hand-produced version.

Unlike many prior approaches, we neither propose a standalone DSL nor try to imbue a full compiler with the intelligence to "auto-magically" recognize and optimize compute-intensive problems. Rather, the main contribution of SEJITS is separation of concerns: expert programmers can express implementation optimizations that make sense only for a particular problem (and perhaps only on specific hardware), and package this expertise in a way that makes it widely reusable by Python programmers. Application writers remain oblivious to the details of specialization, making their code simpler and shorter as well as performance-portable.

We hope that our promising initial results will encourage others to contribute to building up the ecosystem of Asp specializers.

Acknowledgments

Henry Cook and Ekaterina Gonina implemented the GMM specializer. Jeffrey Morlan is implementing the matrix-powers specializer based on algorithmic work by Mark Hoemmen, Erin Carson and Nick Knight. Research supported by DARPA (contract #FA8750-10-1-0191), Microsoft Corp. (Award #024263), and Intel Corp. (Award #024894), with matching funding from the UC Discovery Grant (Award #DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

References

[ATLAS] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. Parallel Computing, vol. 27(1-2), pp. 3–35, 2001.

[Bal09] G. Ballard, J. Demmel, O. Holtz, O. Schwartz. Minimizing Communication in Numerical Linear Algebra. UCB Tech Report (UCB/EECS-2009-62), 2009.

[Biopy] Biopython. http://biopython.org.

[Cat09] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, A. Fox. SEJITS: Getting Productivity and Performance with Selective Embedded Just-in-Time Specialization. Workshop on Programming Models for Emerging Architectures (PMEA), 2009

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

[Co10] H. Cook, E. Gonina, S. Kamil, G. Friedland†, D. Patterson, A. Fox. CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications. 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar) 2011.

[Cython] R. Bradshaw, S. Behnel, D. S. Seljebotn, G. Ewing, et al., The Cython compiler, http://cython.org.

[FFTW] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. Proceedings of the IEEE 93 (2), 216–231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[Hoe10] M. Hoemmen. Communication-Avoiding Krylov Subspace Methods. PhD thesis, EECS Department, University of California, Berkeley, May 2010.

[KaDa09] K. Datta. Auto-tuning Stencil Codes for Cache-Based Multicore Platforms. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[Kam10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. International Parallel and Distributed Processing Symposium, 2010.

[Mako] Mako Templates for Python. http://www.makotemplates.org

[OSKI] OSKI: Optimized Sparse Kernel Interface. http://bebop.cs.berkeley.edu/oski.

[PHiPAC] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology and its Application to Matrix Multiply. LAPACK Working Note 111.

[PIL] Python Imaging Library. http://pythonware.com/products/pil.

[SaWi09] S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. Communications of the ACM (CACM), April 2009.

[SC08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. SC2008: High performance computing, networking, and storage conference, 2008.

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

[SPIRAL] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo. SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation".

[STREAM] The STREAM Benchmark. http://www.cs.virginia.edu/stream

[Tang11] Y.Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir Stencil Compiler. 23rd ACM Symposium on Parallelism in Algorithms and Architectures, 2011.

[Wonn00] D. Wonnacott. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. International Parallel and Distributed Processing Symposium, 2000.

N-th-order Accurate, Distributed Interpolation Library

Stephen M. McQuay‡*, Steven E. Gorrell‡

Abstract—The research contained herein yielded an open source interpolation library implemented in and designed for use with the Python programming language. This library, named smbinterp, yields an interpolation to an arbitrary degree of accuracy. The smbinterp module was designed to be mesh agnostic. A plugin system was implemented that allows end users to conveniently and consistently present their numerical results to the library for rapid prototyping and integration. The library includes modules that allow for its use in high-performance parallel computing environments. These modules were implemented using built-in Python modules to simplify deployment. This implementation was found to scale linearly to approximately 180 participating compute processes.

Index Terms—n-th-order accurate general interpolation, distributed calculation schemes, multiphysics simulation

Introduction and Background

As engineers attempt to find numeric solutions to large physical problems, simulations involving multiple physical models or phenomena, known as multiphysics simulations, must be employed. This type of simulation often involves the coupling of disparate computer codes. When modeling physically different phenomena the numeric models used to find solutions to these problems employ meshes of varying topology and density in their implementation. For example, the unstructured/structured mesh interfaces seen in the combustor/turbo machinery interface [Sha01], or the coupling of Reynolds-Averaged Navier-Stokes and Large Eddy Simulation (RANS/LES) codes in Computational Fluid Dynamics (CFD) [Med06]. A similar situation with disparate meshes arises in the analysis of helicopter blade wake and vortex interactions, as for example when using the compressible flow code SUmb and the incompressible flow code CDP [Hah06]. When this is the case, and the mesh elements do not align, the engineer must perform interpolation from the upstream code to the downstream code.

Frameworks exist that perform interpolation for multiphysics simulations. In general, frameworks of this variety try to solve two problems. First, the framework should rapidly calculate the interpolation. Secondly, the interpolation should be accurate.

CHIMPS (*Coupler for High-performance Integrated Multi-Physics Simulations*) is a Fortran API (with Python bindings) that implements an efficient Distributed Alternating Digital Tree for rapid distributed data lookup [Alo06], [Hah09]. By default, CHIMPS can only provide the user with linear (second-order accurate) interpolations. While CHIMPS can provide third-order and higher accurate interpolations, it is not automatic; higher-order interpolations are only performed if the engineer supplies the CHIMPS API with higher-order terms. If this information is unavailable, then CHIMPS can only yield linear interpolations.

Another interpolation framework exists that can perform automatic higher-order interpolation. AVUSINTERP [Gal06] (*Air Vehicles Unstructured/Structured Interpolation Tool*) is a tool that provides linear and quadratic interpolations requiring only the physical values at points in a donor mesh, i.e. no a priori knowledge of higher-order terms. While this framework implements a superior interpolation scheme to the tri-linear interpolation found in CHIMPS, AVUSINTERP was not implemented in a parallel fashion, nor does it allow for the engineer to arbitrarily choose the order of the interpolation past third-order accuracy.

The research presented herein describes the development of a library that is a union of the best parts of the aforementioned tools. Namely, this research provides a library, named smbinterp, that implements the interpolation of a physical value from a collection of donor points to a destination point and performs this interpolation to an arbitrary degree of accuracy. The library can perform this interpolation in both two- and three-dimensional space. Also, the library was designed and implemented to be used in a high-performance parallel computing environment. The smbinterp library is implemented as a python module that builds upon the numpy and scipy libraries and presents an API for use in multiphysics simulation integration. The library is released under the GPL, and project is available on github [smbinterp].

Method

The numerical method implemented in smbinterp was first proposed by Baker [Bak03]. This interpolation method comprises the adjustment of a linear interpolation by a least squares estimate of higher-order terms. The Baker interpolation of the physical value of interest (denoted q) to the point Ξ is defined by:

$$q(\Xi) = q_{linear}(\Xi) + f(\Xi), \tag{1}$$

where q_{linear} is the linear interpolation, and $f(\Xi)$ is an estimation of the higher-order error terms. The following explanation is specific to two-dimensional space; three-dimensional space is treated in [McQ11].

The participating geometry required to implement this method in two spatial dimensions is shown in figure 1. The blue points (R) and green points (S) represent points in a source mesh, and the red point Ξ is the point to which an interpolation is desired. $\triangle R$ represents a simplex that surrounds the destination point Ξ, and $S_{1..m}$ is a collection of extra points surrounding the simplex R. The triangles $A_1 - A_3$ represent the areas formed by Ξ and $\triangle R$.

* *Corresponding author: stephen@mcquay.me*
‡ *Brigham Young University*

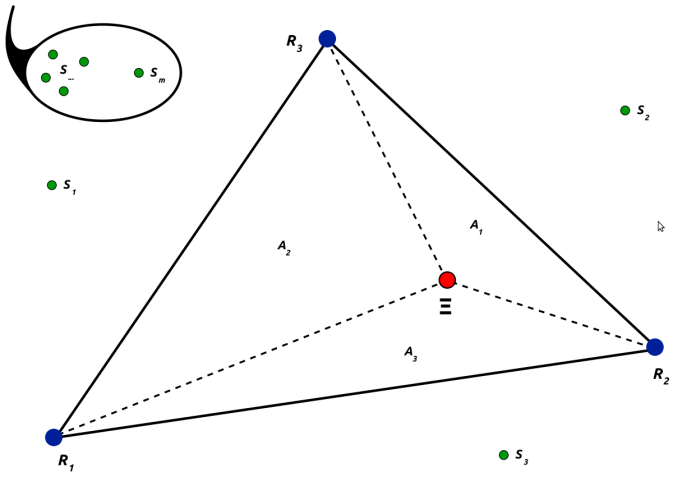

Fig. 1: *Planar Simplex used in Baker's Interpolation Scheme*

Barycentric coordinates , denoted $\phi_j(\Xi)$, are used to perform the linear interpolation. In geometric terms, the barycentric coordinates of a point in a simplex are the values of the normalized areas A_j/A_{total} opposite the vertex R_j in the simplex $\triangle R$.

The barycentric coordinates define the influence that each point in the simplex $\triangle R$ contributes to the linear interpolation. In other words, the ratio of A_j/A_{total} represents the influence from $0 \leq \phi \leq 1$ that $q(R_j)$ has over the linear interpolant. If $\Xi = R_j$, the value of $q_{linear}(\Xi)$ should then be influenced entirely by the known value of $q(R_j)$. If Ξ is placed in such a way as to give $\frac{A_1}{A_{total}} = \frac{A_2}{A_{total}} = \frac{A_3}{A_{total}}$, the value $q(R_j)$ at each point R_j contributes equally to the calculated value of $q_{linear}(\Xi)$.

The linear interpolant, which requires the simplex $\triangle R$ and Ξ as inputs, is defined as

$$q_{linear}(\triangle R, \Xi) = \sum_{j=1}^{N+1} q(R_j)\phi_j(\Xi), \tag{2}$$

where $N+1$ is the number of points in a simplex (3 in two-dimensional space, and 4 in three-dimensional space). The values of the basis functions $\phi_j(\Xi)$ is the only unknown in equation 2.

To solve for $\phi_j(\Xi)$ a system of linear equations will be defined involving the points in the simplex R_j, Ξ, and equation 2. If $q(\Xi)$ is a constant, $q_1 = q_2 = q_3 = q_{linear} = q_{constant}$, and equation 2 can be modified by dividing by $q_{constant}$, that is:

$$\phi_1 + \phi_2 + \phi_3 = 1. \tag{3}$$

Furthermore, the basis functions must be calculated so that equation 2 also interpolates geometric location of the point Ξ, hence

$$R_{1x}\phi_1(\Xi) + R_{2x}\phi_2(\Xi) + R_{3x}\phi_3(\Xi) = \Xi_x \tag{4}$$
$$R_{1y}\phi_1(\Xi) + R_{2y}\phi_2(\Xi) + R_{3y}\phi_3(\Xi) = \Xi_y. \tag{5}$$

The values of the basis functions $\phi_j(\Xi)$ can be found by solving the following system of linear equations involving equations 3, 4 and 5:

$$\begin{bmatrix} 1 & 1 & 1 \\ R_{1x} & R_{2x} & R_{3x} \\ R_{1y} & R_{2y} & R_{3y} \end{bmatrix} \begin{bmatrix} \phi_1(\Xi) \\ \phi_2(\Xi) \\ \phi_3(\Xi) \end{bmatrix} = \begin{bmatrix} 1 \\ \Xi_x \\ \Xi_y \end{bmatrix}, \tag{6}$$

which yields the values for $\phi_j(\Xi)$, providing a solution for equation 2.

At this point the first of two unknowns in equation 1 have been solved, however the least squares approximation of error terms $f(\Xi)$ remains unknown. If $q(\Xi)$ is evaluated at any of the points R_j in the simplex, then $q(R_j)$ is exact, and there is no need for an error adjustment at R_j, hence $f(\Xi) = 0$. Similarly, if $q(\Xi)$ is being evaluated along any of the opposite edges to R_t of the simplex $\triangle R$, the error term should have no influence from $\phi_t(\Xi)$, as $A_t = 0$. This condition is satisfied when expressing the error terms using the linear basis functions as

$$f(\Xi) = a\phi_1(\Xi)\phi_2(\Xi) + b\phi_2(\Xi)\phi_3(\Xi) + c\phi_3(\Xi)\phi_1(\Xi). \tag{7}$$

In equation 7 the three double products of basis functions are the set of distinct products of basis functions that are quadratic in the two spatial dimensions x and y, and zero when evaluated at each of the verticies in $\triangle R$. This term represents a third-order accurate approximation for the error up to and including the quadratic terms. This equation introduces three unknowns whose values must be solved, namely a, b, and c.

Recall that $S_k, k = 1, 2, \ldots, m$ is the set of m points surrounding Ξ that are not in the simplex R_j. A least squares system of equations is defined using the values of the basis functions at these points, the values of a linear extrapolation for each of those points using the simplex $\triangle R$, and the values of a, b, and c in equation 7. Define A as $(a, b, c)^T$. Applying least squares theory a, b, and c are found by inverting the following 3×3 matrix:

$$B^T A = B^T w. \tag{8}$$

The matrix B is defined using the identical basis function pattern as in equation 7. Denote $\phi_j(S_k)$ as the value of ϕ_j evaluated using equation 2 and the data point S_k (in lieu of Ξ). The matrix B in equation 8 is thus defined:

$$B = \begin{bmatrix} \phi_1(S_1)\phi_2(S_1) & \phi_2(S_1)\phi_3(S_1) & \phi_1(S_1)\phi_3(S_1) \\ \phi_1(S_2)\phi_2(S_2) & \phi_2(S_2)\phi_3(S_2) & \phi_1(S_2)\phi_3(S_2) \\ \vdots & \vdots & \vdots \\ \phi_1(S_m)\phi_2(S_m) & \phi_2(S_m)\phi_3(S_m) & \phi_1(S_m)\phi_3(S_m) \end{bmatrix}. \tag{9}$$

The value of $q(S_k)$ is known a priori (values of q at each point S_k in the donor mesh). The value of $q_{linear}(S_k)$ (the linear extrapolant) can also be calculated using equation 2. Define w in equation 8 as

$$w = \begin{bmatrix} q(S_1) - q_{linear}(\triangle R, S_1) \\ q(S_2) - q_{linear}(\triangle R, S_3) \\ \vdots \\ q(S_m) - q_{linear}(\triangle R, S_m) \end{bmatrix}. \tag{10}$$

Equation 8 is populated with the information from each of the surrounding points in S_k, then the unknown A can be calculated. Knowing A, equation 7 is evaluated for $f(\Xi)$. Subsequently the previously calculated value of $q_{linear}(\Xi)$ and the recently calculated value of $f(\Xi)$ are used to solve equation 1 for $q(\Xi)$.

There exist known limitations to this least squares-based interpolation method. First a change in vertex stencil will generally yield a discontinuity in interpolation results. While this property makes this method insufficient for graphical applications, it has been shown to yield sufficiently accurate results to be used in engineering applications [Bak03], [Gal06].

Secondly, while solutions to the linear system in equation 2 are well-behaved, certain vertex configurations can lead to a singular system of equations in equation 7. These pathological vertex configurations occur when more than $n-1$ of the extra points lie on one extended edge of the simplex $\triangle R$ [Bak03]. If this

occurs, the covariance matrix $B^T B$ will be singular, the solution will not be unique, and the error approximation will not generally aid in improving the interpolation.

Extension of this method into three dimensions is non-trivial, and is explained in depth in [McQ11]. A pattern exists to define any error approximation function $f(\Xi)$ and covariance matrix $B^T B$ parametrized by order of approximation and dimension. Define ν as the desired order of accuracy less one (i.e. for cubic interpolation ν is 3). As defined above, N is the spatial degree. The pattern for the combinations of basis functions that are used to define $f(\Xi)$ is collection of ν-th ordered combinations of $N+1$ basis functions ϕ_j that are unique and non-duplicate, triplicate, etc. The following code implements this pattern:

```
from itertools import product

@memoize
def pattern(simplex_size, nu):
  r = []
  for i in product(xrange(simplex_size),
                                  repeat = nu):
    if len(set(i)) !=1:
      r.append(tuple(sorted(i)))
  unique_r = list(set(r))
  return unique_r
```

The dynamic calculation of the basis function pattern in this fashion is powerful, in that it can be calculated for any arbitrary ν, and for any spatial dimension (although only N of 2 and 3 are dealt with herein). However, for each point Ξ the calculation of the pattern must be performed once for the calculation of $f(\Xi)$ and once per extra point S_k participating in the current interpolation for each row in the B matrix. There is only one valid pattern per set of inputs N and ν, which must both remain constant throughout a single interpolation. The calculation of the pattern is a computationally intensive operation, and so a caching mechanism has been implemented in smbinterp that only calculates the pattern if it has not been previously calculated. This concept is known as memoization, and is implemented using the following function wrapper:

```
from functools import wraps

def memoize(f):
  cache = {}
  @wraps(f)
  def memf(simplex_size, nu):
    x = (simplex_size, nu)
    if x not in cache:
      cache[x] = f(simplex_size, nu)
    return cache[x]
  return memf
```

Baker's method gives a reasonable interpolation solution for a general cloud of points. However, the method suggested by Baker for the vertex selection algorithm for the terms $\triangle R$ and S_k consists of simply selecting the points nearest Ξ. While this is the most general point selection algorithm, it can lead to the aforementioned pathological vertex configurations. This configuration is prevalent when the source mesh is composed of a regular grid of vertices, and must be addressed if the method is to yield a good interpolation.

Furthermore a mesh may have been designed to capture the gradient information, and therefore the mesh topology should be respected. Simply selecting the closest points to Ξ would yield inferior results. By selecting the more topologically (according to the mesh) adjacent points the information intended to be captured in the mesh's design will be preserved.

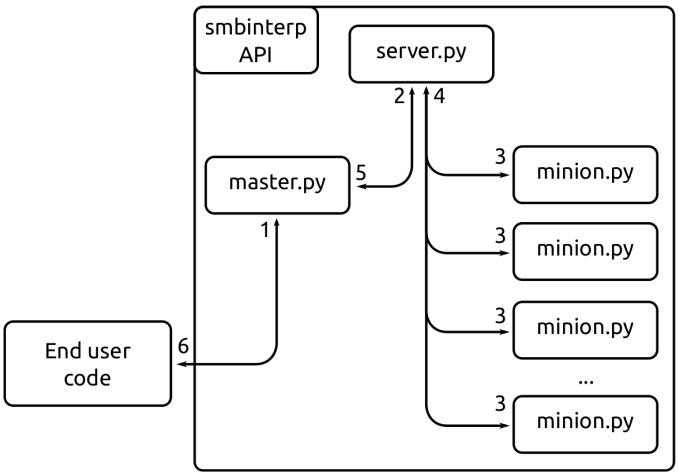

Fig. 2: *Flowchart of the Parallelization Architecture*

A plugin architecture was implemented in smbinterp which yields the requisite flexibility needed to avoid the pathological grid configurations and gives the engineers complete control over the point selection algorithms. The base class for all grid objects that desire to use the interpolation methods is defined as follows:

```
class grid(object):
  def __init__(self, verts, q):
    self.verts = np.array(verts)
    self.tree = KDTree(self.verts)

    self.q = np.array(q)

    self.cells = {}
    self.cells_for_vert = defaultdict(list)

  def get_containing_simplex(self, Xi):
    # ...
    return simplex

  def get_simplex_and_nearest_points(self,
                          Xi, extra_points = 3):
    # ...
    return simplex, extra_points
```

The cells and cells_for_verts data structures are used when searching for a containing simplex. The structures are populated with connectivity information before a round of interpolations. The method employed in the default implementation for the location of the containing simplex in an upstream mesh is straight forward: first the spatial tree structure is used to find the location of the nearest vertex to the point of interest, then the cells are recursively visited in topologically adjacent order and tested for inclusion of the point Ξ.

The selection of the extra points S_k is also implemented in the base grid class. The default algorithm simply queries the kdtree structure for $(N+1)+m$ points and discards the points that are already in the simplex $\triangle R$.

Plugins are defined as classes that inherit from the base grid object, and that implement the requisite functionality to populate the cells and cells_for_vert data structures. If either of the default simplex and vertex selection methods do not provide the desired functionality they could be overridden in the derived class to provide a more tuned $\triangle R$ and S_k selection algorithms. This gives engineers complete control over point selection and makes the interpolation library mesh agnostic.

A parallel mechanism for calculating $q(\Xi)$ was implemented

in smbinterp. As is illustrated in figure 2, a stream of requested interpolations are presented to a queuing mechanism that then distributes the task of calculating the interpolations to a set of minions.

The server.py application implements the four queues required to implement this method: a queue for tasks to be performed, a queue for results, and two queues for orchestrating the control of a round of interpolations between a master and a set of minions. Masters and minions authenticate and connect to these four queues to accomplish the tasks shown in the flowchart in figure 2. The master.py script is responsible for orchestrating the submission of interpolations and events associated with starting and stopping a set of interpolations. Each of the minions has access to the entire domain and are responsible for performing the interpolations requested by the end user.

The crux of the solution lies in providing the minions with a steady stream of work, and a pipeline into which the resultant interpolations can be returned. The mechanism developed in smbinterp uses built-in Python modules to minimize the deployment expense. The multiprocessing module provides a manager class which facilitates the access of general objects to authenticated participants over a network. The built-in Queue objects, which implement a multi-producer, multi-consumer first-in-first-out queue, are presented to the minions and masters using the functionality in the manager class.

Results and Discussion of Results

The root mean square (RMS) of the errors was used to determine the accuracy of the smbinterp module. A continuous function whose values varied smoothly in the test domain was required to calculate the error; the following equation was used:

$$q(x,y) = \left(\sin(x\pi)\cos(y\pi)\right)^2. \tag{11}$$

A plot of this function is found in figure 3. Each error ε_i was calculated as the difference between the actual value (from equation 11) and calculated interpolations (at each point in the destination domain using smbinterp), or $\varepsilon_i(\Xi) = q_{exact}(\Xi) - q_{calculated}(\Xi)$.

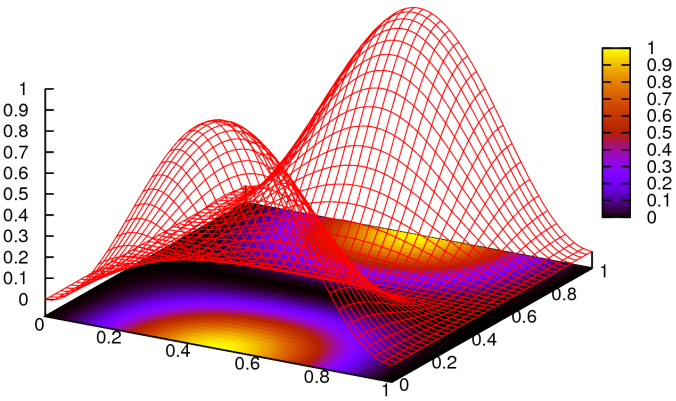

Fig. 3: *Plot of Equation 11*

A mesh resolution study was performed to determine how the RMS of error varied with mesh density. The source mesh was generated using gmsh, and the lowest-resolution mesh is shown in figure 4. The results of this study are show in figure 5. A collection of 1000 random points were used as the destination for interpolation.

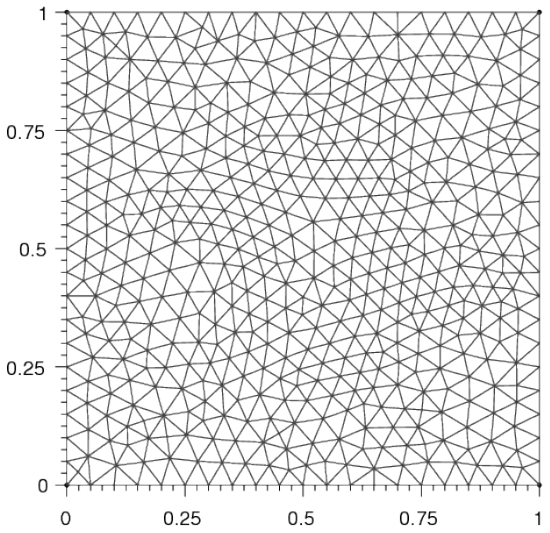

Fig. 4: *Lowest-resolution test mesh*

Figure 5 plots the relationship between mesh spacing and RMS of error of all interpolations in the collection of destination vertexes. The x-axis represents the spacing between the regular mesh elements. The y-axis was calculated by performing interpolation from each resolution of mesh to a static collection of random points. The lines in each plot are representative of the slope that each collection of data should follow if the underlying numerical method is truly accurate to the requested degree of accuracy. As an example, the collection of points for ν of 2 should be third-order accurate, and should follow a line with slope of 3; this is closely demonstrated in the plots.

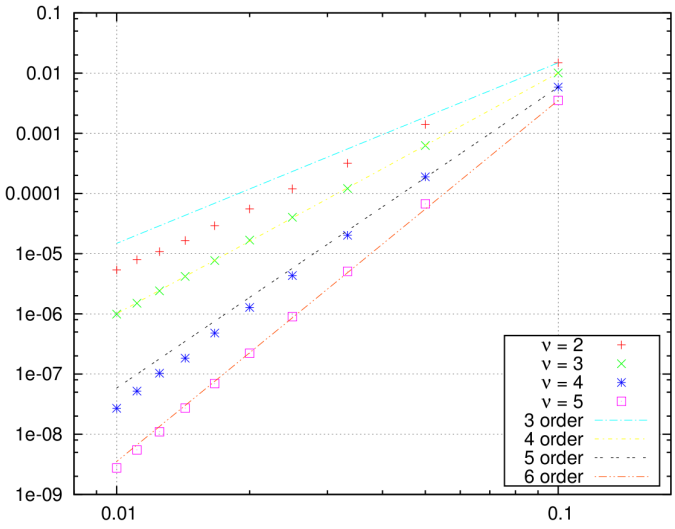

Fig. 5: *RMS of Error vs. Mesh Spacing*

Figure 5 shows the results of the resolution study for the two-dimensional test case meshes. The three dimensional test case meshes yielded similar results and are presented in [McQ11]. As the meshes were refined the RMS of error decreased. The fourth- and sixth-order results (ν of 3 and 5) matched the slope lines almost exactly, whereas the third- and fifth-order results were slightly lower than expected for that level of accuracy.

As mesh element size decreased, the RMS of error decreased

as well. The RMS of error for the highest ν decreased more than that of the lowest ν. The RMS of error of the most coarse mesh (far right) ranges within a single order of magnitude, whereas the RMS of errors at the most fine spacing (far left) span four orders of magnitude. The results exhibit a slight banding, or unevenness between each order. Also, the data very closely matches the plotted lines of slope, indicating that the order of accuracy is indeed provided using this numerical method.

The rate at which error decreases as the average mesh element size decreases in figure 5 is indicative of the order of accuracy of the numerical method implemented in smbinterp. There is slight banding for the two-dimensional meshes between quadratic and cubic interpolation, and again for quartic an quintic interpolation. While this indicates that the method does not perfectly interpolate to those orders of accuracy, in general increasing the ν parameter of the smbinterp library provides a more accurate interpolation. Furthermore, the cases where the points diverge from the slope of appropriate order, the divergence occurs in a favorable direction (i.e. less error). Also, the fine meshes experience a more significant decrease in RMS of error than the coarse meshes while increasing the order of approximation, ν. While this is an intuitive result, it emphasizes the notion that mesh density should be chosen to best match the underlying physical systems and to provide optimally accurate results.

The parallel algorithm employed by smbinterp was found to scale quasi-linearly to approximately 180 participating minion.py processes. Speedup is defined as the ratio of time to execute an algorithm sequentially (T_1) divided by the time to execute the algorithm with p processors [WSU], or $S_p = \frac{T_1}{T_p}$. A parallel algorithm is considered to have ideal speedup if $S_p = p$.

A more meaningful parameter for instrumenting the performance of a parallel algorithm is known as the efficiency of the algorithm, denoted E_p. Efficiency of a parallel algorithm is defined as the speedup divided by the number of participating processors, or $E_p = \frac{T_p}{p}$. The efficiency of an algorithm ranges from 0 to 1, and is shown for smbinterp in figure 6.

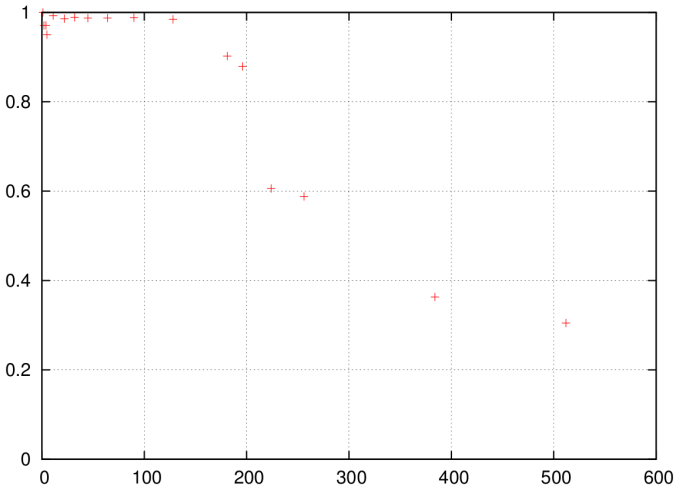

Fig. 6: *Efficiency (E_p) of the Parallel Algorithm*

The parallelization algorithm employed by the smbinterp library has near-linear speedup up to approximately 128 participating minions. It has an efficiency above 90 percent up to 181 participating nodes, but the efficiency drops substantially when using more minions. If an algorithm does not have an efficiency of 1, it is usually indicative of communication overhead or bottlenecks of some form. It was observed that the cpu utilization of the server.py script increased linearly up to 181 minions (CPU utilization of 200%), but then did not increase past that point. The implementation of the server.py script represents the bottleneck of this implementation.

Conclusions

The smbinterp module was developed to provide a high-performance interpolation library for use in multiphysics simulations. The smbinterp module provides an interpolation for a cloud of points to an arbitrary order of accuracy. It was shown, via a mesh resolution study, that the algorithm (and implementation thereof) provides the the end user with the expected level of accuracy, i.e. when performing cubic interpolation, the results are fourth-order accurate, quartic interpolation is fifth-order accurate, etc.

The smbinterp module was designed to be mesh agnostic. A plugin system was implemented that allows end users to conveniently and consistently present their numerical results to the library for rapid prototyping and integration.

The smbinterp module was designed with parallel computing environments in mind. The library includes modules that allow for its use in high-performance computing environments. These modules were implemented using built-in Python modules to simplify deployment. This implementation was found to scale linearly approximately 180 participating compute processes. It is suggested to replace the queuing mechanism with a more high-performance queuing library (e.g. ØMQ) and a more advanced participant partitioning scheme to allow the library to scale past this point.

Acknowledgments

The authors thank Marshall Galbraith for his friendly and crucial assistance which helped clarify the implementation of the numerical method used herein. The authors are especially grateful to have performed this research during a time when information is so freely shared and readily available; they are indebted to all of the contributors to the Python and Scipy projects. The authors would also like to acknowledge the engineers in the aerospace group at Pratt & Whitney for the contribution of the research topic and for the partial funding provided at the beginning of this research.

References

[Sha01] S. Shankaran et al., *A Multi-Code-Coupling Interface for Combustor/Turbomachinery Simulations*, AIAA Paper 2001-974, 39th AIAA Aerospace Sciences Meeting and Exhibit January 8–11, 2001.

[Med06] G. Medic et al., *Integrated RANS/LES computations of turbulent flow through a turbofan jet engine*, Center for Turbulence Research Annual Research Briefs, 2006, pp. 275-285.

[Hah06] S. Hahn et al., *Coupled High-Fidelity URANS Simulation for Helicopter Applications}*, Center for Turbulence Research Annual Research Briefs, 2006, pp 263-274.

[Alo06] J. Alonso et al., *CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations*, AIAA Paper 2006-5274, 42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, Sacramento, CA, July 2006.

[Hah09] S. Hahn et al., *Extension of CHIMPS for unstructured overset simulation and higher-order interpolation*, AIAA Paper 2009-3999, 19th AIAA Computational Fluid Dynamics, San Antonio, Texas, June 22-25, 2009

[Gal06] M. Galbraith, J. Miller. *Development and Application of a General Interpolation Algorithm*, AIAA Paper 2006-3854, 24th AIAA Applied Aerodynamics Conference, San Francisco, California, June 5-8, 2006

[Bak03] Baker, T. *Interpolation from a Cloud of Points*, in 12th International Meshing Roundtable, Santa Fe, NM, 2003, pp. 55-63.

[McQ11] S. M. McQuay, *SMBInterp: an Nth-Order Accurate, Distributed Interpolation Library,* M.S. thesis, Mech. Eng., Brigham Young University, Provo, UT, 2011.

[WSU] http://en.wikipedia.org/wiki/Speedup

[smbinterp] https://github.com/smcquay/smbinterp

Google App Engine Python

Douglas A. Starnes‡*

Abstract—In recent years, one of the fastest growing trends in information technology has been the move towards cloud computing. The scalable concept of computing resources on demand allows applications to dynamically react to increased usage instead of having to keep resources in reserve that are often not in use but are still paid for. There are several popular entrants into this market including Google App Engine. Modeled after Google's own architecture for building applications, Google App Engine (GAE) provides a scalable solution for web-based applications and services including data storage, communications, application deployment and monitoring, and management tools. With GAE, developers have the option of writing applications using an API exposed to Python. The same benefits of using Python in other applications are available in the cloud.

Index Terms—cloud computing, web, google, application development

Overview

To assist developers in writing their applications, GAE provides a variety of frameworks and services which will be discussed later. An broad look at the entire application structure is in order first.

GAE can be broken down into three pieces: your application, the SDK and tools, and the server itself. The application is configured by a set of files using the YAML ("Yet Another Markup Language" or the recursive "YAML Ain't Markup Language") specification. The main configuration file must be named app.yaml and contains metadata about the application such as the name of the application and the version number. The application name must be unique across all of GAE. Google provides a service to look up available application names at registration. The application name will also be a subdomain of appspot.com where the application's default version will be. Also in the app.yaml file is a set of mappings. Both scripts and static content can be mapped to URL endpoints. There are built-in modules and other options such as security settings which can be configured in app.yaml as well.

Several important tools are provided by the SDK for GAE. Primary among these are appcfg and dev_appserver. The appcfg tool is used to deploy applications to the server based on the configuration in the app.yaml file. Other functions the appcfg tool includes are to download the code and server logs for an application and to manage cron jobs and datastore indexes. The other important tool is dev_appserver. The dev_appserver tool is a local development server that emulates the services of GAE. Not all services are equally emulated. For example, the cron service does not run locally. Also, the email service simply dumps a trace of the message contents to the logs. Two other tools of note are the bulkloader and remote_api_shell. The bulkloader is useful for migrating large amounts of existing data to GAE all at one time. The remote_api_shell provides an interactive Python session with the live datastore.

The GAE server exposes the API and services that your application is built upon and is emulated by the local development server. Among the most important services is the datastore. GAE also exposes an HTTP stack in webapp. Other services such as the task queue, federated identity with Google and GMail accounts, email and XMPP messaging exist.

* Corresponding author: douglas@poweredbyalt.net
‡ University of Memphis - Institute For Intelligent Systems

webapp

The webapp framework is a Python library to handle network traffic using the HTTP protocol. With webapp, the application defines request handlers that are mapped to endpoints using both the app.yaml settings as well as a set of URL routes defined within the request handler itself. Each handler is a class derived from a base RequestHandler class provided by webapp. The RequestHandler class primarily defines methods for handling the different HTTP methods (e.g. get, post). Also within the RequestHandler class are objects representing the request and response for the current HTTP transmission so the application can retrieve the URI requested or the status code and also return data to the caller or redirect to another resource. When the blobstore is needed, webapp defines a group of handlers to be used specifically with uploading data to the blobstore.

```
from google.appengine.ext import webapp
    from google.appengine.ext.webapp import util

    class MyHandler(webapp.RequestHandler):
        def get(self):
                # get data from the datastore,
                    #render a template ...
                    self.response.out.write(data)

    if __name__ == '__main__':
        application =
                webapp.WSGIApplication(
                            [('/.*', MyHandler)]
                    )
            util.run_wsgi_app(application)
```

Data Storage

GAE gives developers several avenues of data storage. For long term storage, data is persisted into the datastore and blobstore. Short term storage is available in the memcache.

The datastore is one of the most prominent services of GAE. The datastore is intended for structured table storage. While the

datastore uses tables, it is not a relational database. The non-relational nature of the the datastore puts it in the category of "NoSQL" in the opinions of some. Unlike the popular NoSQL databases MongoDB and CouchDB, the datastore is referred to as "column oriented". MongoDB and CouchDB are document-oriented and schema-less and do not define a formal data model for the documents. The GAE datastore requires a data model but is more flexible. The tables - or "kinds" to use GAE terms - are defined by a classes derived from a base Model class from the db module. This Model class has all of the CRUD (create, read, update and delete) operations built in. The developer then has only to define the properties of the kind along with data types, and options such as if the property is required and its length.

```
from google.appengine.ext import db

class Product(db.Model):
    name = db.StringProperty(required=True)
        price = db.FloatProperty(default=0.99)
        suppliers = db.StringListProperty()
```

Please notice the unusual yet useful StringListProperty. The GAE datastore can have columns that are lists. There is also a ListProperty type for non-string values.

Adding a kind to the datastore needs no preparation from the developer, other than to define the kind. Simply call the put method on an instance of the kind and the first time the datastore will take care of all the housekeeping to define the kind in the datastore.

```
product = Product()
# initialize required properties
product.put()
```

While the datastore does require a data model, the model can be defined or extended after the kind class has been defined. The Expando class in the db module has this capability.

For querying the datastore, there exists a simple language called GQL (Google Query Language) that as the name suggests, is similar to SQL. GQL has several limitations, mainly that it can only retrieve data. All other operations (insert, update, delete) must be performed programmatically. Selects can be performed programmatically as well but GQL provides a simpler way. GQL also has a few enhancements over SQL such as bound parameters that are referenced by position or name. The following code demonstrates this:

```
db.GqlQuery(
    "select * from Product where name = :1",
        "Gadget")

db.GqlQuery(
    "select * from Product where price <= :price_point",
        price_point=1.99)
```

The datastore also supports indexing. For simple queries, indexes are constructed by the datastore. Simple queries include those with only equality comparisons, and those with only one inequality comparison or one sort orders. Other queries must be defined manually. These are in another YAML configuration file called index.yaml. When the index.yaml file is deployed along with the app, the datastore will examine the definitions and build the indexes. Progress can be monitored through the online web control panel for GAE. Manually defining an index is not a lot of work because if GAE requires an index for a query that has not been defined, it will return an error along with a suggested definition. TextProperty and BlobProperty columns cannot be indexed. As is the case with relational databases, it is best to define indexes before GAE notices.

A complement to the datastore is the blobstore. The blobstore is intended to persist unstructured binary data such as images. A blob has a maximum size of 2 gigabytes. These blobs are immutable. Once created, blobs can be read or deleted but not modified. There is also no way to reference a blob from the datastore. The datastore does support the BlobProperty column type but that blob is stored in the datastore. The blobstore is separate. Managing blobs can only be done through the online control panel for the app. Here the user can view, download and delete blobs. The only way to get blobs into the datastore is through a web form. There is experimental support for writing files to the blob. This would be useful for creating blobs in response to a cron job or something else that does not require a user to start it.

For short term storage of small values exists the memcache. Entries in the memcache are simple key/value pairs. Entries stored in the memcache will eventually expire. By default, GAE keeps entries as long as there is enough memory. If an application begins to consume a lot of memory, older entries will be freed to make room for newer ones. Also, in the event of a system failure, entries will not be retained as they are stored in memory, not persisted to disk. Memcache values can be no more than 1MB in size.

Task Management

Most request to GAE should be short lived. There is a 30 second limit on HTTP requests. In the logs, requests are flagged as lengthy when they begin to exceed about 500ms. Anything more than 30 seconds will throw an exception and the request will be terminated. If a longer running task is needed, an application can start a background worker. These have a maximum time limit of 10 minutes. Creating a new worker is easy: call a method on the taskqueue API and pass it the endpoint of the worker along with an object of any parameters. There does not appear to be a mechanism using the default method of processing queues to have a callback method for notification of when a worker is complete. Using pull queues, an application can take over the method of processing queues itself. With pull queues there is a REST API so that the processing can be external to GAE.

If the task requires even more time, it can be handled by a backend. A backend is a separate GAE instance which has no constraints on time to run. Furthermore, backends are more configurable and have access to more resources such as memory and CPU. For applications using pull queues, tasks can be passed to a backend. Backends can consume large amounts of resources so there is an extra charge for them. They are billed in 15 minute increments up until the backend has been idle for 15 minutes. A backend can be resident meaning it must be shut down explicitly or dynamic meaning it will start in response to code and shut down after it have been idle for 15 minutes. Neither seems to have an impact on the hourly rate though. Backends do not scale automatically as normal GAE instances do. The number of backends is allocated explicitly in a configuration file (backends.yaml).

The last method of background processing GAE has is cron jobs. Cron jobs on GAE work similar to cron jobs on UNIX-based systems. In GAE, a configuration file named cron.yaml defines the tasks to be run. The cron.yaml file has entries for the endpoint of the task and the frequency of the task. The task frequency is expressed using a format that is more verbose than the UNIX crontab format but is also easier to interpret. For example to have a task run every 24 hours:

"every 24 hours"

is the expression to use in the cron.yaml file. More specific expressions such as the following are also possible:

"1st tue of november 0:00"

There are two differences to consider when using cron jobs on GAE. First is that cron jobs do not run in the local development environment. You can view what jobs are defined and access the endpoints for them but the schedule will not be followed. Second, cron jobs always run on the default version of the application. If you define a cron job in a development version of an app, it will not be run to avoid conflicts with the default version. Cron jobs always call endpoints using HTTP GET.

To remove a cron job from an application, remove its entry from cron.yaml and deploy the application. To remove all jobs from an application, deploy a cron.yaml without any job entries.

Application Environment

On the server, GAE hosts and serves applications at a subdomain of appspot.com that is the same as the registered application name in the app.yaml file. The official version of Python on GAE is 2.5.2. However, experience has shown that using 2.5.4 has no problems. There have been accounts of applications targeting 2.6 and 2.7 working with the development server. While these applications might run locally, any code that is specific to the later versions will not run on GAE. The development environment makes no attempt to ensure that the supported version of Python is being used. The Python Standard Library is available with a few exceptions. First several libraries such as PyYAML and simplejson have been added. Also, for security reasons, there are libraries which are not allowed such as marshal and socket. Importing these libraries will not cause an exception but import nothing so is the equivalent of a null operation. Any pure Python code that does not have dependencies on C extensions within the constraints above will run on GAE.

The SDK has versions for Windows, Mac OS X and Linux. On Windows and OS X there is a GUI launcher to access some of the command line tools. The launcher is helpful if more than one version of Python is running on the machine. The launcher has a setting to specify the location of Python to use with GAE on the development server. The development server also gives a console in the web control panel to run Python statements against the currently running application. The GAE SDK is also open source although modifying the source code will most likely result in an application which will fail to run correctly on the remote server.

Other Notable Features

GAE has a number of unique features that are outside the scope of this paper. The following are a few that deserve to be mentioned. First, it is worth knowing that there are two other languages supported by GAE, Java and Go. Go is a programming language developed by Google and described as a hybrid between C++ and Python. Go is experimental at the time of this writing. The features between the Python and Java runtimes are close to being identical. Python has a small lead as it was the first language for GAE. GAE also supports federated identity using Google accounts so anyone with a GMail or Google account can be authenticated using GAE with only a few lines of code. Also included with GAE is a profiling package called AppStats. Using this tool, very detailed timelines about the requests the application processes can be analyzed. The call stacks are also recorded and can be navigated through a web-based interface. AppStats works on both the remote server and local development environments. Finally, a new experimental library called ProtoRPC was recently added to the SDK. ProtoRPC simplifies the workflow for creating REST-based web services using GAE.

Conclusion

Benefits of using Python on GAE let developers prototype and develop applications in the cloud using familiar web technologies. A notable benefit is that GAE will support a free quota to test applications on the server before enabling billing. In addition, GAE integrates very well with the Python libraries for GData to access services such as Google Finance, Google Spreadsheets, Google Sites, and Picasa. Finally, there are several maintained application frameworks running on top of GAE that extend its functionality.

Google App Engine is a very thorough platform with many features. This paper has discussed only a few of them. To get more information, the reader is encouraged to visit http://code.google.com/appengine to register for a free developer account, get the documentation, SDK and sample code as well as information about the new pricing model for later this year when GAE leaves beta.

Time Series Analysis in Python with statsmodels

Wes McKinney[¶*], Josef Perktold[‡], Skipper Seabold[§]

Abstract—We introduce the new time series analysis features of scikits.statsmodels. This includes descriptive statistics, statistical tests and several linear model classes, autoregressive, AR, autoregressive moving-average, ARMA, and vector autoregressive models VAR.

Index Terms—time series analysis, statistics, econometrics, AR, ARMA, VAR, GLSAR, filtering, benchmarking

Introduction

Statsmodels is a Python package that provides a complement to SciPy for statistical computations including descriptive statistics and estimation of statistical models. Beside the initial models, linear regression, robust linear models, generalized linear models and models for discrete data, the latest release of scikits.statsmodels includes some basic tools and models for time series analysis. This includes descriptive statistics, statistical tests and several linear model classes: autoregressive, AR, autoregressive moving-average, ARMA, and vector autoregressive models VAR. In this article we would like to introduce and provide an overview of the new time series analysis features of statsmodels. In the outlook at the end we point to some extensions and new models that are under development.

Time series data comprises observations that are ordered along one dimension, that is time, which imposes specific stochastic structures on the data. Our current models assume that observations are continuous, that time is discrete and equally spaced and that we do not have missing observations. This type of data is very common in many fields, in economics and finance for example, national output, labor force, prices, stock market values, sales volumes, just to name a few.

In the following we briefly discuss some statistical properties of the estimation with time series data, and then illustrate and summarize what is currently available in statsmodels.

Ordinary Least Squares (OLS)

The simplest linear model assumes that we observe an endogenous variable y and a set of regressors or explanatory variables x, where y and x are linked through a simple linear relationship plus a noise or error term

$$y_t = x_t\beta + \varepsilon_t$$

* Corresponding author: wesmckinn@gmail.com
¶ Duke University
‡ University of North Carolina, Chapel Hill
§ American University

In the simplest case, the errors are independently and identically distributed. Unbiasedness of OLS requires that the regressors and errors be uncorrelated. If the errors are additionally normally distributed and the regressors are non-random, then the resulting OLS or maximum likelihood estimator (MLE) of β is also normally distributed in small samples. We obtain the same result, if we consider consider the distributions as conditional on x_t when they are exogenous random variables. So far this is independent whether t indexes time or any other index of observations.

When we have time series, there are two possible extensions that come from the intertemporal linkage of observations. In the first case, past values of the endogenous variable influence the expectation or distribution of the current endogenous variable, in the second case the errors ε_t are correlated over time. If we have either one case, we can still use OLS or generalized least squares GLS to get a consistent estimate of the parameters. If we have both cases at the same time, then OLS is not consistent anymore, and we need to use a non-linear estimator. This case is essentially what ARMA does.

Linear Model with autocorrelated error (GLSAR)

This model assumes that the explanatory variables, regressors, are uncorrelated with the error term. But the error term is an autoregressive process, i.e.

$$E(x_t, \varepsilon_t) = 0$$

$$\varepsilon_t = a_1\varepsilon_{t-1} + a_2\varepsilon_{t-1} + ... + a_k\varepsilon_{t-k}$$

An example will be presented in the next section.

Linear Model with lagged dependent variables (OLS, AR, VAR)

This group of models assume that past dependent variables, y_{t-i}, are included among the regressors, but that the error term are not serially correlated

$$E(\varepsilon_t, \varepsilon_s) = 0, \text{ for } t \neq s$$

$$y_t = a_1 y_{t-1} + a_2 y_{t-1} + ... + a_k y_{t-k} + x_t\beta + \varepsilon_t$$

Dynamic processes like autoregressive processes depend on observations in the past. This means that we have to decide what to do with the initial observations in our sample where we do nnt observe any past values.

The simplest way is to treat the first observation as fixed, and analyse our sample starting with the k-th observation. This leads to conditional least squares or conditional maximum likelihood estimation. For conditional least squares we can just use OLS to estimate, adding past *endog* to the *exog*. The vector autoregressive model (VAR) has the same basic statistical structure except that

we consider now a vector of endogenous variables at each point in time, and can also be estimated with OLS conditional on the initial information. (The stochastic structure of VAR is richer, because we now also need to take into account that there can be contemporaneous correlation of the errors, i.e. correlation at the same time point but across equations, but still uncorrelated across time.) The second estimation method that is currently available in statsmodels is maximum likelihood estimation. Following the same approach, we can use the likelihood function that is conditional on the first observations. If the errors are normaly distributed, then this is essentially equivalent to least squares. However, we can easily extend conditional maximum likelihood to other models, for example GARCH, linear models with generalized autoregressive conditional heteroscedasticity, where the variance depends on the past, or models where the errors follow a non-normal distribution, for example Student-t distributed which has heavier tails and is sometimes more appropriate in finance.

The second way to treat the problem of intial conditions is to model them together with other observations, usually under the assumption that the process has started far in the past and that the initial observations are distributed according to the long run, i.e. stationary, distribution of the observations. This exact maximum likelihood estimator is implemented in statsmodels for the autoregressive process in statsmodels.tsa.AR, and for the ARMA process in statsmodels.tsa.ARMA.

Autoregressive Moving average model (ARMA)

ARMA combines an autoregressive process of the dependent variable with a error term, moving-average or MA, that includes the present and a linear combination of past error terms, an ARMA(p,q) is defined as

$$E(\varepsilon_t, \varepsilon_s) = 0, \text{ for } t \neq s$$

$$y_t = \mu + a_1 y_{t-1} + \ldots + a_k y_{t-p} + \varepsilon_t + b_1 \varepsilon_{t-1} + \ldots + b_q \varepsilon_{t-q}$$

As a simplified notation, this is often expressed in terms of lag-polynomials as

$$\phi(L) y_t = \psi(L) \varepsilon_t$$

where

$$\phi(L) = 1 - a_1 L^1 - a_2 L^2 - \ldots - a_k L^p$$

$$\psi(L) = 1 + b_1 L^1 + b_2 L^2 + \ldots + b_k L^q$$

L is the lag or shift operator, $L^i x_t = x_{t-i}, L^0 = 1$. This is the same process that scipy.lfilter uses. Forecasting with ARMA models has become popular since the 1970's as Box-Jenkins methodology, since it often showed better forecast performance than more complex, structural models.

Using OLS to estimate this process, i.e. regressing y_t on past y_{t-i}, does not provide a consistent estimator. The process can be consistently estimated using either conditional least squares, which in this case is a non-linear estimator, or conditional maximum likelihood or with exact maximum likelihood. The difference between conditional methods and exact MLE is the same as described before. statsmodels provides estimators for both methods in tsa.ARMA which will be described in more detail below.

Time series analysis is a vast field in econometrics with a large range of models that extend on the basic linear models with the assumption of normally distributed errors in many ways, and provides a range of statistical tests to identify an appropriate model specification or test the underlying assumptions.

Besides estimation of the main linear time series models, statsmodels also provides a range of descriptive statistics for time series data and associated statistical tests. We include an overview in the next section before describing AR, ARMA and VAR in more details. Additional results that facilitate the usage and interpretation of the estimated models, for example impulse response functions, are also available.

OLS, GLSAR and serial correlation

Suppose we want to model a simple linear model that links the stock of money in the economy to real GDP and consumer price index CPI, example in Greene (2003, ch. 12). We import numpy and statsmodels, load the variables from the example dataset included in statsmodels, transform the data and fit the model with OLS:

```
import numpy as np
import scikits.statsmodels.api as sm
tsa = sm.tsa  # as shorthand

mdata = sm.datasets.macrodata.load().data
endog = np.log(mdata['m1'])
exog = np.column_stack([np.log(mdata['realgdp']),
                        np.log(mdata['cpi'])])
exog = sm.add_constant(exog, prepend=True)

res1 = sm.OLS(endog, exog).fit()
```

`print res1.summary()` provides the basic overview of the regression results. We skip it here to safe on space. The Durbin-Watson statistic that is included in the summary is very low indicating that there is a strong autocorrelation in the residuals. Plotting the residuals shows a similar strong autocorrelation.

As a more formal test we can calculate the autocorrelation, the Ljung-Box Q-statistic for the test of zero autocorrelation and the associated p-values:

```
acf, ci, Q, pvalue = tsa.acf(res1.resid, nlags=4,
                             confint=95, qstat=True,
                             unbiased=True)
acf
#array([1., 0.982, 0.948, 0.904, 0.85])
pvalue
#array([3.811e-045, 2.892e-084,
        6.949e-120, 2.192e-151])
```

To see how many autoregressive coefficients might be relevant, we can also look at the partial autocorrelation coefficients

```
tsa.pacf(res1.resid, nlags=4)
#array([1., 0.982, -0.497, -0.062, -0.227])
```

Similar regression diagnostics, for example for heteroscedasticity, are available in *statsmodels.stats.diagnostic*. Details on these functions and their options can be found in the documentation and docstrings.

The strong autocorrelation indicates that either our model is misspecified or there is strong autocorrelation in the errors. If we assume that the second is correct, then we can estimate the model with GLSAR. As an example, let us assume we consider four lags in the autoregressive error.

```
mod2 = sm.GLSAR(endog, exog, rho=4)
res2 = mod2.iterative_fit()
```

iterative_fit alternates between estimating the autoregressive process of the error term using tsa.yule_walker, and feasible sm.GLS. Looking at the estimation results shows two things, the parameter estimates are very different between OLS and GLS, and the autocorrelation in the residual is close to a random walk:

```
res1.params
#array([-1.502,  0.43 ,  0.886])
res2.params
#array([-0.015,  0.01 ,  0.034])

mod2.rho
#array([ 1.009, -0.003,  0.015, -0.028])
```

This indicates that the short run and long run dynamics might be very different and that we should consider a richer dynamic model, and that the variables might not be stationary and that there might be unit roots.

Stationarity, Unit Roots and Cointegration

Loosely speaking, stationarity means here that the mean, variance and intertemporal correlation structure remains constant over time. Non-stationarities can either come from deterministic changes like trend or seasonal fluctuations, or the stochastic properties of the process, if for example the autoregressive process has a unit root, that is one of the roots of the lag polynomial is on the unit circle. In the first case, we can remove the deterministic component by detrending or deseasonalization. In the second case we can take first differences of the process,

Differencing is a common approach in the Box-Jenkins methodology and gives rise to ARIMA, where the I stands for integrated processes, which are made stationary by differencing. This lead to a large literature in econometrics on unit-root testing that tries to distinguish deterministic trends from unit roots or stochastic trends. statsmodels provides the *augmented Dickey-Fuller test*. Monte Carlo studies have shown that it is often the most powerful of all unit roots test.

To illustrate the results, we just show two results. Testing the log of the stock of money with a null hypothesis of unit roots against an alternative of stationarity around a linear trend, shows an adf-statistic of -1.5 and a p-value of 0.8, so we are far away from rejecting the unit root hypothesis:

```
tsa.adfuller(endog, regression="ct")[:2]
(-1.561, 0.807)
```

If we test the differenced series, that is the growth rate of moneystock, with a Null hypothesis of Random Walk with drift, then we can strongly reject the hypothesis that the growth rate has a unit root (p-value 0.0002)

```
tsa.adfuller(np.diff(endog), regression="c")[:2]
(-4.451, 0.00024)
```

ARMA processes and data

The identification for ARIMA(p,d,q) processes, especially choosing the number of lagged terms, p and q, to include, remains partially an art. One recommendation in the Box-Jenkins methodology is to look at the pattern in the autocorrelation (acf) and partial autocorrelation (pacf) functions

scikits.statsmodels.tsa.arima_process contains a class that provides several properties of ARMA processes and a random process generator. As an example, *statsmodels/examples/tsa/arma_plots.py* can be used to plot autocorrelation and partial autocorrelation functions for different ARMA models.

This allows easy comparison of the theoretical properties of an ARMA process with their empirical counterparts. For example, define the lag coefficients for an ARMA(2,2) process, generate a random process and compare observed and theoretical pacf:

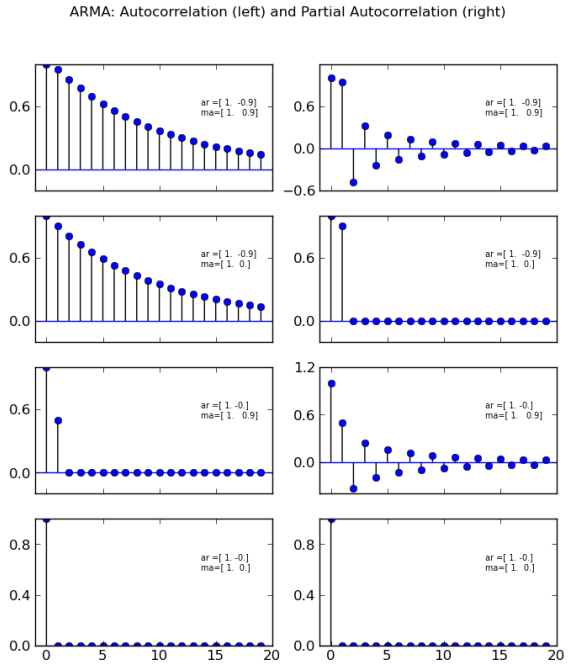

Fig. 1: *ACF and PACF for ARMA(p,q) This illustrated that the pacf is zero after p terms for AR(p) processes and the acf is zero after q terms for MA(q) processes.*

```
import scikits.statsmodels.tsa.arima_process as tsp
ar = np.r_[1., -0.5, -0.2]; ma = np.r_[1.,  0.2, -0.2]
np.random.seed(123)
x = tsp.arma_generate_sample(ar, ma, 20000, burnin=1000)
sm.tsa.pacf(x, 5)
array([1.,  0.675, -0.053,  0.138, -0.018,  0.038])

ap = tsp.ArmaProcess(ar, ma)
ap.pacf(5)
array([1.,  0.666, -0.035,  0.137, -0.034,  0.034])
```

We can see that they are very close in a large generated sample like this. *ArmaProcess* defines several additional methods that calculate properties of ARMA processes and to work with lag-polynomials: *acf*, *acovf*, *ar*, *ar_roots*, *arcoefs*, *arma2ar*, *arma2ma*, *arpoly*, *from_coeffs*, *from_estimation*, *generate_sample*, *impulse_response*, *invertroots*, *isinvertible*, *isstationary*, *ma*, *ma_roots*, *macoefs*, *mapoly*, *nobs*, *pacf*, *periodogram*. The sandbox has a FFT version of some of this to look at the frequency domain properties.

ARMA Modeling

Statsmodels provides several helpful routines and models for working Autoregressive Moving Average (ARMA) time-series models, including simulation and estimation code. For example, after importing *arima_process* as *ap* from *scikits.statsmodels.tsa* we can simulate a series[1]

```
>>> ar_coef = [1, .75, -.25]
>>> ma_coef = [1, -.5]
>>> nobs = 100
>>> y = ap.arma_generate_sample(ar_coef,
...                             ma_coef, nobs)
>>> y += 4 # add in constant
```

We can then estimate an ARMA model of the series

```
>>> mod = tsa.ARMA(y)
>>> res = arma_mod.fit(order=(2,1), trend='c',
...                    method='css-mle', disp=-1)
>>> arma_res.params
array([ 4.0092, -0.7747,  0.2062, -0.5563])
```

The estimation method, 'css-mle', indicates that the starting parameters from the optimization are to be obtained from the conditional sum of squares estimator and then the exact likelihood is optimized. The exact likelihood is implemented using the Kalman Filter.

Filtering

We have recently implemented several filters that are commonly used in economics and finance applications. The three most popular method are the Hodrick-Prescott, the Baxter-King filter, and the Christiano-Fitzgerald. These can all be viewed as approximations of the ideal band-pass filter; however, discussion of the ideal band-pass filter is beyond the scope of this paper. We will [briefly review the implementation details of each] give an overview of each of the methods and then present some usage examples.

The Hodrick-Prescott filter was proposed by Hodrick and Prescott [HPres], though the method itself has been in use across the sciences since at least 1876 [Stigler]. The idea is to separate a time-series y_t into a trend τ_t and cyclical compenent ζ_t

$$y_t = \tau_t + \zeta_t$$

The components are determined by minimizing the following quadratic loss function

$$\min_{\{\tau_t\}} \sum_t^T \zeta_t^2 + \lambda \sum_{t=1}^T \left[(\tau_t - \tau_{t-1}) - (\tau_{t-1} - \tau_{t-2})\right]^2$$

where $\tau_t = y_t - \zeta_t$ and λ is the weight placed on the penalty for roughness. Hodrick and Prescott suggest using $\lambda = 1600$ for quarterly data. Ravn and Uhlig [RUhlig] suggest $\lambda = 6.25$ and $\lambda = 129600$ for annual and monthly data, respectively. While there are numerous methods for solving the loss function, our implementation uses *scipy.sparse.linalg.spsolve* to find the solution to the generalized ridge-regression suggested in Danthine and Girardine [DGirard].

Baxter and King [BKing] propose an approximate band-pass filter that deals explicitly with the periodicity of the business cycle. By applying their band-pass filter to a time-series y_t, they produce a series y_t^* that does not contain fluctuations at frequencies higher or lower than those of the business cycle. Specifically, in the time domain the Baxter-King filter takes the form of a symmetric moving average

$$y_t^* = \sum_{k=-K}^{K} a_k y_{t-k}$$

where $a_k = a_{-k}$ for symmetry and $\sum_{k=-K}^{K} a_k = 0$ such that the filter has trend elimination properties. That is, series that contain quadratic deterministic trends or stochastic processes that are integrated of order 1 or 2 are rendered stationary by application of the filter. The filter weights a_k are given as follows

$$a_j = B_j + \theta \text{ for } j = 0, \pm 1, \pm 2, \ldots, \pm K$$

$$B_0 = \frac{(\omega_2 - \omega_1)}{\pi}$$

$$B_j = \frac{1}{\pi j}\left(\sin(\omega_2 j) - \sin(\omega_1 j)\right) \text{ for } j = 0, \pm 1, \pm 2, \ldots, \pm K$$

where θ is a normalizing constant such that the weights sum to zero

$$\theta = \frac{-\sum_{j=-K}^{K} b_j}{2K+1}$$

and

$$\omega_1 = \frac{2\pi}{P_H}, \omega_2 = \frac{2\pi}{P_L}$$

with the periodicity of the low and high cut-off frequencies given by P_L and P_H, respectively. Following Burns and Mitchell's [] pioneering work which suggests that US business cycles last from 1.5 to 8 years, Baxter and King suggest using $P_L = 6$ and $P_H = 32$ for quarterly data or 1.5 and 8 for annual data. The authors suggest setting the lead-lag length of the filter K to 12 for quarterly data. The transformed series will be truncated on either end by K. Naturally the choice of these parameters depends on the available sample and the frequency band of interest.

The last filter that we currently provide is that of Christiano and Fitzgerald [CFitz]. The Christiano-Fitzgerald filter is again a weighted moving average. However, their filter is asymmetric about t and operates under the (generally false) assumption that y_t follows a random walk. This assumption allows their filter to approximate the ideal filter even if the exact time-series model of y_t is not known. The implementation of their filter involves the calculations of the weights in

$$y_t^* = B_0 y_t + B_1 y_{t+1} + \cdots + B_{T-1-t} y_{T-1} + \tilde{B}_{T-t} y_T +$$

$$B_1 y_{t-1} + \cdots + B_{t-2} y_2 + \tilde{B}_{t-1} y_1$$

for $t = 3, 4, \ldots, T-2$, where

$$B_j = \frac{\sin(jb) - \sin(ja)}{\pi j}, j \geq 1$$

$$B_0 = \frac{b-a}{\pi}, a = \frac{2\pi}{P_u}, b = \frac{2\pi}{P_L}$$

$\tilde{B}_{T-t}$ and $\tilde{B}_{t-1}$ are linear functions of the B_j's, and the values for $t = 1, 2, T-1$, and T are also calculated in much the same way. See the authors' paper or our code for the details. P_U and P_L are as described above with the same interpretation.

Moving on to some examples, the below demonstrates the API and resultant filtered series for each method. We use series for unemployment and inflation to demonstrate 2. They are traditionally thought to have a negative relationship at business cycle frequencies.

```
>>> from scipy.signal import lfilter
>>> data = sm.datasets.macrodata.load()
>>> infl = data.data.infl[1:]
>>> # get 4 qtr moving average
>>> infl = lfilter(np.ones(4)/4, 1, infl)[4:]
>>> unemp = data.data.unemp[1:]
```

To apply the Hodrick-Prescott filter to the data 3, we can do

```
>>> infl_c, infl_t = tsa.filters.hpfilter(infl)
>>> unemp_c, unemp_t = tsa.filters.hpfilter(unemp)
```

The Baxter-King filter 4 is applied as

```
>>> infl_c = tsa.filters.bkfilter(infl)
>>> unemp_c = tsa.filters.bkfilter(unemp)
```

The Christiano-Fitzgerald filter is similarly applied 5

```
>>> infl_c, infl_t = tsa.filters.cfilter(infl)
>>> unemp_c, unemp_t = tsa.filters.cfilter(unemp)
```

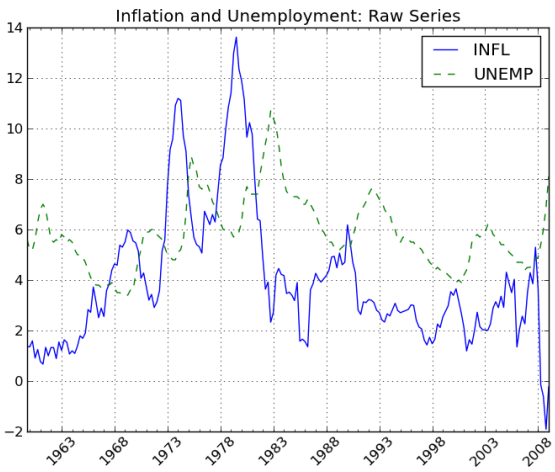

Fig. 2: *Unfiltered Inflation and Unemployment Rates 1959Q4-2009Q1*

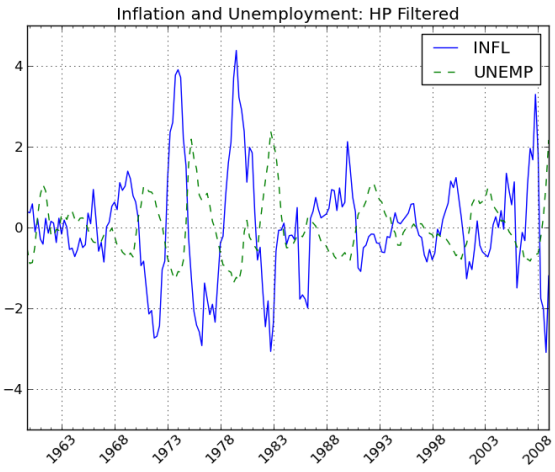

Fig. 3: *Unfiltered Inflation and Unemployment Rates 1959Q4-2009Q1*

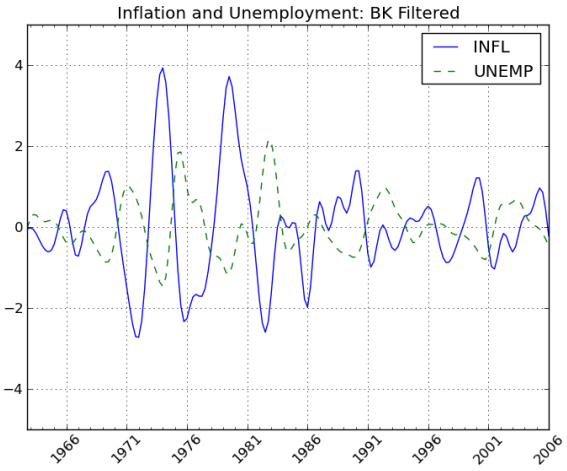

Fig. 4: *Unfiltered Inflation and Unemployment Rates 1959Q4-2009Q1*

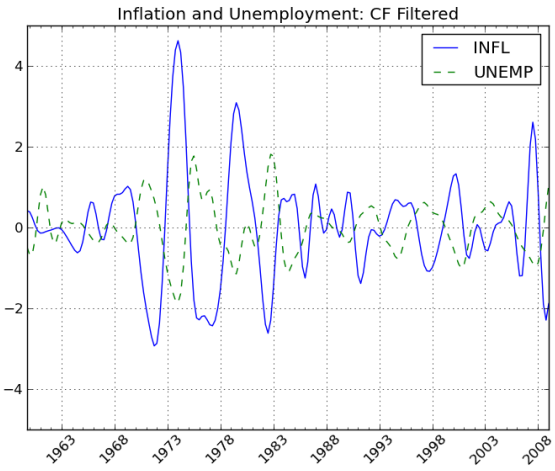

Fig. 5: *Unfiltered Inflation and Unemployment Rates 1959Q4-2009Q1*

Statistical Benchmarking

We also provide for another frequent need of those who work with time-series data of varying observational frequency--that of benchmarking. Benchmarking is a kind of interpolation that involves creating a high-frequency dataset from a low-frequency one in a consistent way. The need for benchmarking arises when one has a low-frequency series that is perhaps annual and is thought to be reliable, and the researcher also has a higher frequency series that is perhaps quarterly or monthly. A benchmarked series is a high-frequency series consistent with the benchmark of the low-frequency series.

We have implemented Denton's modified method. Originally proposed by Denton [Denton] and improved by Cholette [Cholette]. To take the example of turning an annual series into a quarterly one, Denton's method entails finding a benchmarked series X_t that solves

$$\min_{\{X_t\}} \sum_{t}^{T} \left(\frac{X_t}{I_t} - \frac{X_{t-1}}{I_{t-1}} \right)^2$$

subject to

$$\sum_{t=2}^{T} X_t = A_y \; y = \{1, \ldots, \beta\}$$

That is, the sum of the benchmarked series must equal the annual benchmark in each year. In the above A_y is the annual benchmark for year y, I_t is the high-frequency indicator series, and β is the last year for which the annual benchmark is available. If $T > 4\beta$, then extrapolation is performed at the end of the series. To take an example, given the US monthly industrial production index and quarterly GDP data, from 2009 and 2010, we can construct a benchmarked monthly GDP series

```
>>> iprod_m = np.array([ 87.4510, 86.9878, 85.5359,
                         84.7761, 83.8658, 83.5261, 84.4347,
                         85.2174, 85.7983, 86.0163, 86.2137,
                         86.7197, 87.7492, 87.9129, 88.3915,
                         88.7051, 89.9025, 89.9970, 90.7919,
                         90.9898, 91.2427, 91.1385, 91.4039,
                         92.5646])
>>> gdp_q = np.array([14049.7, 14034.5, 14114.7,
                      14277.3, 14446.4, 14578.7, 14745.1,
                      14871.4])
```

```
>>> gdp_m = tsa.interp.dentonm(iprod_m, gdp_q,
                               freq="qm")
```

Modeling multiple time series: Vector autoregressive (VAR) models

It is common in finance, economics, and other fields to model relationships among multiple time series. For example, an economist may wish to understand the impact of monetary policy on inflation and unemployment. A widely used tool for analyzing multiple time series is the vector autoregressive (VAR) model. At each time point we observe a K-vector Y_t of data points, one for each time series. These can be modeled similar to an AR process as above

$$Y_t = A_1 Y_{t-1} + \ldots + A_p Y_{t-p} + \varepsilon_t.$$

In this case, the coefficients A_i are square matrices. As with prior models, the error ε_t is typically assumed to be normally distributed and uncorrelated over time. This model can be estimated by MLE or equivalently by OLS, either case as a single regression or by noticing that the model decouples into K separate linear regressions, one for each time series.

We have recently written a comprehensive implementation of VAR models on stationary data following [Lütkepohl]. In addition to estimation, we are also interested in

- Analysis of impulse responses (the effect of a unit shock to one variable on all of the others)
- Statistical tests: whiteness and normality of residuals, Granger-causality
- Lag order selection: how many lags Y_{t-i} to include
- Multi-step forecasting and forecast error variance decomposition

We will illustrate some of the VAR model features on the `macrodata` data set in statsmodels. Here is a code snippet to load the data and fit the model with two lags of the log-differenced data:

```
mdata = sm.datasets.macrodata.load().data
mdata = mdata[['realgdp','realcons','realinv']]
names = mdata.dtype.names
data = mdata.view((float,3))
data = np.diff(np.log(data), axis=0)

model = VAR(data, names=names)
res = model.fit(2)
```

As with most other models in statsmodels, `res.summary()` provides a console output summary of the estimated coefficients and other standard model diagnostics. The data itself can be visualized by a number of plotting functions:

```
>>> res.plot_sample_acorr()
```

Impulse responses can be generated and plotted thusly:

```
>>> irf = res.irf(10) # 10 periods
>>> irf.plot()
```

n-step ahead forecasts can similarly be generated and plotted:

```
>>> res.plot_forecast(5)
```

The forecast error variance decomposition can also be computed and plotted like so

```
>>> res.fevd().plot()
```

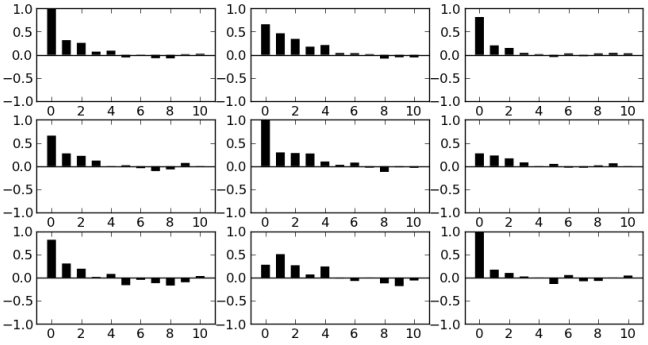

Fig. 6: VAR sample autocorrelation

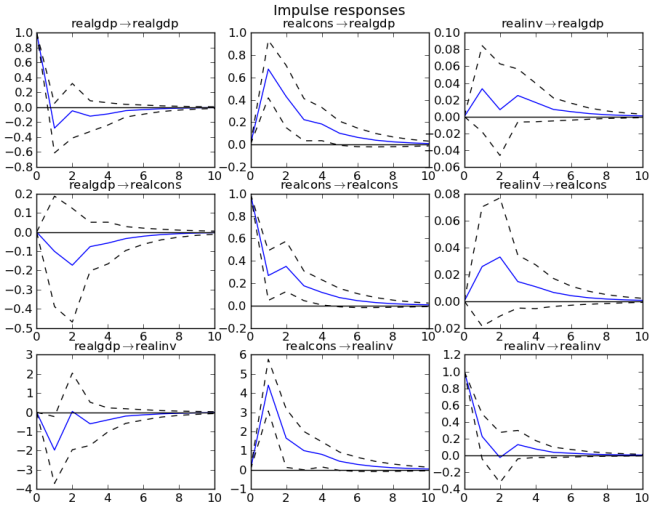

Fig. 7: VAR impulse response functions

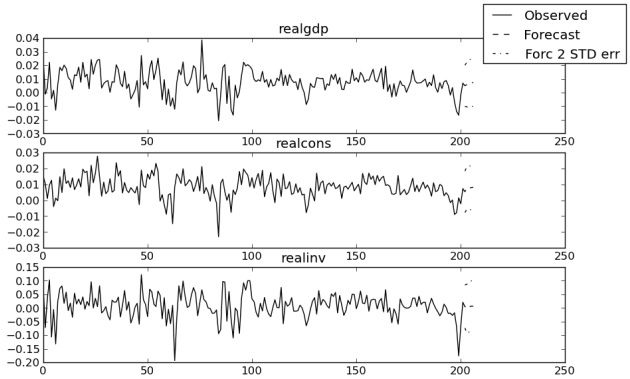

Fig. 8: VAR 5 step ahead forecast

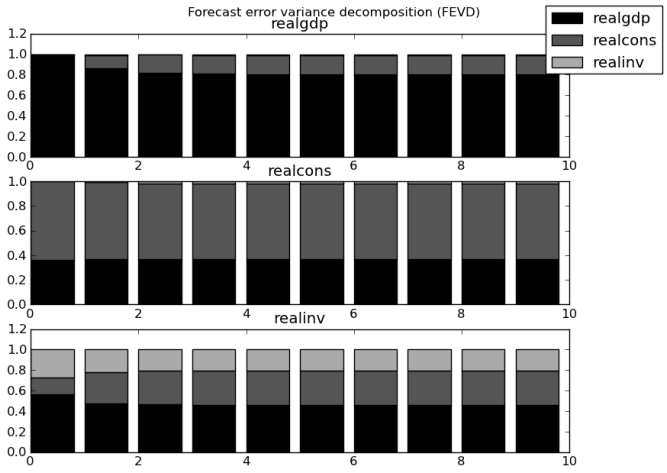

Fig. 9: *VAR Forecast error variance decomposition*

Various tests such as testing Granger causality can be carried out using the results object:

```
>>> res.test_causality('realinv', 'realcons')
H_0: ['realcons'] do not Granger-cause realinv
Conclusion: reject H_0 at 5.00% significance level
{'conclusion': 'reject',
 'crit_value': 3.0112857238108273,
 'df': (2, 579),
 'pvalue': 3.7842822166888971e-10,
 'signif': 0.05,
 'statistic': 22.528593566083575}
```

Obviously we are just providing a flavor of some of the features available for VAR models. The statsmodels documentation has a more comprehensive treatment of the feature set. We plan to continue implementing other related models for multiple time series, such as the vector error correction models (VECM) for analyzing cointegrated (non-stationary) time series data. Other more sophisticated models in the recent time series literature could also be implemented within the same framework.

Conclusions

statsmodels development over the last few years has been focused on building correct and tested implementations of the standard suite of econometric models available in other statistical computing environments, such as R. However, there is still a long road ahead before Python will be on the same level library-wise with other computing environments focused on statistics and econometrics. We believe that, given the wealth of powerful scientific computing and interactive research tools coupled with the excellent Python language, statsmodels can make Python become a premier environment for doing applied statistics and econometrics work. Future work will need to integrate all of these tools to create a smooth and intuitive user experience comparable to industry standard commercial and open source statistical products.

We have built a foundational set of tools for several ubiquitous classes of time series models which we hope will go a long way toward meeting the needs of applied statisticians and econometricians programming in Python.

1. Notice that the AR coefficients and MA coefficients, both include a 1 for the zero lag. Further, the signs on the AR coefficients are reversed versus those estimated by *tsa.ARMA* due to the differing conventions of *scipy.signal.lfilter*.

REFERENCES

[BKing] Baxter, M. and King, R.G. 1999. "Measuring Business Cycles: Approximate Band-pass Filters for Economic Time Series." *Review of Economics and Statistics*, 81.4, 575-93.

[Cholette] Cholette, P.A. 1984. "Adjusting Sub-annual Series to Yearly Benchmarks." *Survey Methodology*, 10.1, 35-49.

[CFitz] Christiano, L.J. and Fitzgerald, T.J. 2003. "The Band Pass Filter." *International Economic Review*, 44.2, 435-65.

[DGirard] Danthine, J.P. and Girardin, M. 1989. "Business Cycles in Switzerland: A Comparative Study." *European Economic Review* 33.1, 31-50.

[Denton] Denton, F.T. 1971. "Adjustment of Monthly or Quarterly Series to Annual Totals: An Approach Based on Quadratic Minimization." *Journal of the American Statistical Association*, 66.333, 99-102.

[HPres] Hodrick, R.J. and Prescott, E.C. 1997. "Postwar US Business Cycles: An Empirical Investigation." *Journal of Money, Credit, and Banking,* 29.1, 1-16.

[RUhlig] Ravn, M.O and Uhlig, H. 2002. "On Adjusting the Hodrick-Prescott Filter for the Frequency of Observations." *Review of Economics and Statistics,* 84.2, 371-6.

[Stigler] Stigler, S.M. 1978. "Mathematical Statistics in the Early States." *Annals of Statistics* 6, 239-65,

[Lütkepohl] Lütkepohl, H. 2005. "A New Introduction to Multiple Time Series Analysis"

Improving efficiency and repeatability of lake volume estimates using Python

Tyler McEwen‡*, Dharhas Pothina‡, Solomon Negusse‡

Abstract—With increasing population and water use demands in Texas, accurate estimates of lake volumes is a critical part of planning for future water supply needs. Lakes are large and surveying them is expensive in terms of labor, time and cost. High spatial resolution surveys are prohibitive to conduct, hence lake are usually surveyed along widely spaced survey lines. While this choice reduces the time spent in field data collection, it increases the time required for post processing significantly. Standard spatial interpolation techniques available in commercial software are not well suited to this problem and a custom procedure was developed using in-house Fortran software. This procedure involved difficult to repeat manual manipulation of data in graphical user interfaces, visual interpretation of data and a laborious manually guided interpolation process. Repeatibility is important since volume differences derived from multiple surveys of individual reservoirs provides estimates of capacity loss over time due to sedimentation. Through python scripts that make use of spatial algorithms and GIS routines available within various Python scientific modules, we first streamlined our original procedure and then replaced it completely with a new pure python implementation. In this paper, we compare the original procedure, the streamlined procedure and our new pure python implementation with regard to automation, efficiency and repeatability of our lake volumetric estimates. Applying these techniques to Lake Texana in Texas, we show that the new pure python implementation reduces data post processing time from approximately 90 man hours to 8 man hours while improving repeatability and maintaining accuracy.

Index Terms—gis, spatial interpolation, hydrographic surveying, bathymetry, lake volume, reservoir volume, anisotropic, inverse distance wieghted, sedimentation

Introduction

With increasing population and water use demands in Texas, accurate estimates of lake volumes is a critical part of planning for future water supply needs. In order to correctly manage surface water supplies for the State of Texas, it is vital that managers and state water planners have accurate estimates of reservoir volumes and capacity loss rates due to sedimentation. To address these issues, in 1991 the Texas Legislature authorized the Texas Water Development Board (TWDB) to develop a cost-recovery hydrographic surveying program. The program is charged with determining reservoir storage capacities, sedimentation levels, sedimentation rates, and available water supply projections to benefit Texas. Since its inception, staff in the hydrographic survey program have completed more than 125 lake surveys. Included in each survey report are updated elevation-area-capacity tables and bathymetric contour maps.

Lakes are large and surveying them is expensive in terms of labor, time and cost. Over the years, the Texas Water Development Board (TWDB) has settled on a 500 ft spacing of survey lines oriented perpendicular to an assumed relic stream channel for hydrographic data collection as a good balance between survey effort and level of data coverage. While this choice reduces the time spent in data collection, it significantly increases the time needed for post-survey processing. Currently, a typical major reservoir (greater than 5,000 acre-feet) survey can consume anywhere between 1 to 7 weeks of time in field data collection and 2 to 8 weeks of time in data post-survey processing before a volumetric estimate is available.

Volumetric estimate algorithms available in commercial software are usually based on the Delaunay triangulation method for actual survey points bounded by digitized lake boundary at a known elevation. When applied to data collected with widely spaced survey lines, these techniques tend to underestimate the true volume of the lake. To overcome this issue, TWDB preconditions the survey point dataset by inserting additional artificial points in between survey lines and using directional linear interpolation to estimate the bathymetry at the inserted points. Delaunay triangulation of the resulting dataset gives a more accurate estimate of lake volume. This technique makes use of the assumption that the profile of the lake between each set of survey lines is similar to that of the survey lines. Figure REF shows the improvement in the representation of the bathymetry of the lake that can be obtained by such preconditioning. Previous surveys have shown that the improved bathymetric representation of the lake increase volume estimates [Furn08].

While effective in improving volume estimates, this technique as currently implemented has a number of flaws. Notably, it depends on exact positions of survey points and hence is difficult to apply repeatibly for repeat surveys of lakes. In addition, it requires manual visual interpretation and manipulation of data in graphical user interfaces as well as a laborious guided interpolation process.

Standard TWDB Surveying Technique

TWDB hydrographic surveys are conducted using a boat mounted single beam multi-frequency (200, 50 and 24 kHz) sub-bottom profiling sonar echo sounder integrated with differential global positioning system (DGPS) equipment along preplanned survey lines. Survey planning, operationally defined here as the spacing and orientation of pre-planned survey lines, is likely to

* Corresponding author: tyler.mcewen@twdb.state.tx.us
‡ Texas Water Development Board

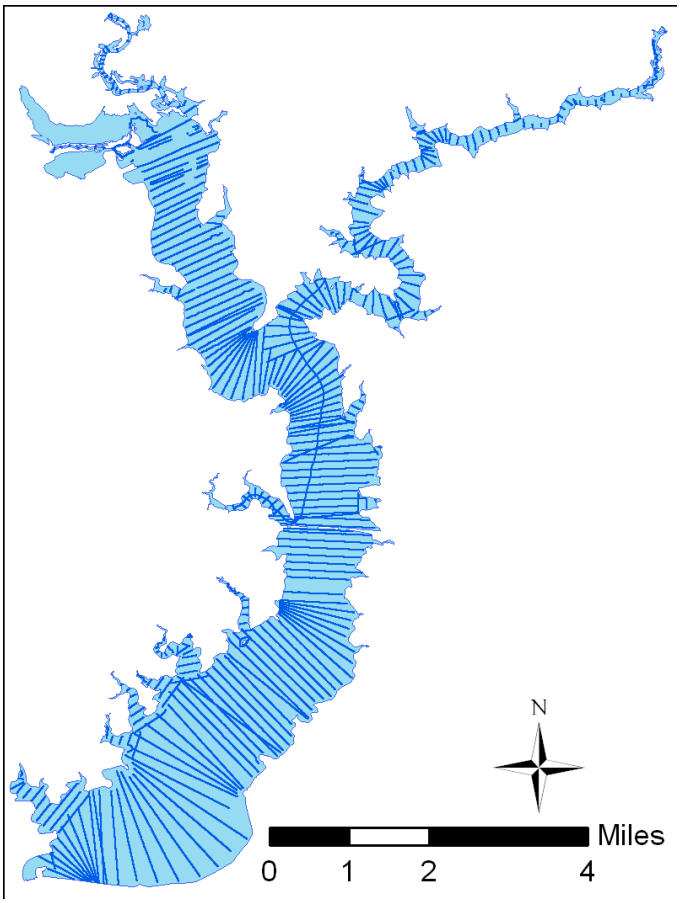

Fig. 1: Survey lines used for the 2010 hydrographic survey of Lake Texana

affect volumetric calculations if there are notable bathymetric changes between surveyed lines. In many cases, however, reservoir bathymetry will not be known before the survey, and survey lines must be planned based on an interpretation of the reservoir shape in map-view and the presumed location and orientation of the submerged stream channel. Previous TWDB surveys have been conducted using lines spaced at from 100 ft to 1000 ft intervals [TWDB06], [TWDB09], [TWDB09b]. Analyses of these surveys showed that greater volumes are obtained from surveys conducted with higher density line spacin. However, with suitable post processing the lower 500 ft resolution survey density is sufficient to accurately estimate the volume of the lake [Furn06], [Furn10].

Figure 1 exibits TWDB standard bathymetric survey data collection along survey lines spaced 500 feet apart and oriented perpendicular to the assumed location of the submerged river channel (usually taken to be along the centerline of the lake). Radial lines are utilized when the shape of the lake and presumed shape of the submerged river channel curve. Data post processing is then used to improve the representation of the bathymetry between survey lines.

Data processing with HydroEdit

Over the years, the TWDB has developed several post processing routines that have been packaged together in an in-house Fortran program, HydroEdit. HydroEdit contains modules to integrate boat GPS and sonar bathymetric data, calculate sediment thicknesses, extrapolate into regions with no survey data, convert data between

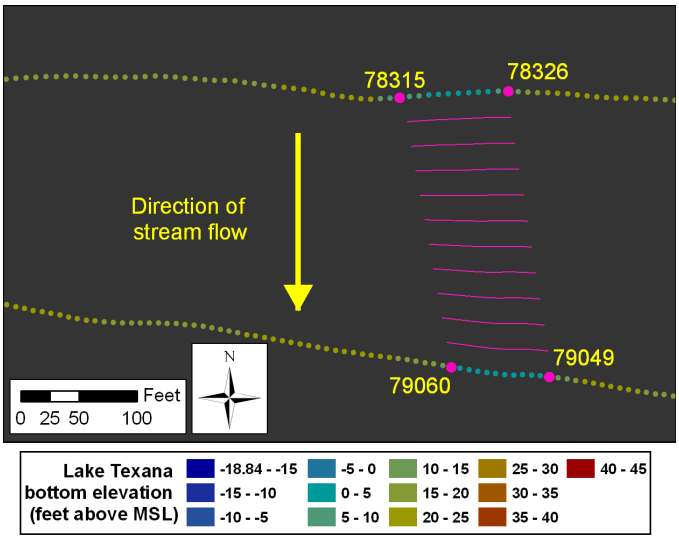

Fig. 2: Example of a single HydroEdit guided interpolation

projected and geographic coordinate systems, merge data files and generate the preconditioned dataset for volumetric estimates [Furn06], [Furn08].

One of the primary functions of the Hydroedit is to perform is to insert extra artificial survey points and interpolate bathymetric data to those points. Using ArcGIS software, areas of desired interpolation from one survey line segment to an adjacent survey line segment are visually located and their point identification numbers are manually recorded into a text file along with parameters that control the number of artificial survey lines to be inserted between the adjacent survey lines and the density of points to be inserted on each artificial survey line. HydroEdit then linearly interpolates the bathymetry from the adjacent survey line segments to the points on the artificial segments. In addition, HydroEdit allows for more complicated interpolations for locations where there is evidence that where a river may curve or double back between survey lines. These require more complicated procedures that include the creation and export of a polygon feature in ArcGIS, as well as text entries in the HydroEdit input file. Figure 2 shows an example of the visual inspection required for a single HydroEdit interpolation between adjacent survey line segments. The portion of the input text file corresponding to this interpolation is as follows:

```
Section1
53 54 0
Section2
53 79049 79060 3 0
54 78326 78315 3 0
```

This procedure has to be followed for every pair of adjacent survey lines in the dataset. In some cases, survey lines must be broken into multiple segments in order to capture a relic river channel than may require interpolation in a direction different from the rest of the transect. This is laborious work and is the cause of the majority of the time consumed in the data post-survey processing. The dependance of the technique on 4 individual survey points on adjacent survey line segments makes the interpolation survey specific requiring that new input files be created if a lake is resurveyed. This is both time consuming and prone to parts of the lake bathymetry being interpolated differently in repeat surveys. In addition, the technique starts to break down when survey lines intersect or are at sharp angles to each other. In addition, the

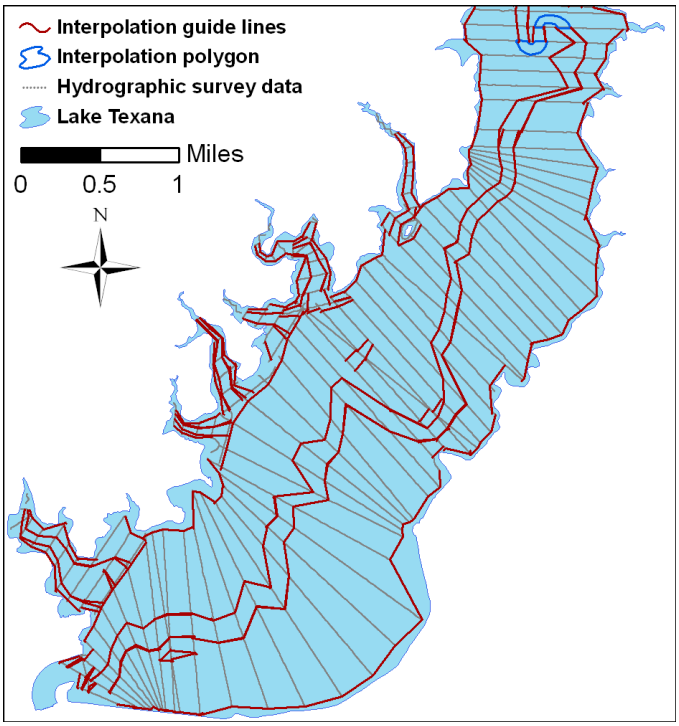

Fig. 3: Line-automated polygons and polylines for the lower portion of Lake Texana

density of the inserted artificial interpolated survey points is not consistent across the lake with some areas of high density and other areas of no interpolations. This is demonstrated in Figure 6.

Line-automated HydroEdit Using Python

Seeking to improve upon the lengthy and tedious process required to manually create a HydroEdit input text file, Python was utilized to automatically generate the HydroEdit input text file after manually drawing paired interpolation guide-lines in ArcGIS. This technique was named line-automated HydroEdit and was an interim step used to improve efficiency without having to abandon the HydroEdit codebase.

The line-automated HydroEdit algorithm is implemented through these simplified steps. Initially, the paired interpolation guide lines are drawn as polyline features and associated attribute fields are populated in ArcGIS. The attribute fields control interpolation options required in the HydroEdit input file. Next, the density of vertices for the interpolation guide lines is increased to ensure identification of the intersections with survey points. The intersection of the paired interpolation guide lines and survey lines are found efficiently using the KDTree algorithm available in scipy.spatial. Once the intersection points are identified the polyline attributes are used along with survey line metadata to autogenerate the corresponding entries in the HydroEdit input file. Figure 3 shows examples of paired guide lines used for the Line-automated HydroEdit interpolation of Lake Texana.

Anisotropic Elliptical Inverse Distance Weighting (AEIDW)

Merwade discusses at length how river channel bed morphology is anisotropic in that the bathymetric variability is greater transverse to the flow direction than along the flow direction. In addition, the direction of this anisotropy is not consistent; it varies with the orientation of the channel as exhibited by any sinous channel. He proposes and elliptical inverse distance weighting algorithm that follows this anisotropy as a simpler and computationally more efficient technique than anisotropic kriging [Merw06]. AEIDW involves first transforming the survey point dataset from cartesian coordinates to a flow oriented *s*-*n* coordinate system; Where *n* is the perpendicular distance of a point from a defined channel centerline and *s* is the distance along the centerline. Looking downstream points to the left of the centerline are assigned a positive *n* and points to the right a negative *n*, *s* is always positive. Since the flow direction is now always along the *s* coordinate, this transformation has the effect of removing the variation in direction of anisotropy.

Inverse Distance Weighting(IDW) is a form of interpolation the value at a point is approximated by a weighted average of observed values within an circular search neighborhood, whose radius is defined by the range of a fixed number of closest points. A common weighting function is the inverse of the distance squared. Elliptical Inverse Distance Weighting(EIDW) modifies the search radius to an ellipse by modifying the distance used in IDW by an elliptical measure of distance. By orienting the mmajor axis of this ellipse along the *s* axis where the topographic variability is lower, a point along the direction of flow will have greater predictive control at the point of interest than one transverse to flow at the same distance.

To increase the computational efficiency of the algorithm, rather than calculate the elliptical measure of distance, we multiply the *n* coordinates of the transformed dataset by the inverse of the ellipse's eccentricity. This trick along with the use of a KDTree to find the points within the search radius make the python implementation of AIEDW significantly faster than regular IDW interpolation algorithms in commerical packages.

Applying AEIDW to a Lake

The AEIDW python implementation was originally designed to generate bathymetric representations of river channels. For lakes, the technique is by segmenting the lake and applying AEIDW to each segment. For each lake segment a centerline polyline and a bounding polygon is drawn in ArcGIS. In practice, a segment is drawn for the original river channel, the main stem of the lake and each of the secondary stems. A high resolution grid of artificial survey points is generated that covers the entire lake. A python script cycles through the segments and applies AEIDW based on the segment centerline and interpolates data survey lines to artificial survey points that lie within the segment bounding polygon. Figure 4 shows the polygons and associated polylines for the lower portion of Lake Texana.

Lake Texana

The Palmetto Bend Dam was completed in 1979, impounding the Navidad River and creating Lake Texana [TWDB74]. At approximately 9,727 acres, Lake Texana is a small to medium major reservoir in Texas; the minimum acreage of major reservoirs in Texas is 5,000 acres.

TWDB collected bathymetric data for Lake Texana between January 12 and March 4, 2010. The daily average water surface elevations during that time ranged between 43.89 and 44.06 feet above mean sea level (NGVD29). During the survey, team mbers collected nearly 244,000 individual data points over cross-sections

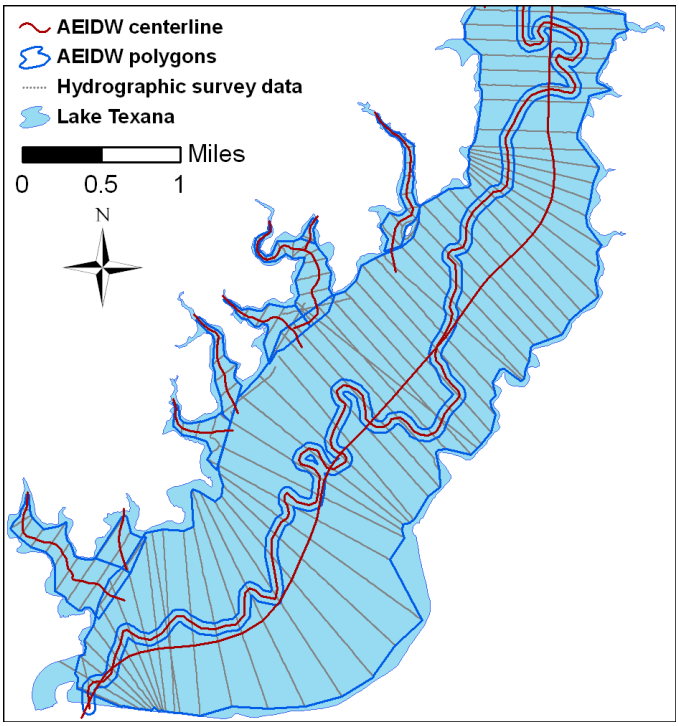

Fig. 4: AEIDW segment polygons and centerline polylines for the lower portion of Lake Texana

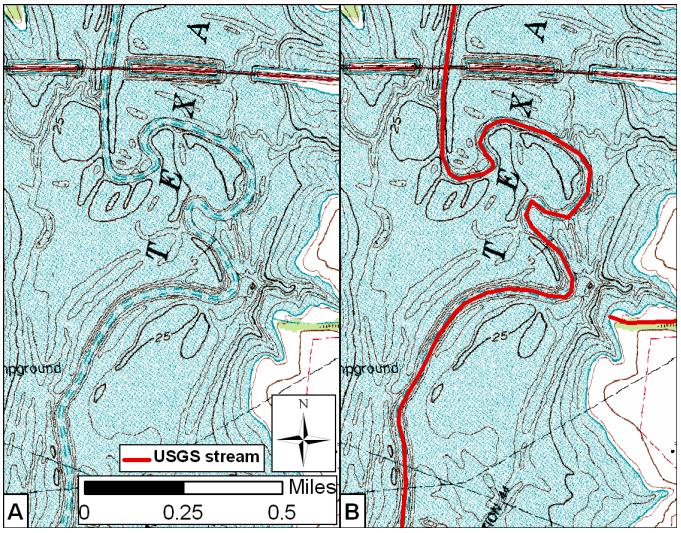

Fig. 5: USGS topographic map with delineated stream channel (A) and TWDB delineation of USGS stream channel of Lake Texana

totaling approximately 160 miles in length. Figure 2 shows where data collection occurred during the survey.

Figure 5 (A) below shows the USGS 24,000 scale topographic map and USGS delineated stream channel. Figure 5 (B) shows the TWDB delineation of USGS stream channel. These maps where used to guide the drawing of the Line-automated HydroEdit guide lines and the AIDW river centerline. This type of information is alwailable for some lakes in Texas, but not all.

Results

As a baseline for comparison, using HydroEdit, Lake Texana had approximately 3050 manually entered interpolations requiring approximately 90 man hours to complete. The overall increase in the estimated volume due to this post processing was 3.11%. In comparing methods,first we look at density and distribution of artificial survey points in the three methods. As can be seen from Figure 6, both the HydroEdit and the Line-automated HydroEdit methods have inconsistant point density. However much care is taken, the dependence of the basic HydroEdit technique on pairs of points on adjacent survey lines inevitably causes large variations in the artificial survey line density. These means that certain regions of the lake may not be interpolated well. The AEIDW technique on the otherhand allows for uniform point density throughout the each lake segment and allows for increased density in highly variable areas like near the stream channel.

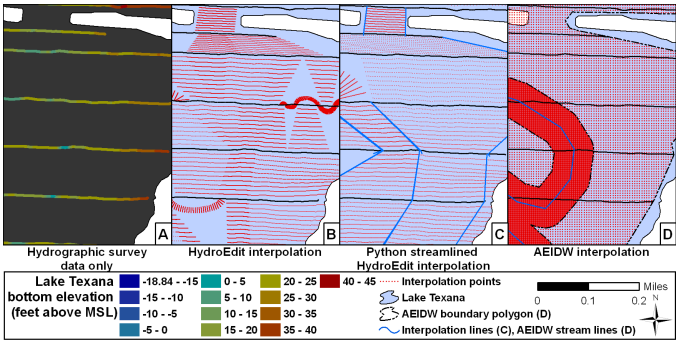

Fig. 6: Comparison of artificial survey point density

Figures 7, 8 and 9, compare bathymetric contour maps of the lower, upper part of the lakes and as well as an area of high channel sinuousity respectively. The comparisons of the lower and upper regions of the lake show that all three methods capture the major features of the lake reasonably well. HydroEdit, AEIDW do an excellent job of delineating the main stem river channel along with its sinuosity, while line-automated Hydroedit is able to capture the major features but not the details. This becomes even more evident when we look at Figure 9, here it can be seen that HydroEdit and AEIDW are able to correctly connect the deep areas in the original survey data into the sinous relic stream channel that can be seen on the USGS topographical maps.

Differences can also be seen between all three methods near the lake boundaries. This is due to a difference in the current

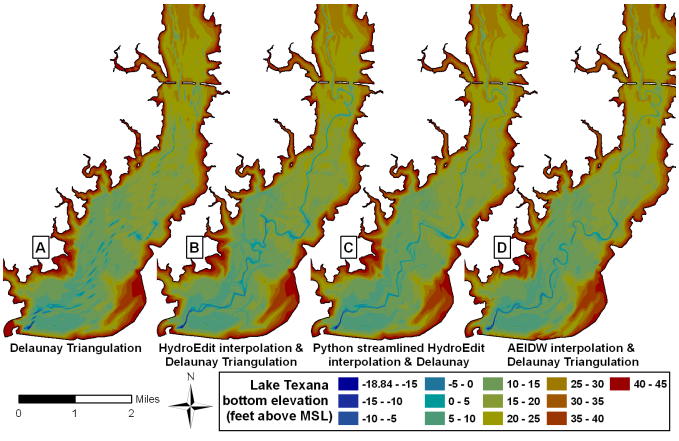

Fig. 7: Comparison of Interpolation Methods for the lower part of Lake Texana

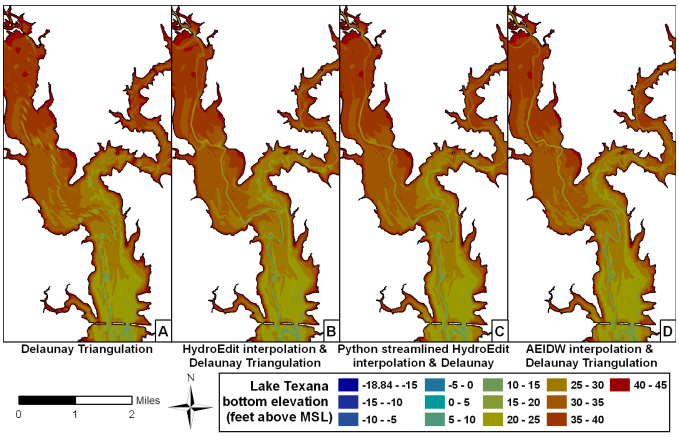

Fig. 8: *Comparison of Interpolation Methods for the upper part of Lake Texana*

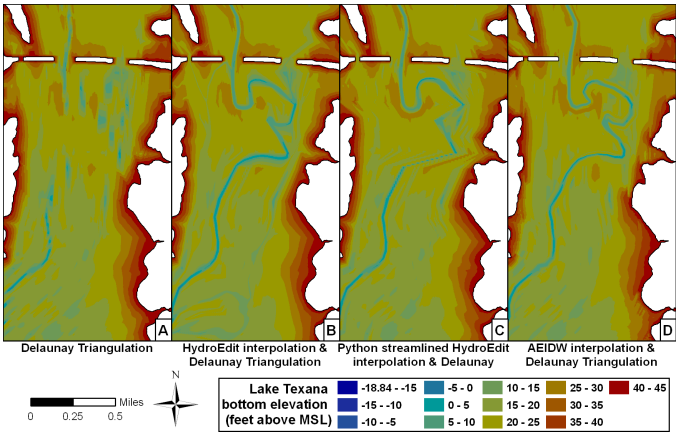

Fig. 9: *Comparison of interpolation methods for a section of Lake Texana with a sinous channel*

omplementations of extrapolations to the shore between the three methods and does not effect the volume estimates significantly.

Analysis shows a 63% reduction of processing time by using the line-automated HydroEdit method for Lake Texana when compared to the original HydroEdit method. Using the AEIDW method resulted in a 91% and 76% reduction of processing time when when compared to the original HydroEdit and line-automated Hydroedit methods. A summary and comparison table is presented in Table 1. The table shows that all three methods add volume to the lake volumetric estimate. The volume added by the line-automated HydroEdit method is lower probably due to it not capturing much of the sinousity of the relic stream channel. The remaining variance between AIEDW and HydroEdit can be explained by differences in the way boundaries have been handled.

Interpolation Method	Volume (acre-feet)	Increase in Lake Volume	Hours for completion
Delaunay Triangulation	156,283	<-->	0
HydroEdit	161,139	3.27%	90
Line-automated HydroEdit	159,845	2.28%	33
AEIDW (eccentricity=1/25)	161,693	3.46%	8

TABLE 1: *Comparison of interpolation methods.*

Conclusions

The pure python AEIDW method for estimating lake volumes shows a drastic increase in post-survey processing efficiency when compared to both the original HydroEdit method and the line-automated HydroEdit. In addition, the new technique is completely independant of the exact survey line locations, being defined completely by a best available description of lake morphology. This enhances the efficieny and accuracy of volume estimates of repeat surveys of the same lake, thus also improving sedimentation rate analyses.

The original HydroEdit fortran codebase ran over 10,000 lines of code (loc). by using available scientific, GIS and file handling modules available in Python the new suite of python tools being used for lake hydrographic survey analysis runs less than 1000 loc, besides being much easier for new staff to pick. This order of magnitude reduction in code complexity has allowed the the TWDB hydrosurvey program to rapidly innovate new techniques to improve the efficieny and accuracy of lake hydrographic surveys.

References

[ESRI95] Environmental Systems Research Institute, 1995, *ARC/INFO Surface Modeling and Display, TIN Users Guide*, ESRI, 380 New York Street, Redlands, CA 92373.

[Furn06] Furnans, Jordan, 2006, *HydroEdit User's Manual*, Texas Water Development Board.

[Furn08] Furnans, J. and B. Austin, 2008, *Hydrographic survey methods for determining reservoir volume*, Environmental Modelling and Software, Volume 23, Issue 2, February 2008, Pages 139-146, ISSN 1364-8152, DOI: 10.1016/j.envsoft.2007.05.011.

[Furn10] Furnans, J., D. Pothina, T. McEwen, and B. Austin, *Hydrographic Survey Porgram Assessment*, Texas Water Development Board, Austin, Texas 78711.

[Merw04] Merwade V. and D. Maidment, *A GIS framework for describing river channel bathymetry*, Center for Research in Water Resources, J.J. Pickle Research Campus, University of Texas at Austin, Austin, TX 78712.

[Merw06] Merwade V. M., Maidment D. R., and Goff J. A., Anisotropic considerations while interpolating river channel bathymetry. Journal of Hydrology, Vol.331(3-4), pp. 731-741 (2006).

[TWDB74] TWDB (Texas Water Development Board), 1974, *Iron Bridge Dam and Lake Texana*, Report 126, Engineering Data on Dams and Reservoirs in Texas, Part 1.

[TWDB06] TWDB (Texas Water Development Board), 2006, *Volumetric Survey of Lake Kemp*, TWDB, Austin, Texas 78711.

[TWDB09] TWDB (Texas Water Development Board), 2009, *Volumetric Survey of Aquilla Lake*, TWDB, Austin, Texas 78711.

[TWDB09b] TWDB (Texas Water Development Board), 2009b, *Volumetric Survey of Lady Bird Lake*, TWDB, Austin, Texas 78711.