Proceedings of the 9th

Python in Science Conference

June 28 - July 3 • Austin, Texas

Stéfan van der Walt
Jarrod Millman

PROCEEDINGS OF THE 9TH PYTHON IN SCIENCE CONFERENCE

Edited by Stéfan van der Walt and Jarrod Millman.

SciPy 2010
Austin, Texas
June 28 - July 3, 2010

ISSN:2575-9752
https://doi.org/10.25080/Majora-92bf1922-012

ORGANIZATION

Conference Chairs

ERIC JONES, Enthought, Inc., USA
JARROD MILLMAN, Neuroscience Institute, UC Berkeley, USA

Program Chair

STÉFAN VAN DER WALT, Applied Mathematics, Stellenbosch University, South Africa

Program Committee

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
DAVID COURNAPEAU, School of Informatics, Kyoto University, Japan
DARREN DALE, Cornell High Energy Synchrotron Source, Cornell University, USA
FERNANDO PÉREZ, Neuroscience Institute, UC Berkeley, USA
PRABHU RAMACHANDRAN, Aerospace Engineering, IIT Bombay, India
JAMES TURNER, Gemini Observatory, Chile
WARREN WECKESSER, Enthought, Inc., USA

Tutorial Chair

BRIAN GRANGER, Physics, CalPoly, USA

Student Sponsorship Chair

TRAVIS OLIPHANT, Enthought, Inc., USA

Biomedical/Bioinformatics Track Chair

GLEN OTERO, Dell, USA

Parallel Processing/Cloud Computing Track Chairs

KEN ELKABANY, PiCloud, USA
BRIAN GRANGER, Physics, CalPoly, USA

Proceedings Reviewers

MATTHEW BRETT, Neuroscience Institute, UC Berkeley, USA
C. TITUS BROWN, Computer Science and Biology, Michigan State, USA
DAMIAN EADS, Computer Science, UC Santa Cruz, USA
RALF GOMMERS, Healthcare Department, Philips Research Asia, Shanghai, China
EMMANUELLE GOUILLART, CNRS Saint Gobain, France
YAROSLAV O. HALCHENKO, PBS Department, Dartmouth College, USA
JOSH HEMANN, Marketing Analytics and Business Strategy, Sports Authority, USA
NEIL MULLER, Applied Mathematics, University of Stellenbosch, South Africa
EMANUELE OLIVETTI, NeuroInformatics Laboratory, Bruno Kessler Foundation and University of Trento, Italy
DAG SVERRE SELJEBOTN, Theoretical Astrophysics, University of Oslo, Norway
ANDREW STRAW, Molecular Pathology, Austria
JAMES TURNER, Data Processing Software Group, Gemini Observatory, Chile

Program Staff

AMENITY APPLEWHITE, Enthought, Inc., USA
JODI HAVRANEK, Enthought, Inc., USA
LEAH JONES, Enthought, Inc., USA

SCHOLARSHIP RECIPIENTS

ELAINE ANGELO, Harvard University
JAMES BERGSTRA, University of Montreal
KADAMBARI DEVARAJAN, University of Illinois
GERARDO GUTIERREZ, Universidad de Antioquia
RYAN MAY, University of Oklahoma
KRISTOPHER OVERHOLT, University of Texas
FABIAN PEDREGOSA, University of Granada
NICOLAS PINTO, Massachusetts Institute of Technology
SKIPPER SEABOLD, American University
KURT SMITH, University of Wisconsin
STÉFAN VAN DER WALT, Stellenbosch University
DAVID WARDE FARLEY, University of Toronto
OMAR ANDRES ZAPATA MESA, Universidad de Antioquia

JUMP TRADING AND NUMFOCUS DIVERSITY SCHOLARSHIP RECIPIENTS

,

CONTENTS

Keeping the Chandra Satellite Cool with Python

Tom Aldcroft‡*

Abstract—The Chandra X-ray Observatory has been providing groundbreaking astronomical data since its launch by NASA in July of 1999. Now starting the second decade of science the Chandra operations team has been using Python to create predictive thermal models of key spacecraft components. These models are being used in the mission planning and command load review process to ensure that the series of planned observations and attitudes for each week will maintain a safe thermal environment. Speaking from my perspective as a scientist working to create and calibrate the models, I will discuss the process and the key off-the-shelf tools that made it all possible. This includes fitting many-parameter models with the Sherpa package, parallel computation with mpi4py/MPICH2, large table manipulations with pytables/HDF5, and of course fast array math with NumPy.

Index Terms—telescope, NASA, MPI, astronomy, control

Motivation

This paper describes the use of off-the-shelf tools in Python to tackle a relatively challenging engineering problem facing the Chandra X-ray Observatory satellite [CHANDRA]. While presenting no fundamentally new algorithms or packages, the goal here is to take this as a case study for understanding how scientists and engineers can make Python the foundation of their analysis toolkit.

Chandra

The Chandra satellite was launched in July of 1999 as one of NASA's four "Great Observatories". This satellite can be compared in size and scope to the Hubble Space Telescope except that it views the universe in X-rays instead of optical or UV light. Some people will argue that the universe is a much more exciting place when viewed with X-ray photons, for then the sky comes alive with black holes, supernova explosions, massive galaxy clusters, pulsars, and many other highly energetic phenomena.

Early in the mission it became apparent that temperatures on the spacecraft, particularly on the side which normally faces the Sun, were increasing at a rate well above pre-launch predictions. It turned out that the ionizing particle radiation environment was higher than expected and that it was degrading the silverized teflon insulation which wraps much of the spacecraft. Since this time the constraint of keeping spacecraft components within safe operating temperatures has been a major driver in operations and schedule planning. Note that Chandra is in a high elliptical orbit, so unlike Hubble no repair is possible.

Fig. 1: *Artist's rendering of the Chandra X-ray satellite. The silverized teflon which wraps the spacecraft has degraded so it is now far less reflective than shown.*

Different parts of the spacecraft are heated at different pitch angles (the angle that the telescope boresight axis makes to the sun line). This is shown in Figure 2 which presents a side view of Chandra along with the subsystems that are sensitive over different pitch ranges. Temperatures can be maintained within limits by ping-ponging to different attitudes, never letting one part get too hot. Thus in order to plan the weekly observing schedule a few simple models were created to predict temperatures or otherwise constrain the observing duration at certain attitudes.

As the constraints became more significant a need developed to improve the models in order to maintain the highest scientific output without endangering the spacecraft. In response, about three years ago the CXC Science Operations Team (SOT, scientists closely involved in satellite operations) and the engineers of the Flight Operations Team (FOT) formed a joint working group to study thermal issues and to develop higher fidelity thermal models. This paper discusses one facet of the results coming out of the thermal modeling working group.

Early in the process the author chose Python as the programming language for supporting this effort. Around this time NumPy had emerged as a strong (and consolidated) numeric array manipulation tool for Python. Adding in IPython, Matplotlib and SciPy provided a development and interactive analysis environment that was ideal for the task.

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

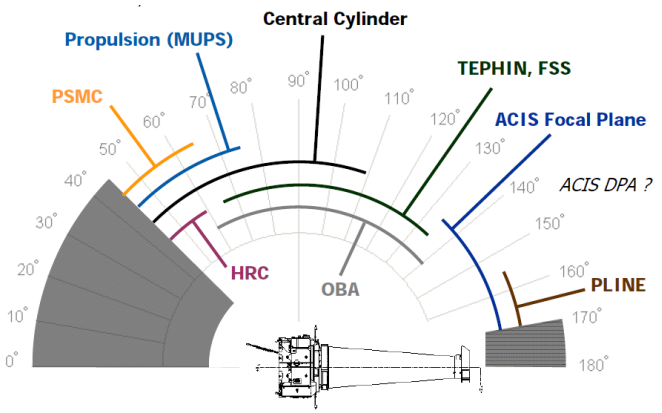


Fig. 2: Thermal constraint regions depending on the pitch angle to the sun line.

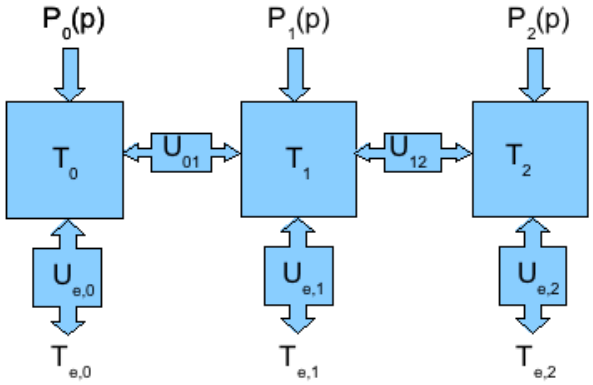


Fig. 3: Schematic diagram of the thermal Chandra thermal model. Boxes (T_0, T_1, T_2) represent physical nodes on the spacecraft where a thermistor is located. External solar heat input is shown as $P_i(p)$, conductances are $U_{i,j}$, and external heat bath temperatures are $T_{e,i}$.

Telemetry access

A key starting point for developing complex thermal models is fast access to thermal and other engineering data from the satellite. The Chandra satellite records a 32 kbit/s stream of telemetry that contains both the science data and satellite engineering (housekeeping) data. Although the data volumes are meager by modern standards, the tools and data structure used for processing and access were largely designed with mid-1990's era hardware and storage capabilities in mind.

Two standard methods exist for retrieving archival telemetry. Within the Flight Operations Team (FOT) environment at the Operations Control Center the primary tool stores no intermediate products and instead always falls back to the raw telemetry stream. This stream contains over 6000 individual engineering telemetry items (MSIDs) that are interleaved according to different sampling rates and sub-formats. FOT engineers using this system are accustomed to waiting hours or more to retrieve a year of data needed for analysis.

Within the Chandra X-ray Center (CXC) which is responsible for processing the science data, the situation is somewhat better. In this case the "Level-0 decommutation" process is done just once and the results stored in FITS [FITS] files available through an archive server. These files each contain about two hours of data for MSIDs that are related by subsystem (thermal, pointing control, etc) and sampling rate. However, access to a single MSID of interest (e.g. a particular thermistor readout) requires retrieving and unzipping a large amount of uninteresting data.

The solution to this problem was found in the pytables [PYT] package which provides a robust interface to the powerful Hierarchical Data Format [HDF5] library. Pytables/HDF5 is designed to create, update, and access very large data tables with ease. The key here was creating a separate HDF5 table for each MSID which could easily store all the readouts for that MSID for the *entire* mission. This is especially optimal because many of the MSIDs change infrequently and thus compress very well. HDF5 natively supports an assortment of compression options which makes this a snap. Initially creating the table based on a NumPy data array is simple using the `createEArray` method to create an extendible homogenous dataset:

```
filts = tables.Filters(complevel=5, complib='zlib')
h5 = tables.openFile(filename, mode='w', filters=filts)
h5shape = (0,) + data.shape[1:]
h5type = tables.Atom.from_dtype(data.dtype)
h5.createEArray(h5.root, 'data', h5type, h5shape,
                title=colname, expectedrows=n_rows)
h5.createEArray(h5.root, 'quality', tables.BoolAtom(),
           (0,), title='Quality', expectedrows=n_rows)
h5.close()
```

A minor complication seen here is the boolean `quality` table which accounts for bad or missing telemetry. Once the table has been created it is a simple matter to extend it with new data values after a communication pass:

```
h5 = tables.openFile(filename, mode='a')
h5.root.data.append(new_data)
h5.root.quality.append(new_quality)
h5.close()
```

At this time the largest individual tables have about 1.3×10^9 rows (for the highest sampling rate of 4 times per second). The data retrieval speed from this archive of HDF5 tables is approximately 10^7 items per second. This means that typical data retrieval requests can be handled in seconds rather than hours. Such an improvement changes the landscape of questions that can be asked and then answered.

In addition to the data acquisition back-end, a user-friendly front-end was needed to access the telemetry data in the HDF5 archive. A challenge in this regard was that most of the intended user community (FOT engineers) had absolutely no experience with Python. Thus the interface, documentation and examples had to be clear and explicit. The final documentation package included a tutorial covering the telemetry access interface as well as IPython, NumPy, and Matplotlib.

Creating a thermal model

The thermal model which was developed for modeling Chandra subsystems is illustrated in Figure 3.

Here each of the boxes (T_0, T_1, T_2) represents a physical node on the spacecraft where a thermistor is located. It is then assumed that each node i has an external heat input $P_i(p)$ and has conductances $U_{i,j}$ to other nodes and an external heat bath with temperature $T_{e,i}$. For most models the external heat input is Solar and depends purely on the spacecraft pitch angle with respect to the Sun. In some cases, however, the heat input due to internal electronics is also included. Given these definitions and the nodal connectivity the temperatures can be written in matrix form as a

simple first order differential equation:

$$\begin{aligned}
\dot{\mathbf{T}} &= \tilde{\mathbf{A}}\mathbf{T} + \mathbf{b} \\
\mathbf{T}(t) &= \int_0^t e^{\tilde{\mathbf{A}}(t-u)}\mathbf{b}du + e^{\tilde{\mathbf{A}}t}\mathbf{T}(0) \\
&= [\mathbf{v}_1\ \mathbf{v}_2]\begin{bmatrix} \frac{e^{\lambda_1 t}-1}{\lambda_1} & 0 \\ 0 & \frac{e^{\lambda_2 t}-1}{\lambda_2} \end{bmatrix}[\mathbf{v}_1\ \mathbf{v}_2]^{-1}\mathbf{b} \\
&\quad + [\mathbf{v}_1\ \mathbf{v}_2]\begin{bmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{bmatrix}[\mathbf{v}_1\ \mathbf{v}_2]^{-1}\mathbf{T}(0)
\end{aligned}$$

Here $\mathbf{T}$ is a vector of node temperatures, $\tilde{\mathbf{A}}$ is the matrix describing the coupling between nodes, $\mathbf{b}$ is a vector describing the heat inputs, $\mathbf{v}_i$ and λ_i are the eigenvectors and eigenvalues of $\tilde{\mathbf{A}}$, and t is time.

The solution can be expressed analytically as long as the model parameters (external heat inputs, conductances) are constant. Most of the time Chandra dwells at a particular attitude and so this is a good assumption during such a dwell. The computational strategy for making a model prediction of temperatures is to identify "states" where the parameters are constant and propagate temperatures from the beginning to the end of the state, then use the end temperatures as the starting point for the next state.

The first implementation of this core model calculation was a literal transcription of the analytic solution for each time step within a state. This was quite inefficient because of repeated creation and computation of intermediate 2-d arrays. A slight modification allowed for adding the time dimension into the arrays and computing all time steps at once with a single expression of NumPy dot products. This resulted in a factor of 10-20 speed increase. Further optimization to avoid repeating certain calculations within inner loops plus caching of results eventually yielded code that is 50 times faster than in the initial literal version. In the end the code takes less than a second to predict a year of temperatures at 5-minute resolution for a 5-node model of the sun-pointed side of the spacecraft.

Fitting the model parameters

The next step is to tune the model parameters to best fit the existing thermal data for the subsystem of interest. In typical cases there are two to five thermistors whose data are averaged over 5 minute intervals. Up to five years of such data are fit at once.

What is not immediately apparent in the concise matrix formulation $\dot{\mathbf{T}} = \tilde{\mathbf{A}}\mathbf{T} + \mathbf{b}$ of the thermal model is that it contains a lot of free parameters. In addition to the conductances and external heat bath temperatures, the external Solar power input for each node is complicated. First it is a function of the pitch angle with respect to the Sun, but it also has an annual variation term (due to the elliptical orbit) as well as a long-term change due to the continued slow degradation of the protective insulation. All this needs to be fit in order to predict temperature profiles at any time, including years in advance. One key 5-node model being used in planning operations has a total of 80 free parameters. All of those parameters need to calibrated using at least 5 years of existing thermal data to train the model.

Two immediate objections can be raised. First, that with so many free parameters one can fit almost anything. In a sense for this application that is just fine, as long as the resultant model has stable predictive power beyond the time range over which it is calibrated. But at a more fundamental level experience has shown that it is simply not true that the complex and coupled time-dependent behavior of temperatures on the spacecraft can necessarily be captured by any model with a large number of parameters.

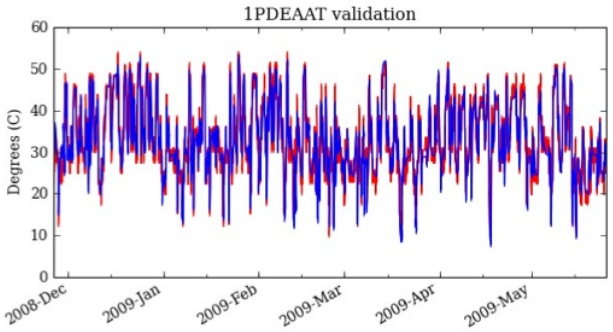


Fig. 4: *Long-term comparison of the actual spacecraft thermistor data (red) with the model prediction (blue). This span of data is used for fitting the model coefficients.*

The second objection is that fitting for so many parameters is bound for failure. However, what makes this problem tractable is that many of the parameters are only loosely coupled. This makes it possible to selectively fit for subsets of the parameters and iteratively home in on a reasonable global set of parameters. Unlike many problems in parameter estimation where the derived parameter values and uncertainties are the primary goal, in this case it is the model prediction that matters.

The Sherpa [SHP] package is used to handle the actual optimization of parameters to achieve the best model fit to the data. Sherpa is a modeling and fitting application for Python that contains a powerful language for combining simple models into complex expressions that can be fit to the data using a variety of statistics and optimization methods. It is easily extendible to include user models, statistics and optimization methods. For this application the key feature is a robust implementation of the Nelder-Mead (aka Simplex) optimization method that is able to handle many free parameters. Sherpa provides within the model language a natural way of manipulating and linking model parameters using Python expressions.

The result of the fitting process is a calibrated thermal model that can be used to accurately predict the system temperatures given the planned sequence of maneuvers and instrument configurations. Figure 4 shows an example of the data for one thermistor "1PDEAAT" in red with the model prediction in blue.

Figure 5 now shows the post-facto model prediction (blue) for a two-week period of data (red) that is outside the calibration time range. Most of the features are well reproduced and the distribution of residuals is roughly gaussian.

Parallelization of fitting

Despite the good model calculation performance with vectorized NumPy, fitting for 5 years of data and dozens of parameters can benefit from the further speed increase of parallelization. This is particularly helpful for the exploratory phase of developing a new model and getting the parameters in the right ball park.

The thermal models being discussed here can easily be parallelized by splitting into independent time segments. There is a slight issue with the starting conditions for each segment, but there are straightforward ways to finesse this problem. In the context of a fitting application a master-worker architecture works well. Here the master is responsible for controlling the fit

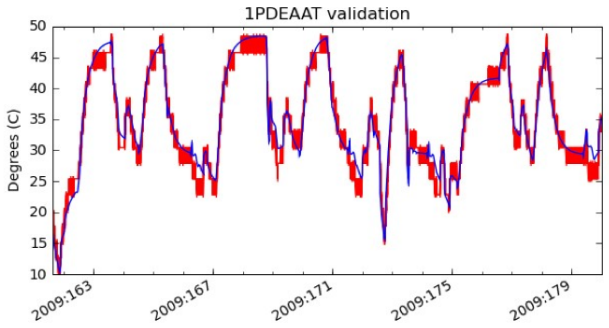


Fig. 5: *Detailed comparison of the actual spacecraft thermistor data (red) with the model prediction (blue). The thermistor is located within the power-supply box for one of the main science instruments.*

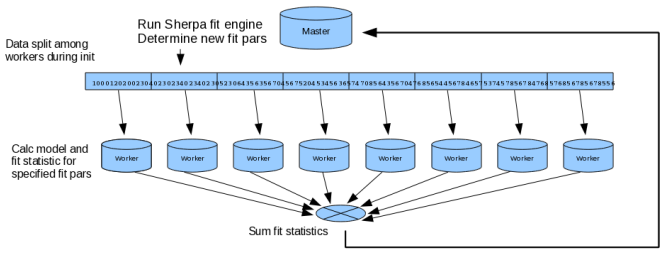


Fig. 6: *Schematic illustration of parallelizing the fitting process by breaking the data and model generation into smaller time slices.*

optimization process while each of the workers takes care of all model computations for a particular time segment. The worker is initially sent the time range and model definition and it is then responsible for retrieving the appropriate telemetry data. After initialization the model parameters for each fit iteration are sent and the worker computes the model and Chi^2 fit statistic. All of the individual Chi^2 values are then summed. In this way the communication overhead between master and workers is minimal. Figure 6 illustrates the process.

The actual job of handling the interprocess communication and job creation is done with the mpi4py [MPI4PY] package using the MPICH2 [MPICH2] library. As is often the case, the choice of these particular packages over other similar ones was driven by the depth of documentation, availability of relevant looking examples, and ease of installation. Starting with no previous experience with distributed computing, a working prototype of the parallel fitting code was created in less than a day. This is a testament largely to the quality of documentation.

As for computing resources, our division within SAO is perhaps like other academic science institutes with a collection of similarly configured linux machines on a local network. These are often available off-hours for "borrowing" CPU cycles with consent of the primary user. A more formal arrangement (for instance using an application like Condor for distributed job scheduling) has been in consideration but not yet adopted. For this application up to twelve 4-core machines were used. Dynamic worker creation was supported by first starting up `mpd` servers on the target hosts (from file `mpd.hosts`) with a command like the following:

```
mpdboot --totalnum=12 --file=mpd.hosts --maxbranch=12
```

An abridged version of three key functions in the main parallel fitting code is shown below. These functions support communication with and control of the workers:

```
def init_workers(metadata)
    """Init workers using values in metadata dict"""
    msg = {'cmd': 'init', 'metadata': metadata}
    comm.bcast(msg, root=MPI.ROOT)

def calc_model(pars):
    """Broadcast a message to each worker to calculate
       the model for given pars."""
    comm.bcast(msg={'cmd': 'calc_model', 'pars': pars},
               root=MPI.ROOT)

def calc_stat()
    """Broadcast message to calculate chi^2 diff between
       model and data.  After that collect the sum of
       results from workers using the Reduce function."""
    msg = {'cmd': 'calc_statistic'}
    comm.bcast(msg, root=MPI.ROOT)
    fit_stat = numpy.array(0.0, 'd')
    comm.Reduce(None, [fit_stat, MPI.DOUBLE],
                op=MPI.SUM, root=MPI.ROOT)
    return fit_stat
```

After defining the above functions the main processing code first uses the MPI Spawn method to dynamically create the desired number of worker instances via the previously created `mpd` servers. Then the workers receive an initialization command with the start and stop date of the data being used in fitting. The Sherpa user model and fit statistic are configured, and finally the Sherpa fit command is executed:

```
comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['fit_worker.py'],
                           maxprocs=12)
init_workers({"start": date_start, "stop": date_stop})

# Sherpa commands to register and configure a function
# as a user model for fitting to the data.
load_user_model(calc_model, 'mpimod')
set_model(mpimod)

# Set function to be called to calculate fit statistic
load_user_stat('mpistat', calc_stat)
set_stat(mpistat)

# Do the fit
fit()
```

The `fit_worker.py` code is likewise straightforward. First get a communication object to receive messages, then simply wait for messages with the expected commands. The `init` command calls the `get_data()` function that gets the appropriate data given the `metadata` values and the `rank` of this worker within the ensemble of `size` workers.

```
comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

while True:
    msg = comm.bcast(None, root=0)

    if msg['cmd'] == 'stop':
        break

    elif msg['cmd'] == 'init':
        # Get the vectors of times and temperatures
        # for this worker node
        x, y = get_data(msg['metadata'], rank, size)

    elif msg['cmd'] == 'calc_model':
        # Calculate the thermal model for times
        # covered by this worker
        model = worker_calc_model(msg['pars'], x, y)

    elif msg['cmd'] == 'calc_statistic':
        # Calculate the chi^2 fit statistic and send
```

```
        # back to the master process
        fit_stat = numpy.sum((y - model)**2)
        comm.Reduce([fit_stat, MPI.DOUBLE], None,
                    op=MPI.SUM, root=0)
comm.Disconnect()
```

Putting it to work

Using the techniques and tools just described, two flight-certified implementations of the models have been created and are being used in Chandra operations. One models the temperature of the power supply for the ACIS science instrument [ACIS]. The other models five temperatures on the Sun-pointed side of the forward structure that surrounds the X-ray mirror. Each week, as the schedule of observations for the following week is assembled the models are used to confirm that no thermal limits are violated. Separate cron jobs also run daily to perform post-facto "predictions" of temperatures for the previous three weeks. These are compared to actual telemetry and provide warning if the spacecraft behavior is drifting away from the existing model calibration.

Summary

The current Python ecosystem provides a strong platform for production science and engineering analysis. This paper discussed the specific case of developing thermal models for subsystems of the Chandra X-ray Observatory satellite. These models are now being used as part of the flight operations process.

In addition to the core tools (NumPy, IPython, Matplotlib, SciPy) that get used nearly every day in the author's work, two additional packages were discussed:

- Pytables / HDF5 is an easy way to handle the very large tables that are becoming more common in science analysis (particularly astronomy). It is simple to install and use and brings high performance to scientists.
- MPI for Python (mpi4py) with the MPICH2 library provides an accessible mechanism for parallelization of compute-intensive tasks.

Acknowledgments

Thanks to the reviewer James Turner for a detailed evaluation and helpful comments.

REFERENCES

[ACIS] http://cxc.harvard.edu/proposer/POG/html/ACIS.html
[CHANDRA] http://chandra.harvard.edu/
[FITS] http://fits.gsfc.nasa.gov/
[HDF5] http://www.hdfgroup.org/HDF5/
[MPI4PY] http://mpi4py.scipy.org/
[MPICH2] http://www.mcs.anl.gov/research/projects/mpich2/
[PYT] http://www.pytables.org
[SHP] http://cxc.harvard.edu/contrib/sherpa

Astrodata

Craig Allen[‡*]

Abstract—The astrodata package is a part of the Gemini Telescope's python-based Data Reduction Suite. It is designed to help us deal in a normalized way with data from a variety of instruments and instrument-modes. All Gemini specific code configurations are isolated in configuration packages separate from the astrodata source. The configuration packages define a lexicon of terms associated with a family of dataset types and implements the behaviors associated with each terms.

Index Terms—Python, Scientific Computing

The Problem Domain: Handling Data Across Instrument-Modes

Gemini

Gemini Observatory is a multinational partnership which operates two telescopes, Gemini North from Hilo, Hawaii, and Gemini South from La Serena, Chile. We mount multiple instruments simultaneously, and have a suite of instruments which rotate onto the telescopes periodically. These instruments have been made by a variety of different teams and institutions from our partner countries.

Multi-Extension FITS

Gemini Observatory relies on a file format called "Multi-Extension FITS" (MEF) format to store all datasets, one standard all instruments obey. MEF is a common file format in Astronomy and is an extension of the older "single extension" FITS file format. Signle extension FITS files consited of a single expandable ASCII header section and single binary data section. MEF extends this so that the file appears as a list of such header-data units (HDUs).

The FITS standard contains definitions for standardized metadata in HDU headers. For example, standard header keys are defined for the telescope, observer, object name, the RA and DEC, and some other properties one expects to be associated with an astronomical observation. There are also suffient standardized headers to describe the binary data section such as needed to load it, such as its dimensions and pixel type (if it is pixel data). However, many other bits of metadata which are ubiquitous for Gemini data, such as "gain" and "filter name", do not have standard headers names in the FITS standard.

Since the FITS headers are expandable there is ample information in the datasets to retreive the desired information, but the retrieval is subject to incidental differences in naming and storage layout. While helpful, the list-like shape of the associated HDUList is merely one incremental improvement toward the goal of associating these related HDUs into a cohesive whole. MEF is limited in this regard by the fact that standard metadata to describe relationships between extensions have not been developed. Such metadata could for example introduce a heirarchical relationship and explicit dependencies among extensions. Metadata to infer such relationships does exist in the headers but for the reasons mentioned it tends to be instrument-mode specific.

The problem domain is dominated by processes which are conceptually the same across instrument-modes from the perspective of the user and scientist, but which require implementations unique to the instrument-mode.

Removing Incidental Differences

Our goal with a new abstraction higher than the level of the "HDU list" is to remove the incidental differences for the user of the new abstraction, and move handling of the differences into type-specific compartments. Within the compartments it is safe to make type-specific assumptions and use mode-specific heuristics without compromising the generality of the system.

We also seek to extend the scopes at which a particular difference can be considered "incidental". For example, in the case of dataset transformation (reduction), most instrument-modes involve a step to "subtract the sky". This is generally done by taking a picture of the sky near the object and literally subtracting its pixel values from the exposure. However, the details of doing sky subtraction do depend on the particular instrument-mode. There exist important differences between imaging and spectroscopy, and between different wavelengths, which means performing this step is type-dependent.

Nevertheless, at the higher level of consideration, and scientifically, the step is "the same". Thus while the differences regarding how one performs the sky subtraction are not, ultimately, to be accurately described as "incidental", they can still be generalized over. At some scopes the differences are not considered significant so long as they are performed *properly*.

At the same time, we want to maintain flexibility about which scopes we commit to implementing in either generic or specific ways. We want safe refactoring paths available so that we can, for example, safely integrate instrument-mode specific code into generalized code when possible. But we also want a system that allows patching an instrument-mode specific solution over a general solution that may be failing for that instrument mode, as a quick way to address problems to for example support time-critical nightime operation.

These goal are accomplished by the adopting of a core classification system which is used to assign behavior to dataset types

* *Corresponding author: callen@gemini.edu*
‡ *Gemini Observatory*

which can be general, e.g. "GEMINI", or a specific instrument mode, "GMOS_IFU".

Incidental Dataset Differences Normalized

- differences in how a dataset is recognized as a particular type
- differences in low level metadata
- differences in the implementation of scientifically similar transformations
- differences in storage structure

The AstroData Class

MEF I/O

To load a MEF into an AstroData instance one generally gives the filename to the AstroData contructor:

```
ad = AstroData("trim_gN20091027S0133.fits")
```

The instance, referenced in the `ad` variable will have loaded the extensions' headers and have detected type information. However, the data, represented as a numpy array, is not loaded until referenced. AstroData relies on the python Pyfits Library for this low level MEF access. Like the HDUList object that pyfits returns, the AstroData instance is also iterable, behaving as though consisting of a collection of AstroData instances, one per HDU in the MEF.

To iterate over the list one would write a loop as so:

```
for ext in ad:
    print ext.infoStr()
```

In this case `ext` is also an AstroData instance, created by the iteration-related members of AstroData. The `ext` instance shares its single HDU member with the original `ad` instance, as well as its primary header unit, but has its own HDUlist object. This means changes to the shared information will be reflected in the outer AstroData object, `ad`, but that new HDUs appended to `ext` would not be appended to `ad`.

This behavior extends the general behavior of numpy and pyfits, and is considered desirable so that it is possible to avoid unnecessary copying of memory, but use of the feature does require care.

The `__getitem__(..)` member of AstroData is overridden. It creates and returns an AstroData instance with a new HDUList containing the HDU(s) identified in the argument, i.e.

```
adsci = ad["SCI"]
```

This call to `__getitem__(..)` uses causes it to use the extension naming information to find all extensions with the name "SCI", and return an AstroData instance containing just those found.

Astrodata Grammar

Breaking our knowledge of our datasets into parts involves creating a language of terms for our family of datasets. The terms defined will belong to a grammar understood by the astrodata package, of course. This family of terms, or language, turns out to be valuable in general as a tool to discuss dataflow, separately from the implementation details. It turns out we have good reason to understand what the terms *mean* prior deciding how they will perform the action satisfying that meaning.

In practice, the developer of a type-family will work interatively, creating and testing continually improved versions of their configuration package. This sort of iterative "test and refactor" process is well supported by the astrodata package, and supporting "refactoring paths" is part of our intent. Conceptually, however, the work to define the meaning of the terms is logically prior to implementation. These definitions are, in fact, the conceptual specification for all configuration implementations.

The astrodata grammar is expandable, but at this time consists of three primary types of term:

- the dataset types: *AstroData Types*
  - e.g. GMOS_SPECT is "a GMOS dataset taken in any spectroscopic mode"
- high level metadata: *Descriptors*
  - e.g. "filter_name" is "a string value concatenation of all filters in the light path during the exposure"
- transformations: *Primitives*
  - e.g. "skySubtract" is "a transformation where sky conditions at the time of the observation are subtracted from the exposure"

Each of these terms, once defined, will have a specific behavior associated:

- for *AstroData Type*: code to recognize the type of dataset based on low level metadata
- for *AstroData Descriptors*: code to calculate and return the high level-metadata from the low-level metadata
- for *Primitives*: code to perform the tranformation

AstroData Type

From the user of astrodata's point of view, AstroData Types are string names accessed through AstroData members. The objects used to detect the type criteria and assign the names to the AstroData object are hidden within the RecipeLibrary which AstroData uses to provide type features. The DataClassification objects which load the type definition, also check to see if it applies to a given HDUList object.

Many features are assigned to datasets by AstroData Type, such that behind a common name lies implicitly type-specific behavior. Different implementations of what is conceptually the same descriptor, or primitive, are assigned to the same descriptor or primitive *name*, meaning the interfaces to invoking them are regular. Since the descriptor or high-level metadata system requires the dataset type to know which particular *descriptor calculator* to load, the type system cannot in turn rely on high-level metadata to recognize datasets, as that would be circular. Thus, the classification system uses low level metadata, ideally from the PHU, which is the 0-th HDU in the HDUList.

A typical type definition is stored as a class descending from astodata's DataClassification class, allowing it the ability to overwrite the base methods if need be. However, the general intention is that in the typical case the known members of the DataClassification parent are set in the child class so the definition is essentially a data structure used by the parent class. Members of DataClassification parent class execute the type check.

Here is a relatively typical type definition from our type library, in this case for GMOS_IMAGE:

```
class GMOS_IMAGE(DataClassification):
  name="GMOS_IMAGE"
  usage = """Any datset from the GMOS_N or GMOS_S
             instruments."""
  parent = "GMOS"
  requirement = ISCLASS("GMOS") & PHU(GRATING="MIRROR")
```

Name,

The `name` member specifies the string name used to identify this type.

Usage

The `usage` member is a printable string containing information about the meaning of the type, used to generate documentation.

Parent

The `parent` member is the string name of the parent classification, if any. The `parent` member is used to build the overall classification hierarchy. This hierarchy is in turn used to resolve conflicts in feature assignments, children overriding parents.

Requirements

The `requirement` member contains a single instance of an astrodata *Requirement* class, which is how the classification actually checks the dataset in question. Use Requirement subclasses for logical operators allows the appearance of compound requirments using *and*, *or*, and *not*.

Specific checks are performed by the ISCLASS and PHU Requirement subclasses which, respectively, check for adherance to another type definition, and check primary header unit headers for key-value matches. The all caps naming convention was adopted to help these classes stand out due to their peculiar use in the classification definitions.

PHU Requirements:: The PHU class is a Requirement subclass which ultimately is the workhorse of the system. Classifications generally resolve to sets of PHU header checks, since ideally, being the header for the dataset as a whole, the PHU will contain complete identifying information. The PHU constructor accepts a dictionary containing keys and values to check, or will roll one from its argument list. Values are regular expressions, keys are strings but allow modifiers to prohibit the specified match or to use regular expressions for matching keys as well as values.

ISCLASS Requirements:: ISCLASS in this example is a Requirement subclass which checks that the dataset in question is also the type named in the ISCLASS constructor argument. No hierarchical or other relationships are assumed due to the ISCLASS requirement. The classification named is considered as merely shorthand for whatever checks are associated with it.

Often the type specified in an ISCLASS Requirement will in fact be the parent, but this is not universally true. For example below, in the case of the base GMOS instrument type itself, the parent and requirement classes are distinct:

```
class GMOS(DataClassification):
  name="GMOS"
  usage = '''
      Applies to all data from either GMOS-North
      or GMOS-South instruments in any mode.
      '''
  parent = "GEMINI"
  requirement = ISCLASS("GMOS_N") | ISCLASS("GMOS_S")
```

Since the GMOS type is an abstraction meaning "from either GMOS North or GMOS South" this appears in the requirements. However, the parental relationship cannot be GMOS_N or GMOS_S as parent, because features such as Primitives or Descriptors, if assigned to GMOS_N, for example, would be intended to override the GMOS assignments. Instead, GMOS overrides the instrument-agnostic GEMINI type, which is therefore given as its explicit parent.

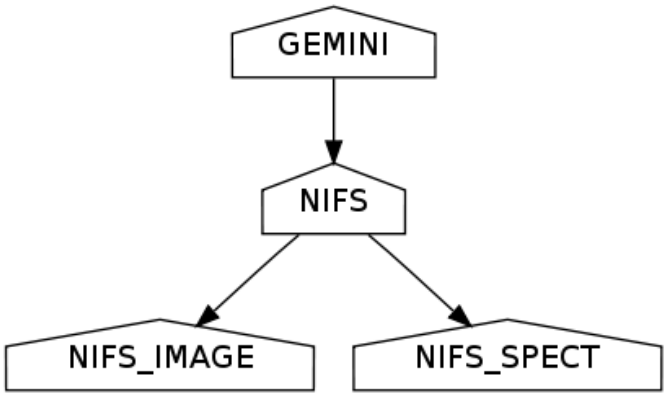


Fig. 1: NIFS Type Tree *A minimal type tree for the NIFS instrument: One each to identify the Instrument itself, its imaging and spectroscopic mode, as well as the general GEMINI type which acts as NIFS' parent type.*

Logical Requirements:: Three Requirement subclasses execute logical operations to combine other requirements, AND, OR, and NOT. These each override the "&", "|", and "!" operators respectively, for convienience. By design the constructors take a list of requirements to combine logically, though they will, again for convienience, roll the list from arguments.

Adding New Requirements:: This general arrangement allows easy addition of other types of requirement classes. We know, for example, that for some types we must detect we will have to create an "EHU" requirement object to check headers in data extensions.It will be a simple matter to add such a class and utlize it in combination with other requirement subclasses.

The DataClassification classes are passed the pyfits HDUList object to perform the detection and so have complete access to the dataset. Therefore, a classification can technically look at any characteristic of the data. However, by policy, for efficiency reasons we specifically do not look at pixel data.

Examples

Access to type (aka "classification") information goes throug the AstroData instance. The AstroData class relies internally on the Classification Library to provide type information:

```
>>> from astrodata import AstroData
>>> ad = AstroData("trim_gN20091027S0133.fits")
>>> ad.types
['GEMINI_NORTH', 'GEMINI', 'IMAGE', 'GMOS_N',
        'GMOS_IMAGE', 'GMOS', 'PREPARED']
```

Also, a single type can be checked in a call to the "isType" member of AstroData. The single line replaces groups of conditional checks that otherwise appear in reduction scripts at Gemini:

```
>>> ad.isType("GMOS_IMAGE")
True
```

This saves lines in scripts but more importantly, it centralizes the type checking heuristics.

Gemini Types Trees:: The following is a simple type tree for our NIFS instrument (Near-Infrared Integral Field Spectrometer). It is an example of a minimalist type tree, which covers only the instrument and its general IMAGE and SPECT modes.

The text and detail in Figure 2 will be difficult to read, but I have included it to shows a more complete tree of types, in this case for GMOS, the Gemini Multi-Object Spectrometer.

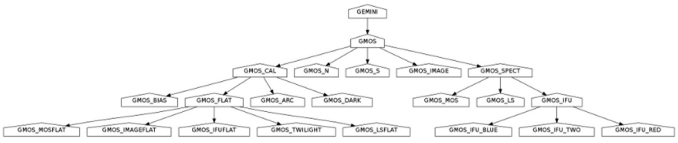


Fig. 2: GMOS Type Tree *This is a fully defined type tree, taken from the Gemini AstroData Type Library, the GMOS instrument tree,.*

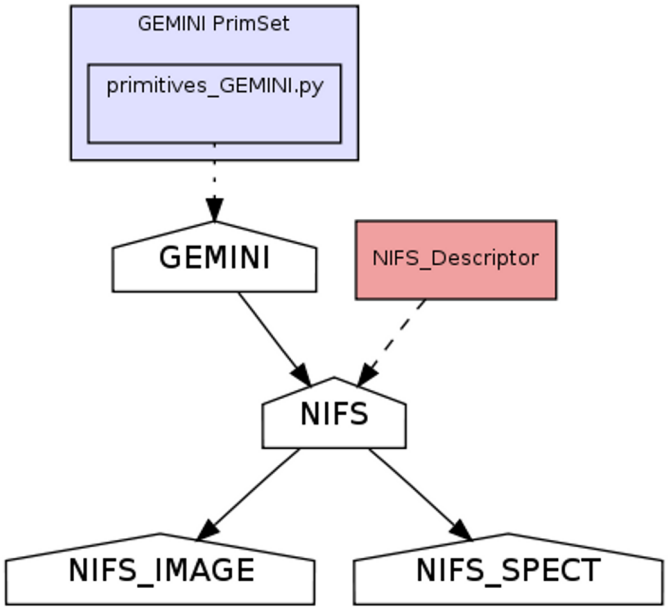


Fig. 3: NIFS Type Tree *The simple NIFS type tree showing which type has the Descriptor calculator assigned.*

AstroData Descriptors

AstroData *Descriptors* are terms naming high-level metadata which should be accessible for any dataset in the dataset family, either with generic or classification-specific calculators. The code implementing descriptors are functions bundled together in classes called Descriptor Calculators which are assigned to particular AstroData types.

This design allows a mix of generic and special-case descriptor implementations, using python's object oriented class definition to inherit generic implementations while overwriting descriptor functions that require special processing for that type.

For example, currently the NIFS descriptor calculator is a single calculator assigned to all NIFS data. This means this calculator has to handle both imaging and spectroscopic data. This can of course be done by placing type-specific code within conditionals and using AstroData to check classifications. Still the code can and will get convoluted if the different types rely on very different methods to return the information.

If a particular instrument-mode requires a special calculation, and if the developers do not want to complicate the more generic code, then another descriptor calculator descending from the NIFS general descriptor calculator class would be created, and the descriptor requiring special handling would be overriden, and this class would be assigned to the type which requires a special case, e.g. NIFS_SPECT.

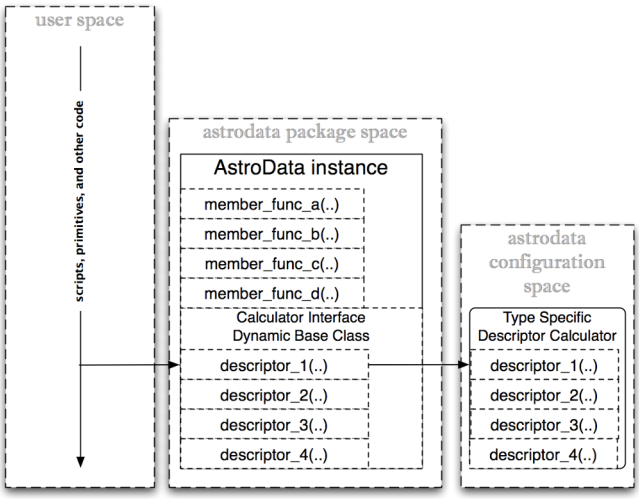


Fig. 4: Descriptor Calls*: The Descriptors are called as members of type-specific Descriptor Calculators through the type-agnostic Calculator Interface, which is melded into AstroData via metaprogramming techniques.*

Descriptor Calculator Classes

A descriptor function, associated and named with the official descriptor name, is implemented as a member function of a "Descriptor Calculator" (descending from the astrodata "Calculator" base-class). An instance of the correct calculator is stored in a private AstroData member, with there being just one correct calculator for any given AstroData instance. The classification hierarchy is used if multiple calculator assignments are found to apply to a dataset. Child nodes override parent nodes, siblings or cousin nodes with conflicting feature assignments will cause the system to complain and an exceptions to be thrown.

Interfaces to the descriptor functions are added as members at runtime to the AstroData instance using metaprogramming techniques. The configuration's "CalculatorInterface" class is used as an AstroData "base" class at runtime (this is called a mixin pattern). Currently the class is generated by a script, but in the future this class will be dynamically generated by the infrastructure from descriptor metadata.

After construction of the AstroData instance, descriptors such as `gain` and `filter_name` are available to call as member functions, e.g. continuing from the previous examples:

```
gain = ad.gain()
```

This line will call the correct `gain` implementation, having loaded the correct calculator for the dataset loaded into the "ad" variable. The calculator interface is constructed of "thunk" functions which proxy calls to the calculator and are called for all types of dataset. This makes the calculator interface a potential place to perform global features such as validation of descriptor inputs and values or processing globally supported parameters. These thunk functions call the appropriate descriptor in the calculator.

Examples

Correctly defined and assigned descriptors ensure high level metadata can be retrieved in the same way regardless of datatype, e.g. to retrieve the filter_name descriptor regardless of dataset type:

```
>>> from astrodata import AstroData
```

```
>>> ad = AstroData("trim_gN20091027S0133.fits")
>>> ad.filter_name()
'i_G0302'
```

Descriptors are presented as functions rather than data members to emphasize that they are indeed functions and to allow arguments which modify the return value, e.g. to get the filtername without the unique filter ID, filter_name accepts the "stripID" argument:

```
>>> ad.filter_name(stripID=True)
'i'
```

Some descriptors apply at the header-data unit level and only work on AstroData instances with a single extension. For example, a GMOS image prior to being mosaic-ed, will have three science extensions, one for each CCD in the GMOS instruments, and each of these has its own associated gain relating to the amp it was read out with. A descriptor will in this case have to return a collection if asked to return gain for the whole dataset. By default Descriptors only return single values of a specific type, so gain must return a double. In general this is not an issue, since it's more common in such a case to be iterating over AstroData-wrapped header-data units, in which case one naturally gets single-HDU AstroData instances:

```
>>> for ext in ad["SCI"]:
...     print ext.gain()
...
2.1
2.337
2.3
```

Similarly single extension AstroData instances can be picked out of the dataset by their naming information, if present, or by the integer index:

```
>>> gain_sci_1 = ad[("SCI", 1)].gain()
>>> gain_sci_1
2.1000000000000001
>>> gain_1 = ad[0].gain()
>>> gain_1
2.1000000000000001
```

To override the default descriptor return type to return collections when called on a multiple-extension dataset, affected descriptors support "asList" and "asDict" arguments:

```
>>> gainlist = ad.gain(asList=True)
>>> gainlist
[2.1000000000000001, 2.3370000000000002, 2.2999999999999998]
```

Lists are returned in order of the extensions for which there is a gain value (e.g. "SCI" extensions), and dictionaries returned are keyed by the extension naming information if present, or integer location in the list otherwise.

Primitives

Primitives are the third type of term defined in the astrodata grammar. Primitives name transformations, and conceptually recieve a list of input data and produce a list of output data. More technically primitives recieve a *ReductionContext*, and this is what they transform. Thus, strictly speaking they do not have to transform datasets, and even may not transform the ReductionContext (i.e. they may perform "the identity transformation").

The motivation for such primitives is to execute useful code during a reduction, for example primitives that print information to the log don't modify the reduction context at all, much less the data in the data stream. Also, some primitives can make queries about which files to process, and put these filenames in the datastream. This type of primitive will not have modified any datasets, but will have modified the reduction context which contains all information about an ongoing primitive-based reduction.

Nevertheless, most primitives exist to reduce data, so we still think of primitives as transforming data, and the fact that they actually transform reduction contexts is a technical detail only sometimes important. As with descriptors different implementations share a common name. This is so type-specific implementations can be executed in a regular way at higher scopes where the differences are not significant so long as incidental differences in the dataset types are accomodated.

Unlike descriptors, primitives are not added as AstroData members but are instead arranged into "recipes", which are simple sequential lists of primitives. As mere lists of steps, *recipes* contain no explicit conditionals. However, since each primitive executed is guaranteed to load an implimentation appropriate for the input dataset, recipes have an implicit type-based conditionality, or "type adaptativity".

Take for example our "prepare" recipe. The "prepare" transformation is meant to take raw data from any instrument and produce a somewhat normalized dataset, e.g. with standard namings, order, some validation performed, and standard headers set correctly.

The prepare recipe:

```
validateData(repair=True)
standardizeStructure
standardizeHeaders
validateWCS
```

All Gemini data needs to be "prepared", and this recipe descibes the procedure for them all. When executing this recipe, a list of files are fed into the first primitive. This primitive does whatever work on the inputs it is designed to do, and places its outputs in the reduction context, where they are used as input for the next step.

At each step the system checks the AstroData type of the inputs for the about-to-be-executed step to ensure the correct primitive implementation for that type is loaded and will be executed.

Some of the primitives in "prepare" are general purpose primitives, shared by all Gemini datasets and assigned to the general purpose GEMINI Primitive Set. For example `standardizeHeaders` is a fairly generic operation applying to all Gemini data. The meager type-sensitive differences are easilly handled in a single all purpose primitive.

On the other hand, the `standardizeStructure` primitive will not be the same for all types of dataset, nor even for all the modes within an instrument. For example, in the case of SPECT types (spectroscopy), `standardizeStructure` will add the appropriate Mask Definition File from our mask definition database, while the implementation of the same primitive for IMAGE types will not do this, since that table-HDU does not apply to imaging.

Final Thoughts

Current and Future Activities

We are currently deploying the astrodata package internally at Gemini for development and preliminary dataflow operations. We have a medium term project to use astrodata's primitive transformation and automation features (aka "the Recipe System") for Night Time Operations, but this is not in place at the moment.

The astrodata infrastructure code is largely stable. Though there is ongoing work, most work finishing the package is going into the astrodata_Gemini configuration package. Descriptors for all instruments already exist, and we are creating primitives for the GMOS instrument's imaging mode. We are making primitives needed by GMOS-imaging as general as possible, and will hopefully benefit from some momentum as we work through primitive sets for other instruments and modes. The type library of any given instrument will be filled in detail during creation of primitives for the given types. At the moment there are at least one type for each instrument, and one for their IMAGE and SPECT modes as applicable.

A fourth foundational term in the astrodata grammar exists in prototype form and will be developed in the near future, called "AstroData Structures", used for validation and also projecting hierarchical structure onto the dataset.

As we develop AstroData and deploy it for Gemini-specific purposes, we are interested in working with others in the future to extend the system's infrastructure and to support more types of data with configuration packages designed to handle other telescope's data. Anyone interested should contact Craig Allen, callen@gemini.edu, at Gemini Observatory, Data Processing Software Group.

Speaking About Data

Creating a language about our data in order to inform the astrodata software how our data should be organized has already helped us to be more efficient and apt in our own communication about dataflow, in our design and on our work to finished parts of the system under development. We can apply terms directly, because they map one to one with features AstroData can provide.

A large part of the advantage that has emerged from designing the terms and details within the configuration in the way described is that it focuses us on concepts first, seperately from implementation. Recipes, turn out to be good conceptual lynchpins for human discussion on the type of reduction the recipe performs. Software engineering details are compartmentalized to other discussions about how to provide a well defined transformation in the case of a particular AstroData Type.

Recipes *support* defining common steps separately from steps that tend to requiring specialization, but moreover they promote the practice since the system rewards proper granularity decisions with more effective type adaptation. The result is that we are concieving of better ways to describe transformations and what we are transforming.

Prior to discussing recipes as such, the high concept, four or five step description of a particlar reduction was hidden somewhat opaquely in the machinations of the reduction script itself. Such a script will tend to have the high level concepts obscured by low level software plumbing. The ability to describe reductions in terms of reasonably short recipes allows us to focus on this descriptive level, and yet to know that the recipe discussed in principle *is actually what is executed*.

Our configurations are becoming not merely where the astrodata software system is told how to support a given instrument-mode. They are instead becoming the official location of such knowledge, because the configurations are largely human readable, and insofar as otherwise this knowledge is not recorded clearly in a centralized way, but lives in the minds and distributed web pages of Gemini instrument scientists and data analysts.

By inspiring us to think in terms of the abstract concepts behind our data, we create and benefit from a language about Gemini data. This in turn is affecting how we think about our data. In the future, when we have incorporated the current state of affairs into our AstroData configuration package, I suspect it will greatly inform how we incorporate new instruments into the Gemini data family, and to match their new, powerful, observations modes, with the new powerful data reduction features needed to support them.

Terms

astrodata

- *astrodata*, uncapitalized, is the astrodata package, i.e. `import astrodata` or "when importing `astrodata` the Classification Library will be discovered and loaded".
- *AstroData*, with "CamelCase" names the AstroData class, i.e.e `ad = AstroData("f.fits")` or "When loading a MEF into AstroData, the type information is always loaded and available after instantiation".
- *Astrodata*, with an initial capital names the package in a general way, such as in a title or description, e.g. "The Astrodata Package can be imported using the name, 'astrodata'".

Note, it's a subtle distinction, and probably best to rely primarily on context to know which sense was intended.

HDU

from pyfits, "Header Data Unit"

HDUList

from pyfits, list-like structure returned from pyfits.open(..), and used internally by AstroData as the open file handle.

pyfits

A library for loading MEF files in python, using numpy for data sections. see STScI, http://www.stsci.edu/resources/software_hardware/pyfits

Divisi: Learning from Semantic Networks and Sparse SVD

Rob Speer[‡*], Kenneth Arnold[§], Catherine Havasi[§]

Abstract—Singular value decomposition (SVD) is a powerful technique for finding similarities and patterns in large data sets. SVD has applications in text analysis, bioinformatics, and recommender systems, and in particular was used in many of the top entries to the Netflix Challenge. It can also help generalize and learn from knowledge represented in a sparse semantic network.

Although this operation is fundamental to many fields, it requires a significant investment of effort to compute an SVD from sparse data using Python tools. Divisi is an answer to this: it combines NumPy, PySparse, and an extension module wrapping SVDLIBC, to make Lanczos' algorithm for sparse SVD easily usable within cross-platform Python code.

Divisi includes utilities for working with data in a variety of sparse formats, including semantic networks represented as edge lists or NetworkX graphs. It augments its matrices with labels, allowing you to keep track of the meaning of your data as it passes through the SVD, and it can export the labeled data in a format suitable for separate visualization GUIs.

Index Terms—SVD, sparse, linear algebra, semantic networks, graph theory

Introduction

Singular value decomposition (SVD) is a way of factoring an arbitrary rectangular matrix, in order to express the data in terms of its *principal components*. SVD can be used to reduce the dimensionality of a large matrix, a key step in many domains, including recommender systems, text mining, search, statistics, and signal processing.

The *truncated SVD*, in which only the largest principal components are calculated, is a particularly useful operation in many fields because it can represent large amounts of data using relatively small matrices. In many applications, the input to the truncated SVD takes the form of a very large, sparse matrix, most of whose entries are zero or unknown.

Divisi provides the Lanczos algorithm [Lan98] for performing a sparse, truncated SVD, as well as useful tools for constructing the input matrix and working with the results, in a reusable Python package called `divisi2`. It also includes important operations for preparing data such as normalization and mean-centering. More experimentally, Divisi also includes implementations of some SVD-inspired algorithms such as CCIPCA [Wen03] and landmark multi-dimensional scaling [Sil04].

* Corresponding author: rspeer@mit.edu
‡ MIT Media Lab
§ MIT

Using singular value decomposition, any matrix A can be factored into an orthonormal matrix U, a diagonal matrix Σ, and an orthonormal matrix V^T, so that $A = U\Sigma V^T$. The singular values in Σ can be ordered from largest to smallest, where the larger values correspond to the vectors in U and V that are more significant components of the initial A matrix. The largest singular values, and their corresponding rows of U and columns of V, represent the principal components of the data.

To create the truncated SVD, discard all but the first k components—the principal components of A—resulting in the smaller matrices U_k, Σ_k, and V_k^T. The components that are discarded represent relatively small variations in the data, and the principal components form a low-rank approximation of the original data. One can then reconstruct a smoothed version of the input matrix as an approximation: $A \approx U_k\Sigma_k V_k^T = A_k$.

To make it easier to work with SVD in understandable Python code, Divisi provides an abstraction over sparse and dense matrices that allows their rows and columns to be augmented with meaningful labels, which persist through various matrix operations.

The documentation for installing and using Divisi is hosted at http://csc.media.mit.edu/docs/divisi2/.

Architecture

Divisi is built on a number of other software packages. It uses NumPy [Oli10] to represent dense matrices, and PySparse [Geu08] to represent sparse ones, and uses a Cython wrapper around SVDLIBC [Roh10] to perform the sparse SVD. It can optionally use NetworkX [Net10] to take input from a directed graph such as a semantic network.

Divisi works with data in the form of *labeled arrays*. These arrays can be sparse or dense, and they can be 1-D vectors or 2-D matrices.

Figure 1 shows the relationships between classes in Divisi2. The yellow-highlighted classes are the ones that are intended to be instantiated. The core representations use multiple inheritance: for example, the properties of a SparseMatrix are separately defined by the fact that it is sparse and the fact that it is a 2-D matrix.

Sparse arrays encapsulate a PysparseMatrix from the `pysparse` package, while dense arrays are a subclass of `numpy.ndarray` and therefore support most NumPy operations. Both representations support NumPy-style "fancy indexing".

A vector contains a single, optional list of labels: if it exists, each entry in the list corresponds to one entry in the vector. A matrix may have two lists of labels: one assigns a label to each

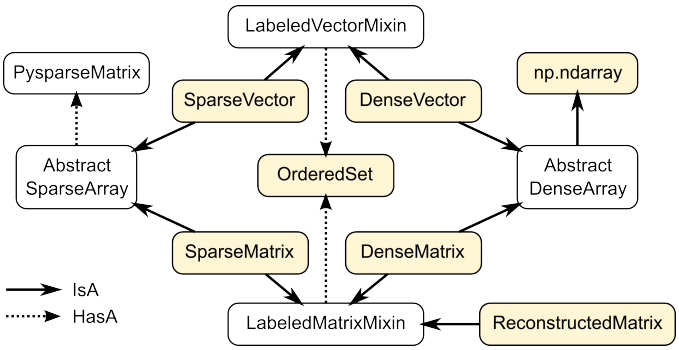


Fig. 1: *Relationships between the main classes in Divisi 2.0, as well as some externally-defined classes.*

row, and another assigns a label to each column. The purpose of these labels is to associate a meaning with each entry in a vector or matrix, so that code can look up entries by what they *mean* instead of simply by their position in the matrix.

The label lists themselves are instances of Divisi's OrderedSet class, which augments a list with a dictionary of its values, so that it can perform the `.index()` operation—finding an entry by its value—in constant time. This enables methods such as `SparseMatrix.row_named(label)`, efficiently returning the row with a particular label.

One more important class is ReconstructedMatrix, which lazily evaluates the matrix product of two DenseMatrices. This allows using the SVD as an approximation to a large matrix, but stores that large approximate matrix as a product of its SVD factors instead, which requires much less memory.

Next, we will explain the features of Divisi through three examples: performing latent semantic analysis (LSA) over documents from the Brown corpus, making movie recommendations from a MovieLens data set, and drawing conclusions based on ConceptNet (a semantic network of general knowledge).

Latent semantic analysis

One common use for Divisi is to make a topic model from a number of documents using latent semantic analysis (LSA). LSA typically consists of constructing a bag-of-words matrix of the words that appear in various documents, normalizing that matrix using tf-idf, and taking the SVD.

We'll use as an example a collection of 44 documents from the "news" section of the Brown corpus, a sample of which is available through NLTK:

```
>>> import nltk
>>> nltk.download('brown')
>>> len(nltk.corpus.brown.fileids(['news']))
44
```

When searching for articles about, say, books, we don't generally care whether the document contained "book" or "books". NLTK includes the Porter stemmer, which strips off endings:

```
>>> stemmer = nltk.PorterStemmer()
>>> stemmer.stem('books')
'book'
```

In the most basic form of LSA, each document is treated as a "bag of words", ignoring all sequence and punctuation. The following function yields all the stemmed words from a document in the Brown corpus:

```
>>> import re; word_re = re.compile(r'[A-Za-z]')
>>> categories=['news']
>>> fileids = nltk.corpus.brown.fileids(categories)
>>> def normalized_words(fileid):
...     for word in nltk.corpus.brown.words(fileid):
...         if word_re.match(word):
...             yield stemmer.stem(word.lower())
```

Now that we have the input data, we can load it into a Divisi sparse matrix. The function `divisi2.make_sparse`[1] creates a sparse matrix from a list of entries, each of which is a tuple of `(value, row, col)`:

```
>>> from csc import divisi2
>>> entries = ((1, term, doc)
...            for doc in fileids
...            for term in normalized_words(doc))
>>> matrix = divisi2.make_sparse(entries)
>>> print matrix
SparseMatrix (8976 by 44)
         ca01       ca02       ca03       ca04     ...
the      1.55e+02   1.34e+02   1.50e+02   1.60e+02
fulton  14.000000      ---        ---        ---
counti  17.000000   8.000000   2.000000      ---
grand    4.000000      ---     3.000000      ---
juri    19.000000      ---     5.000000      ---
said    24.000000  14.000000  17.000000   3.000000
...
```

A Divisi sparse matrix behaves like a NumPy array, but has additional facilities for labeling entries. Notice that `row` and `col` were both specified as strings (a term and a filename) rather than numbers. The `row_labels` and `col_labels` attributes keep track of what label is assigned to each row or column index:[2]:

```
>>> matrix.row_labels
<OrderedSet of 8976 items like the>
>>> matrix.col_labels
<OrderedSet of 44 items like ca01>
>>> matrix[0,0]
155.0
>>> matrix.entry_named('the', 'ca01')
155.0
```

That entry indicates that the word "the" appeared 155 times in the first document alone. Such common words would overwhelm the analysis: we should give less weight to words that appear in nearly every document. Also, a document that is twice as long as average should not necessarily be twice as influential. The standard solution to these problems is called *tf-idf normalization* and is one of several normalization capabilities provided by Divisi:

```
>>> normalized = matrix.normalize_tfidf().squish()
```

All Divisi normalization routines return a copy of their input. The final `.squish()` call deals with words like "the": since they appear in every document, their idf value, and thus the value of every entry in the corresponding row, is 0. Rows and columns that are all zeros leave part of the SVD result unconstrained, so we remove them for numerical stability.

Next we can compute the SVD. The only parameter is the number of singular values ("components") to keep. The optimal value depends on the corpus and task at hand; it essentially controls how much you want to fill in gaps in your data. Since the corpus is small, we arbitrarily choose 10 for this example.

```
>>> u, sigma, v = normalized.svd(k=10)
```

1. The version of Divisi described in this paper, Divisi 2.0, would be installed in a namespace package called `csc`. Divisi 2.2 can now be imported directly as `divisi2`, but references to `csc.divisi2` still work.

2. Example output in this paper is truncated or rounded for brevity.

Here, `sigma` is an array of diagonal entries; the actual diagonal matrix Σ is given by `np.diag(sigma)`.

Since $A \approx U\Sigma V^T$, we can execute various queries simply by matrix multiplication. For example, which documents are likely to contain terms like "book"? That's just a row of A. Using the approximation, we can compute that row:

```
>>> from pprint import pprint
>>> booky = divisi2.dot(u.row_named('book'),
                        divisi2.dot(np.diag(sigma), v.T))
>>> pprint(booky.top_items(3))
[('ca44', 0.0079525209393728428),
 ('ca31', 0.0017088410316380212),
 ('ca18', 0.0010004880691358573)]
```

`divisi2.dot` is a wrapper around `numpy.dot` that ensures that labels are maintained properly.

Reconstructing an approximate matrix

Divisi provides simpler ways of working with matrix reconstructions: the `ReconstructedMatrix` class:

```
>>> reconstructed = divisi2.reconstruct(u, sigma, v)
>>> booky2 = reconstructed.row_named('book')
>>> assert np.allclose(booky, booky2)
```

Another common query, often seen in blog posts, is which articles are similar to the one in question. Mathematically, which other document has the term vector with the highest dot product with the term vector of this document? The answer is again found in a matrix slice, this time of

$$A^T A = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T.$$

Again, Divisi provides functionality for easily slicing similarity matrices:

```
>>> similar_docs = \
...   divisi2.reconstruct_similarity(v, sigma)\
...     .row_named('ca44')
>>> pprint(similar_docs.top_items(3))
[('ca44', 0.99999999999999978),
 ('ca31', 0.82249752503164653),
 ('ca33', 0.6026564223332086)]
```

By default, `reconstruct_similarity` normalizes the result values to lie between -1 and 1.

Making recommendations

In the above example, we assumed that unspecified entries in the input matrix were zero, representing a lack of knowledge. When using SVD over a data set whose numeric values do not meaningfully start at zero, some adjustments are necessary.

In the domain of movie recommendations, for example, the input data often takes the form of star ratings that people assign to movies, ranging from 1 to 5. A 5-star rating and a 1-star rating are as different as can be, so a 5-star rating certainly does not have the meaning of "a 1-star rating, but five times more so".

In fact, the scale of ratings differs among people and movies. A movie rater may be very stingy with high ratings, so if they give a movie five stars it is very meaningful. Likewise, a movie could be widely panned, receiving a 1.1 star rating on average, so when someone gives the movie five stars it says that there is something very different about their taste in movies.

The movie rating problem can be broken down into two steps [Kor09]: accounting for the *biases* in ratings inherent to each movie and each person, and learning how people's particular preferences differ from those biases. We can represent the second step as an SVD where zero *does* represent a lack of information, and add the biases back in when we reconstruct the matrix.

To begin the example, load the MovieLens dataset of 100,000 movie ratings [Kon98], which is provided free from http://grouplens.org:

```
>>> from csc import divisi2
>>> from csc.divisi2.dataset import movielens_ratings
>>> movie_data = divisi2.make_sparse(
      movielens_ratings('data/movielens/u')).squish(5)
```

The "squish" method at the end discards users and movies with fewer than 5 ratings.

With this data, for example, we can query for the movies with the highest row bias (and therefore the highest average rating):

```
>>> import numpy as np
>>> movie_goodness = movie_data.row_op(np.mean)
>>> movie_goodness.top_items(5)
[('Pather Panchali (1955)', 4.625),
 ('Close Shave, A (1995)', 4.4910714285714288),
 ("Schindler's List (1993)", 4.4664429530201346),
 ('Wrong Trousers, The (1993)', 4.4661016949152543),
 ('Casablanca (1942)', 4.4567901234567904)]
```

We use the `SparseMatrix.mean_center()` method to remove the biases, leaving only the differences from the mean, calculate a 20-dimensional truncated SVD from those differences, and reconstruct an approximate matrix that predicts people's movie ratings.

```
>>> movie_data2, row_shift, col_shift, total_shift =\
...   movie_data.mean_center()
>>> recommendations = divisi2.reconstruct(
...   U, S, V,
...   shifts=(row_shift, col_shift, total_shift))
```

Let's look in particular at user number 5, who rated 174 movies. We can get a vector of their recommendations and query for the best ones:

```
>>> recs_for_5 = recommendations.col_named(5)
>>> recs_for_5.top_items(5)
[('Star Wars (1977)', 4.816),
 ('Return of the Jedi (1983)', 4.549),
 ('Wrong Trousers, The (1993)', 4.529),
 ('Close Shave, A (1995)', 4.416),
 ('Empire Strikes Back, The (1980)', 4.392)]
```

We see that this user should really like the Star Wars Trilogy, but this is unsurprising because the user in fact already told MovieLens they liked those movies. To get true recommendations, we should make sure to filter for movies they have not yet rated.

```
>>> recs_for_5 = recommendations.col_named(5)
>>> unrated = list(set(xrange(movie_data.shape[0]))
...   - set(recs_for_5.nonzero_indices()))
>>> rec[unrated].top_items(5)
[('Wallace & Gromit: [...] (1996)', 4.197),
 ('Terminator, The (1984)', 4.103),
 ('Casablanca (1942)', 4.044),
 ('Pather Panchali (1955)', 4.004),
 ('Dr. Strangelove [...] (1963)', 3.998)]
```

And on the other end of the scale, if we look for the best anti-recommendation in `(-rec[unrated])`, we find that user 5 should give "3 Ninjas: High Noon At Mega Mountain" a rating of 0.24 stars.

SVD alone does not make a cutting-edge, high-quality recommender system, but it does a reasonable part of the job. This process has been used as a component of many recommender systems, including the Netflix Prize-winning system, Bellkor's Pragmatic Chaos [Kor09], and Divisi makes it easy to do in Python.

Learning from a semantic network

Divisi contains methods for learning from data in a semantic network in NetworkX format. The network can contain labeled nodes and labeled edges with weights on each edge, and can build matrices that relate these to each other in a variety of ways.

This is an important feature of Divisi, because it extends its scope to data that is not traditionally represented as a matrix. It can learn from and generalize patterns that appear in any semantic network, and it is especially effective if that network contains redundancies or incomplete information. For this reason, we often use it to learn from ConceptNet [Hav07], a network of people's general "common sense" knowledge about the real world. A graph representation of ConceptNet 4.0 is included with Divisi 2.0.

The `divisi2.network` module defines the various ways to extract information from these labeled semantic networks. Its `sparse_triples()` function turns the list of edges into a list of (value, rowlabel, columnlabel) triples that can be used to build a sparse matrix, and uses the arguments `row_labeler` and `col_labeler` to specify how the values are assigned to labels. `sparse_matrix()` goes the extra step to turn these triples into a matrix.

In many cases, the labeler will give two results for each edge, because each edge connects two nodes. When the row and column labelers both give two results, they will be paired up in contrary order. The next example will clarify why this is useful.

One simple labeler is `'nodes'`, which extracts the source and target nodes of each edge. If an edge of weight 1 connects "dog" to "bark", then because of the contrary order rule, `sparse_matrix(graph, 'nodes', 'nodes')` will put a 1 in the entry whose row is "dog" and column is "bark", as well as the entry whose row is "bark" and whose column is "dog". The resulting overall matrix is the adjacency matrix of the graph.

`'features'` is a more complex labeler: it takes the edge label into account as well, and describes an incoming or outgoing edge, including the node on the other side of it. The idea is that a node can be combined with a feature to completely describe an edge.

For example, consider a weight-1 edge from "dog" to "mammal", labeled with "IsA", expressing the assertion that "a dog is a mammal". The matrix `sparse_matrix(graph, 'nodes', 'features')` will then express both the fact that the node "dog" has the feature "IsA mammal", and that "mammal" has the feature "dog IsA".

These features are represented with Divisi as 3-tuples of (*direction*, *edge label*, *node label*), where *direction* is "left" or "right" depending on whether this is an incoming or outgoing edge.

Other possible labelers are "relations", which extracts just the edge label, and "pairs", extracting the source and target nodes as tuples, and more can be defined as functions.

The process called AnalogySpace [Spe08] involves making a node vs. feature matrix of common sense knowledge and generalizing it with a truncated SVD. We will show an example of doing this with ConceptNet here.

Learning from ConceptNet

Start by loading the pre-defined ConceptNet 4.0 graph:

```
>>> conceptnet_graph = divisi2.load(
      'data:graphs/conceptnet_en.graph')
```

We can break this graph down into nodes and features, and see a sample of what it looks like:

```
>>> from csc.divisi2.network import sparse_matrix
>>> A = sparse_matrix(graph, 'nodes', 'features',
                      cutoff=3)
>>> print A
SparseMatrix (12564 by 19719)
         IsA/spor   IsA/game   UsedFor/   UsedFor/
baseball 3.609584   2.043731   0.792481   0.500000
sport       ---     1.292481      ---     1.000000
yo-yo       ---        ---        ---        ---
toy         ---     0.500000      ---     1.160964
dog         ---        ---        ---     0.792481
...
```

And with that, we can make a truncated SVD and reconstruct an approximation to A:

```
>>> U, S, V = A.svd(k=100)
>>> Ak = divisi2.reconstruct(U, S, V)
>>> Ak.entry_named('pig', ('right', 'HasA', 'leg'))
0.15071150848740383
>>> Ak.entry_named('pig',
                   ('right', 'CapableOf', 'fly'))
-0.26456066802309008
```

As shown in the earlier LSA example, we can also reconstruct an approximation to the similarity matrix $A^T A$, describing how similar the nodes are to each other:

```
>>> sim = divisi2.reconstruct_similarity(U, S)
>>> sim.entry_named('horse', 'cow')
0.827
>>> sim.entry_named('horse', 'stapler')
-0.031
>>> sim.row_named('table').top_items()
[('table', 1.000), ('newspaper article', 0.694),
 ('dine table', 0.681), ('dine room table', 0.676),
 ('table chair', 0.669), ('dine room', 0.663),
 ('bookshelve', 0.636), ('table set', 0.629),
 ('home depot', 0.591), ('wipe mouth', 0.587)]
```

Recall that `reconstruct_similarity` normalizes its values to between -1 and 1. Here, this normalization makes some nodes, such as "newspaper article" and "home depot", get a spuriously high weight because their truncated SVD vectors had low magnitude. When ranking possible similarities—or, for that matter, predictions for new assertions that could be true—we have found it more useful to normalize the vectors to unit vectors *before* the SVD, so that nodes that are weakly described by the SVD do not end up magnified.

Divisi allows for pre-SVD normalization with the SparseMatrix methods `normalize_rows()`, `normalize_cols()`, and `normalize_all()`. (tf-idf normalization, like in the LSA example, is also an option, but it is inappropriate here because it de-emphasizes common concepts.) The first two scale the rows or columns, respectively, of the input so that they become unit vectors. However, normalizing the rows can further distort the magnitudes of the columns, and vice versa, and there is no way to exactly normalize both the rows and columns of an arbitrary matrix.

We have found that a compromise works best: normalize each entry by the geometric mean of its row and column magnitudes. This is what `SparseMatrix.normalize_all()` does, and we favor it in this case because not only does it put all the rows and columns on approximately the same scale, it also increases the predictive accuracy of the reconstructed SVD (which we will be able to quantify in a moment).

In this representation, we can look again at the similarities for "table":

```
>>> U, S, V = A.normalize_all().svd(k=100)
>>> sim = divisi2.reconstruct_similarity(U, S)
>>> sim.row_named('table').top_items()
[('table', 1.718), ('desk', 1.195),
 ('kitchen', 0.988), ('chair', 0.873),
 ('restaurant', 0.850), ('plate', 0.822),
 ('bed', 0.772), ('cabinet', 0.678),
 ('refrigerator', 0.652), ('cupboard', 0.617)]
```

Choosing parameters

So far, we have used two parameters in this process without justification: the method of normalization, and the value of k.

Instead of simply tweaking these parameters by hand, we can bring in some test data and search for the parameters that maximize the predictive value of the SVD. Because what we care about is the relative ranking of statements, not the numerical values they are assigned, a traditional mean-squared evaluation does not exactly make sense.

However, using Divisi, we can evaluate how often the relative ranking of a pair of assertions agrees with the ranking that a human would give them. In the case of ConceptNet, we have already acquired many such human-evaluated statements from evaluations such as the one in [Spe08], so we use those as the source of gold-standard rankings.

The `ReconstructedMatrix.evaluate_ranking()` method is what we use to compare pairwise rankings in this way. We can use it, first of all, to confirm that `normalize_all()` performs better than the other possible normalization methods on ConceptNet, leaving k fixed at 100. The results are:

- Without normalization: 68.47% agreement
- Using `normalize_rows`: 67.66% agreement
- Using `normalize_cols`: 67.30% agreement
- Using `normalize_all`: 70.77% agreement

Then, after applying that normalization method, we can try truncated SVDs with various values of k.

```
>>> from csc.divisi2.network import conceptnet_matrix
>>> conceptnet = conceptnet_matrix('en').normalize_all()
>>> testdata = divisi2.load('usertest_data.pickle')
>>> accuracy_data = []
>>> for k in xrange(1, 200):
...     U, S, V = conceptnet.svd(k=k)
...     rec = divisi2.reconstruct(U, S, V)
...     correct, total, accuracy =\
...         rec.evaluate_ranking(testdata)
...     accuracy_data.append(accuracy)
```

The plot of the resulting `accuracy_data` in Figure 2 shows a plateau of good values of k, roughly between $k = 100$ and $k = 200$.

Memory use and scalability

The main use case of Divisi2 is to decompose a sparse matrix whose entries fit in memory. The objects that primarily consume memory are:

- The linked lists that comprise the PySparse matrix
- The compressed-sparse-column copy of this matrix used by SVDLIBC
- The dense matrices U and V, and the vector S, that are returned by SVDLIBC and used directly by NumPy
- The optional OrderedSets of labels (each using a Python list and dictionary)

Each nonzero entry in a sparse matrix and each entry in a dense matrix requires the space of a C double (assumed to be 8 bytes).

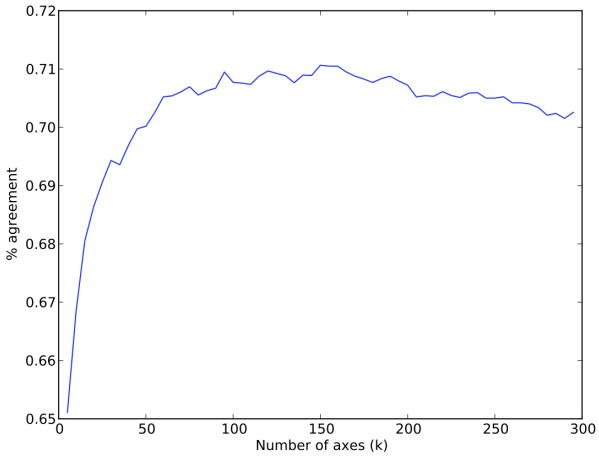


Fig. 2: *Evaluating the predictive accuracy of the truncated SVD on ConceptNet for various values of* k.

The PySparse matrix also requires an integer (4 bytes), acting as a pointer, for each entry. (This implementation incidentally limits matrices to having fewer than 2^{31} nonzero entries.) The non-zero entries in the compressed-sparse-column matrix also come with integer row numbers. Finally, each allocated row requires two integer pointers.

So, without labels, a rank k decomposition of an $m \times n$ matrix with z non-zero entries requires $(24z+8m+8k(m+n))$ bytes, plus a negligible amount of overhead from Python and C structures. As a practical example, it is possible within the 4 GiB memory limit of 32-bit CPython to take a rank-100 decomposition of a $10^6 \times 10^6$ matrix with 10^8 entries, or a rank-10 decomposition of a $10^7 \times 10^7$ matrix with 10^8 entries, each of which requires 3.7 to 3.8 GiB plus overhead.

In order to support even larger, denser data sets, Divisi 2.2 includes an experimental implementation of Hebbian incremental SVD that does not require storing the sparse data in memory.

Conclusion

The SVD is a versatile analysis tool for many different kinds of data. Divisi provides an easy way to compute the SVD of large sparse datasets in Python, and additionally provides Pythonic wrappers for performing common types of queries on the result.

Divisi also includes a variety of other functionality. For example, it can analyze combinations of multiple matrices of data, a technique called *blending*, which is useful for drawing conclusions from multiple data sources simultaneously.

Further documentation about Divisi2, including the presentation from SciPy 2010, is available at http://csc.media.mit.edu/docs/divisi2/.

References

[Kor09] Y. Koren, R. Bell, and C. Volinsky. *Matrix Factorization Techniques for Recommender Systems*. Computer, 42(8):30-37, August 2009.

[Kon98] J. Konstan, J. Riedl, A. Borchers, and J. Herlocke. *Recommender Systems: A GroupLens Perspective*. Papers from the 1998 Workshop on Recommender Systems, Chapel Hill, NC. 1998.

[Net10] NetworkX Developers. *NetworkX*. Viewable online at:http://networkx.lanl.gov/, 2010.

[Roh10] Doug Rohde. *SVDLIBC*. Viewable online at: http://tedlab.mit.edu/~dr/SVDLIBC/, 2010.

[Geu08] Roman Geus, Daniel Wheeler, and Dominique Orban. *PySparse*. Viewable online at: http://pysparse.sourceforge.net/, 2008.

[Oli10] Travis Oliphant. *Guide to Numpy*. Viewable online at: http://www.tramy.us/, 2010.

[Sil04] Vin de Silva and Joshua B. Tenenbaum. *Sparse multidimensional scaling using landmark points*. Stanford University Technical Report, 2004.

[Wen03] Juyang Weng and Yilu Zhang and Wey-Shiuan Hwang. *Candid covariance-free incremental principal component analysis*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(8):1034-1040, August 2003.

[Lan98] Cornelius Lanczos and William R. Davis (ed). *Collected published papers with commentaries*. North Carolina State University, 1998.

[Hav07] Catherine Havasi, Robert Speer, and Jason Alonso. *ConceptNet 3: a Flexible, Multilingual Semantic Network for Common Sense Knowledge*. Recent Advances in Natural Language Processing, September 2007.

[Spe08] Robert Speer and Catherine Havasi and Henry Lieberman. *AnalogySpace: Reducing the Dimensionality of Common Sense Knowledge*. Proceedings of AAAI 2008, July 2008.

Theano: A CPU and GPU Math Compiler in Python

James Bergstra[‡*], Olivier Breuleux[‡], Frédéric Bastien[‡], Pascal Lamblin[‡], Razvan Pascanu[‡], Guillaume Desjardins[‡], Joseph Turian[‡], David Warde-Farley[‡], Yoshua Bengio[‡]

Abstract—Theano is a compiler for mathematical expressions in Python that combines the convenience of NumPy's syntax with the speed of optimized native machine language. The user composes mathematical expressions in a high-level description that mimics NumPy's syntax and semantics, while being statically typed and functional (as opposed to imperative). These expressions allow Theano to provide symbolic differentiation. Before performing computation, Theano optimizes the choice of expressions, translates them into C++ (or CUDA for GPU), compiles them into dynamically loaded Python modules, all automatically. Common machine learning algorithms implemented with Theano are from 1.6× to 7.5× faster than competitive alternatives (including those implemented with C/C++, NumPy/SciPy and MATLAB) when compiled for the CPU and between 6.5× and 44× faster when compiled for the GPU. This paper illustrates how to use Theano, outlines the scope of the compiler, provides benchmarks on both CPU and GPU processors, and explains its overall design.

Index Terms—GPU, CUDA, machine learning, optimization, compiler, NumPy

Introduction

Python is a powerful and flexible language for describing large-scale mathematical calculations, but the Python interpreter is in many cases a poor engine for executing them. One reason is that Python uses full-fledged Python objects on the heap to represent simple numeric scalars. To reduce the overhead in numeric calculations, it is important to use array types such as NumPy's `ndarray` so that single Python objects on the heap can stand for multidimensional arrays of numeric scalars, each stored efficiently in the host processor's native format.

[NumPy] provides an N-dimensional array data type, and many functions for indexing, reshaping, and performing elementary computations (`exp`, `log`, `sin`, etc.) on entire arrays at once. These functions are implemented in C for use within Python programs. However, the composition of many such NumPy functions can be unnecessarily slow when each call is dominated by the cost of transferring memory rather than the cost of performing calculations [Alted]. [numexpr] goes one step further by providing a loop fusion optimization that can glue several element-wise computations together. Unfortunately, numexpr requires an unusual syntax (the expression must be encoded as a string within the code), and at the time of this writing, numexpr is limited to optimizing element-wise computations. [Cython] and [scipy.weave] address Python's performance issue by offering a simple way to hand-write crucial segments of code in C (or a dialect of Python which can be easily compiled to C, in Cython's case). While this approach can yield significant speed gains, it is labor-intensive: if the bottleneck of a program is a large mathematical expression comprising hundreds of elementary operations, manual program optimization can be time-consuming and error-prone, making an automated approach to performance optimization highly desirable.

Theano, on the other hand, works on a symbolic representation of mathematical expressions, provided by the user in a NumPy-like syntax. Access to the full computational graph of an expression opens the door to advanced features such as symbolic differentiation of complex expressions, but more importantly allows Theano to perform local graph transformations that can correct many unnecessary, slow or numerically unstable expression patterns. Once optimized, the same graph can be used to generate CPU as well as GPU implementations (the latter using CUDA) without requiring changes to user code.

Theano is similar to [SymPy], in that both libraries manipulate symbolic mathematical graphs, but the two projects have a distinctly different focus. While SymPy implements a richer set of mathematical operations of the kind expected in a modern computer algebra system, Theano focuses on fast, efficient evaluation of primarily array-valued expressions.

Theano is free open source software, licensed under the New (3-clause) BSD license. It depends upon NumPy, and can optionally use SciPy. Theano includes many custom C and CUDA code generators which are able to specialize for particular types, sizes, and shapes of inputs; leveraging these code generators requires gcc (CPU) and nvcc (GPU) compilers, respectively. Theano can be extended with custom graph expressions, which can leverage `scipy.weave`, PyCUDA, Cython, and other numerical libraries and compilation technologies at the user's discretion. Theano has been actively and continuously developed and used since January 2008. It has been used in the preparation of numerous scientific papers and as a teaching platform for machine learning in graduate courses at l'Université de Montréal. Documentation and installation instructions can be found on Theano's website [theano]. All Theano users should subscribe to the announce[1] mailing list (low traffic). There are medium traffic mailing lists for developer discussion[2] and user support[3].

This paper is divided as follows: Case Study: Logistic Regression shows how Theano can be used to solve a simple problem in statistical prediction. Benchmarking Results presents some results of performance benchmarking on problems related to machine learning and expression evaluation. What kinds of work does Theano support? gives an overview of the design of Theano and the sorts of computations to which it is suited. Compilation by

* *Corresponding author: james.bergstra@umontreal.ca*
‡ *Université de Montréal*

theano.function provides a brief introduction to the compilation pipeline. Limitations and Future Work outlines current limitations of our implementation and currently planned additions to Theano.

Case Study: Logistic Regression

To get a sense of how Theano feels from a user's perspective, we will look at how to solve a binary logistic regression problem. Binary logistic regression is a classification model parameterized by a weight matrix W and bias vector b. The model estimates the probability $P(Y=1|x)$ (which we will denote with shorthand p) that the input x belongs to class $y=1$ as:

$$P(Y=1|x^{(i)}) = p^{(i)} = \frac{e^{Wx^{(i)}+b}}{1+e^{Wx^{(i)}+b}} \tag{1}$$

The goal is to optimize the log probability of N training examples, $\mathscr{D} = \{(x^{(i)}, y^{(i)}), 0 < i \leq N\}$, with respect to W and b. To maximize the log likelihood we will instead minimize the (average) negative log likelihood[4]:

$$\ell(W,b) = -\frac{1}{N}\sum_i y^{(i)} \log p^{(i)} + (1-y^{(i)}) \log(1-p^{(i)}) \tag{2}$$

To make it a bit more interesting, we can also include an ℓ_2 penalty on W, giving a cost function $E(W,b)$ defined as:

$$E(W,b) = \ell(W,b) + 0.01 \sum_i \sum_j w_{ij}^2 \tag{3}$$

In this example, tuning parameters W and b will be done through stochastic gradient descent (SGD) on $E(W,b)$. Stochastic gradient descent is a method for minimizing a differentiable loss function which is the expectation of some per-example loss over a set of training examples. SGD estimates this expectation with an average over one or several examples and performs a step in the approximate direction of steepest descent. Though more sophisticated algorithms for numerical optimization exist, in particular for smooth convex functions such as $E(W,b)$, stochastic gradient descent remains the method of choice when the number of training examples is too large to fit in memory, or in the setting where training examples arrive in a continuous stream. Even with relatively manageable dataset sizes, SGD can be particularly advantageous for non-convex loss functions (such as those explored in Benchmarking Results), where the stochasticity can allow the optimizer to escape shallow local minima [Bottou].

According to the SGD algorithm, the update on W is

$$W \leftarrow W - \mu \frac{1}{N'} \sum_i \left. \frac{\partial E(W,b,x,y)}{\partial W} \right|_{x=x^{(i)}, y=y^{(i)}}, \tag{4}$$

where $\mu = 0.1$ is the step size and N is the number of examples with which we will approximate the gradient (i.e. the number of rows of `x`). The update on `b` is likewise

$$b \leftarrow b - \mu \frac{1}{N'} \sum_i \left. \frac{\partial E(W,b,x,y)}{\partial b} \right|_{x=x^{(i)}, y=y^{(i)}}. \tag{5}$$

1. http://groups.google.com/group/theano-announce
2. http://groups.google.com/group/theano-dev
3. http://groups.google.com/group/theano-users
4. Taking the mean in this fashion decouples the choice of the regularization coefficient and the stochastic gradient step size from the number of training examples.

Implementing this minimization procedure in Theano involves the following four conceptual steps: (1) declaring symbolic variables, (2) using these variables to build a symbolic expression graph, (3) compiling Theano functions, and (4) calling said functions to perform numerical computations. The code listings in Figures 1 - 4 illustrate these steps with a working program that fits a logistic regression model to random data.

```
 1: import numpy
 2: import theano.tensor as T
 3: from theano import shared, function
 4:
 5: x = T.matrix()
 6: y = T.lvector()
 7: w = shared(numpy.random.randn(100))
 8: b = shared(numpy.zeros(()))
 9: print "Initial model:"
10: print w.get_value(), b.get_value()
```

Fig. 1: *Logistic regression, part 1: declaring variables.*

The code in Figure 1 declares four symbolic variables `x`, `y` `w`, and `b` to represent the data and parameters of the model. Each tensor variable is strictly typed to include its data type, its number of dimensions, and the dimensions along which it may broadcast (like NumPy's broadcasting) in element-wise expressions. The variable `x` is a matrix of the default data type (`float64`), and `y` is a vector of type `long` (or `int64`). Each row of `x` will store an example $x^{(i)}$, and each element of `y` will store the corresponding label $y^{(i)}$. The number of examples to use at once represents a tradeoff between computational and statistical efficiency.

The `shared()` function creates *shared variables* for W and b and assigns them initial values. Shared variables behave much like other Theano variables, with the exception that they also have a persistent value. A shared variable's value is maintained throughout the execution of the program and can be accessed with `.get_value()` and `.set_value()`, as shown in line 10.

```
11: p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
12: xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
13: cost = xent.mean() + 0.01*(w**2).sum()
14: gw,gb = T.grad(cost, [w,b])
15: prediction = p_1 > 0.5
```

Fig. 2: *Logistic regression, part 2: the computation graph.*

The code in Figure 2 specifies the computational graph required to perform stochastic gradient descent on the parameters of our cost function. Since Theano's interface shares much in common with that of NumPy, lines 11-15 should be self-explanatory for anyone familiar with that module. On line 11, we start by defining $P(Y=1|x^{(i)})$ as the symbolic variable `p_1`. Notice that the matrix multiplication and element-wise exponential functions are simply called via the `T.dot` and `T.exp` functions, analogous to `numpy.dot` and `numpy.exp`. `xent` defines the cross-entropy loss function, which is then combined with the ℓ_2 penalty on line 13, to form the cost function of Eq (3) and denoted by `cost`.

Line 14 is crucial to our implementation of SGD, as it performs symbolic differentiation of the scalar-valued `cost` variable with respect to variables `w` and `b`. `T.grad` operates by iterating backwards over the expression graph, applying the chain rule of differentiation and building symbolic expressions for the gradients on `w` and `b`. As such, `gw` and `gb` are also symbolic Theano variables, representing $\partial E/\partial W$ and $\partial E/\partial b$ respectively. Finally, line

15 defines the actual prediction (`prediction`) of the logistic regression by thresholding $P(Y = 1|x^{(i)})$.

```
16: predict = function(inputs=[x],
17:                    outputs=prediction)
18: train = function(
19:             inputs=[x,y],
20:             outputs=[prediction, xent],
21:             updates={w:w-0.1*gw, b:b-0.1*gb})
```

Fig. 3: *Logistic regression, part 3: compilation.*

The code of Figure 3 creates the two functions required to train and test our logistic regression model. Theano functions are callable objects that compute zero or more *outputs* from values given for one or more symbolic *inputs*. For example, the `predict` function computes and returns the value of `prediction` for a given value of `x`. Parameters `w` and `b` are passed implicitly - all shared variables are available as inputs to all functions as a convenience to the user.

Line 18 (Figure 3) which creates the `train` function highlights two other important features of Theano functions: the potential for multiple outputs and updates. In our example, `train` computes both the prediction (`prediction`) of the classifier as well as the cross-entropy error function (`xent`). Computing both outputs together is computationally efficient since it allows for the reuse of intermediate computations, such as `dot(x,w)`. The optional `updates` parameter enables functions to have side-effects on shared variables. The updates argument is a dictionary which specifies how shared variables should be updated after all other computation for the function takes place, just before the function returns. In our example, calling the `train` function will update the parameters `w` and `b` with new values as per the SGD algorithm.

```
22: N = 4
23: feats = 100
24: D = (numpy.random.randn(N, feats),
25:      numpy.random.randint(size=N,low=0, high=2))
26: training_steps = 10
27: for i in range(training_steps):
28:     pred, err = train(D[0], D[1])
29: print "Final model:",
30: print w.get_value(), b.get_value()
31: print "target values for D", D[1]
32: print "prediction on D", predict(D[0])
```

Fig. 4: *Logistic regression, part 4: computation.*

Our example concludes (Figure 4) by using the functions `train` and `predict` to fit the logistic regression model. Our data `D` in this example is just four random vectors and labels. Repeatedly calling the `train` function (lines 27-28) fits our parameters to the data. Note that calling a Theano function is no different than calling a standard Python function: the graph transformations, optimizations, compilation and calling of efficient C-functions (whether targeted for the CPU or GPU) have all been done under the hood. The arguments and return values of these functions are NumPy `ndarray` objects that interoperate normally with other scientific Python libraries and tools.

Benchmarking Results

Theano was developed to simplify the implementation of complex high-performance machine learning algorithms. This section presents performance in two processor-intensive tasks from that domain: training a multi-layer perceptron (MLP) and training a convolutional network. We chose these architectures because of their popularity in the machine learning community and their different computational demands. Large matrix-matrix multiplications dominate in the MLP example and two-dimensional image convolutions with small kernels are the major bottleneck in a convolutional network. More information about these models and their associated learning algorithms is available from the Deep Learning Tutorials [DLT]. The implementations used in these benchmarks are available online [dlb].

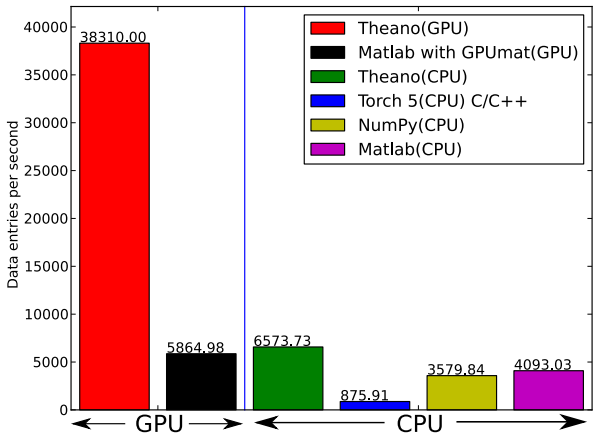


Fig. 5: *Fitting a multi-layer perceptron to simulated data with various implementations of stochastic gradient descent. These models have 784 inputs, 500 hidden units, a 10-way classification, and are trained 60 examples at a time.*

CPU timing was carried out on an an Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz with 2 GB of RAM. All implementations were linked against the BLAS implemented in the Intel Math Kernel Library, version 10.2.4.032 and allowed to use only one thread. GPU timing was done on a GeForce GTX 285. CPU computations were done at double-precision, whereas GPU computations were done at single-precision.

Our first benchmark involves training a single layer MLP by stochastic gradient descent. Each implementation repeatedly carried out the following steps: (1) multiply 60 784-element input vectors by a 784×500 weight matrix, (2) apply an element-wise hyperbolic tangent operator (tanh) to the result, (3) multiply the result of the tanh operation by a 500×10 matrix, (4) classify the result using a multi-class generalization of logistic regression, (5) compute the gradient by performing similar calculations but in reverse, and finally (6) add the gradients to the parameters. This program stresses element-wise computations and the use of BLAS routines.

Figure 5 compares the number of examples processed per second across different implementations. We compared Theano (revision #ec057beb6c) against NumPy 1.4.1, MATLAB 7.9.0.529, and Torch 5 (a machine learning library written in C/C++) [torch5] on the CPU and GPUMat 0.25 for MATLAB ([gpumat]) on the GPU.

When running on the CPU, Theano is 1.8x faster than NumPy, 1.6x faster than MATLAB, and 7.5x faster than Torch 5.[5] Theano's speed increases 5.8x on the GPU from the CPU, a total increase of 11x over NumPy (CPU) and 44x over Torch 5 (CPU). GPUmat

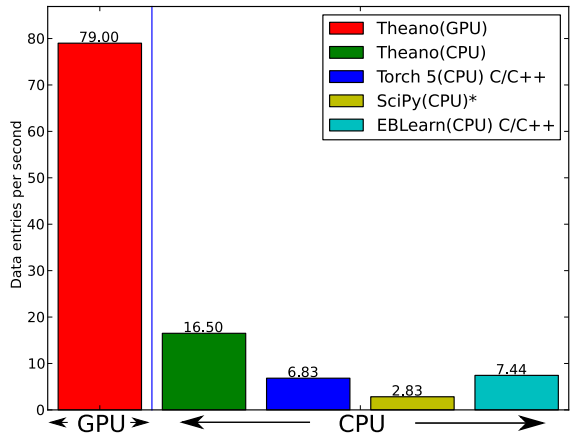


Fig. 6: *Fitting a convolutional network using different software. The benchmark stresses convolutions of medium-sized (256 by 256) images with small (7 by 7) filters.*

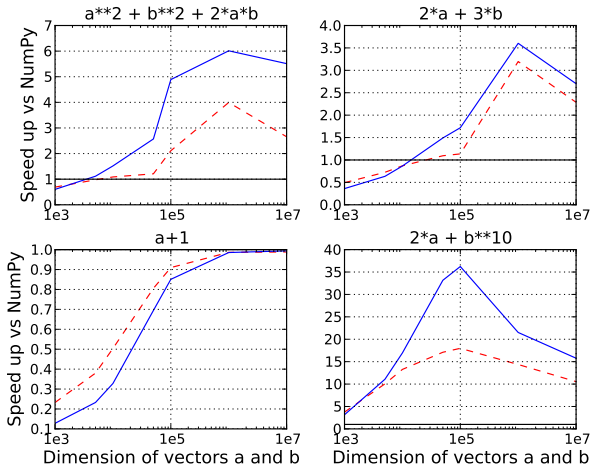


Fig. 7: *Speed comparison between NumPy, numexpr, and Theano for different sizes of input on four element-wise formulae. In each subplot, the solid blue line represents Theano, the dashed red line represent numexpr, and performance is plotted with respect to NumPy.*

brings about a speed increase of only 1.4x when switching to the GPU for the MATLAB implementation, far less than the 5.8x increase Theano achieves through CUDA specializations.

Because of the difficulty in implementing efficient convolutional networks, we only benchmark against known libraries that offer a pre-existing implementation. We compare against EBLearn [EBL] and Torch, two libraries written in C++. EBLearn was implemented by Yann LeCun's lab at NYU, who have done extensive research in convolutional networks. To put these results into perspective, we implemented approximately half (no gradient calculation) of the algorithm using SciPy's `signal.convolve2d` function. This benchmark uses convolutions of medium sized images (256×256) with small filters (7×7). Figure 6 compares the performance of Theano (both CPU and GPU) with that of competing implementations. On the CPU, Theano is 2.2x faster than EBLearn, its best competitor. This advantage is owed to the fact that Theano compiles more specialized convolution routines. Theano's speed increases 4.9x on the GPU from the CPU, a total of 10.7x over EBLearn (CPU). On the CPU, Theano is 5.8x faster than SciPy even though SciPy is doing only half the computations. This is because SciPy's convolution routine has not been optimized for this application.

We also compared Theano with numexpr and NumPy for evaluating element-wise expressions on the CPU (Figure 7). For small amounts of data, the extra function-call overhead of numexpr and Theano makes them slower. For larger amounts of data, and for more complicated expressions, Theano is fastest because it uses an implementation specialized for each expression.

What kinds of work does Theano support?

Theano's expression types cover much of the same functionality as NumPy, and include some of what can be found in SciPy. Table 1 lists some of the most-used expressions in Theano. More extensive reference documentation is available online [theano].

Theano's strong suit is its support for strided N-dimensional arrays of integers and floating point values. Signed and unsigned integers of all native bit widths are supported, as are both single-precision and double-precision floats. Single-precision and double-precision complex numbers are also supported, but less so - for example, gradients through several mathematical functions are not implemented. Roughly 90% of expressions for single-precision N-dimensional arrays have GPU implementations. Our goal is to provide GPU implementations for all expressions supported by Theano.

Random numbers are provided in two ways: via NumPy's random module, and via an internal generator from the MRG family [Ecu]. Theano's `RandomStreams` replicates the `numpy.random.RandomState` interface, and acts as a proxy to NumPy's random number generator and the various random distributions that use it. The `MRG_RandomStreams` class implements a different random number generation algorithm (called MRG31k3p) that maps naturally to GPU architectures. It is implemented for both the CPU and GPU so that programs can produce the same results on either architecture without sacrificing speed. The `MRG_RandomStreams` class offers a more limited selection of random number distributions than NumPy though: uniform, normal, and multinomial.

Sparse vectors and matrices are supported via SciPy's `sparse` module. Only compressed-row and compressed-column formats are supported by most expressions. There are expressions for packing and unpacking these sparse types, some operator support (e.g. scaling, negation), matrix transposition, and matrix multiplication with both sparse and dense matrices. Sparse expressions currently have no GPU equivalents.

There is also support in Theano for arbitrary Python objects. However, there are very few expressions that make use of that support because the compilation pipeline works on the basis of inferring properties of intermediate results. If an intermediate result can be an arbitrary Python object, very little can be inferred. Still, it is occasionally useful to have such objects in Theano graphs.

Theano has been developed to support machine learning research, and that has motivated the inclusion of more specialized expression types such as the logistic sigmoid, the softmax func-

5. Torch was designed and implemented with flexibility in mind, not speed (Ronan Collobert, p.c.).

Operators	+, −, /, *, **, //, eq, neq, <, <=, >, >=, &, |, ^
Allocation	alloc, eye, [ones,zeros]_like, identity{_like}
Indexing*	basic slicing (see set_subtensor and inc_subtensor for slicing lvalues); limited support for advanced indexing
Mathematical Functions	exp, log, tan[h], cos[h], sin[h], real, imag, sqrt, floor, ceil, round, abs
Tensor Operations	all, any, mean, sum, min, max, var, prod, argmin, argmax, reshape, flatten, dimshuffle
Conditional	cond, switch
Looping	Scan
Linear Algebra	dot, outer, tensordot, diag, cholesky, inv, solve
Calculus*	grad
Signal Processing	conv2d, FFT, max_pool_2d
Random	RandomStreams, MRG_RandomStreams
Printing	Print
Sparse	compressed row/col storage, limited operator support, dot, transpose, conversion to/from dense
Machine Learning	sigmoid, softmax, multi-class hinge loss

TABLE 1: *Overview of Theano's core functionality. This list is not exhaustive, and is superseded by the online documentation. More details are given in text for items marked with an asterisk.* `dimshuffle` *is like* `numpy.swapaxes`.

tion, and multi-class hinge loss.

Compilation by theano.function

What happens under the hood when creating a function? This section outlines, in broad strokes, the stages of the compilation pipeline. Prior to these stages, the expression graph is copied so that the compilation process does not change anything in the graph built by the user. As illustrated in Figure 8, the expression graph is subjected to several transformations: (1) canonicalization, (2) stabilization, (3) specialization, (4) optional GPU transfer, (5) code generation. There is some overlap between these transformations, but at a high level they have different objectives. (The interested reader should note that these transformations correspond roughly, but not exactly to the optimization objects that are implemented in the project source code.)

Canonicalization

The canonicalization transformation puts the user's expression graph into a standard form. For example, duplicate expressions are merged into a single expression. Two expressions

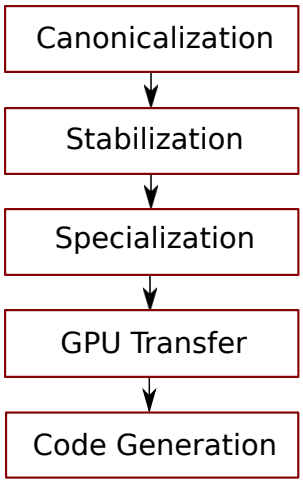


Fig. 8: *The compilation pipeline for functions compiled for GPU. Functions compiled for the CPU omit the GPU transfer step.*

are considered duplicates if they carry out the same operation and have the same inputs. Since Theano expressions are purely functional (i.e., cannot have side effects), these expressions must return the same value and thus it is safe to perform the operation once and reuse the result. The symbolic gradient mechanism often introduces redundancy, so this step is quite important. For another example, sub-expressions involving only multiplication and division are put into a standard fraction form (e.g. `a / (((a * b) / c) / d) -> (a * c * d) / (a * b) -> (c * d) / (b)`). Some useless calculations are eliminated in this phase, for instance cancelling out uses of the `a` term in the previous example, but also reducing `exp(log(x))` to `x`, and computing outright the values of any expression whose inputs are fully known at compile time. Canonicalization simplifies and optimizes the graph to some extent, but its primary function is to collapse many different expressions into a single normal form so that it is easier to recognize expression patterns in subsequent compilation stages.

Stabilization

The stabilization transformation improves the numerical stability of the computations implied by the expression graph. For instance, consider the function `log(1 + exp(x))`, which tends toward zero as $\lim_{x\to-\infty}$, and `x` as $\lim_{x\to-\infty}$. Due to limitations in the representation of double precision numbers, the computation as written yields infinity for `x > 709`. The stabilization phase replaces patterns like one with an expression that simply returns `x` when `x` is sufficiently large (using doubles, this is accurate beyond the least significant digit). It should be noted that this phase cannot guarantee the stability of computations. It helps in some cases, but the user is still advised to be wary of numerically problematic computations.

Specialization

The specialization transformation replaces expressions with faster ones. Expressions like `pow(x,2)` become `sqr(x)`. Theano also performs more elaborate specializations: for example, expressions involving scalar-multiplied matrix additions and multiplications may become BLAS General matrix multiply (GEMM) nodes and `reshape`, `transpose`, and `subtensor` expressions (which create copies by default) are replaced by constant-time versions

that work by aliasing memory. Expressions subgraphs involving element-wise operations are fused together (as in numexpr) in order to avoid the creation and use of unnecessary temporary variables. For instance, if we denote the `a + b` operation on tensors as `map(+, a, b)`, then an expression such as `map(+, map(*, a, b), c)` would become `map(lambda ai,bi,ci: ai*bi+ci, a, b, c)`. If the user desires to use the GPU, expressions with corresponding GPU implementations are substituted in, and transfer expressions are introduced where needed. Specialization also introduces expressions that treat inputs as workspace buffers. Such expressions use less memory and make better use of hierarchical memory, but they must be used with care because they effectively destroy intermediate results. Many expressions (e.g. GEMM and all element-wise ones) have such equivalents. Reusing memory this way allows more computation to take place on GPUs, where memory is at a premium.

Moving Computation to the GPU

Each expression in Theano is associated with an implementation that runs on either the host (a host expression) or a GPU device (a GPU expression). The GPU-transfer transformation replaces host expressions with GPU expressions. The majority of host expression types have GPU equivalents and the proportion is always growing.

The heuristic that guides GPU allocation is simple: if any input or output of an expression resides on the GPU and the expression has a GPU equivalent, then the GPU equivalent is substituted in. Shared variables storing `float32` tensors default to GPU storage, and the expressions derived from them consequently default to using GPU implementations. It is possible to explicitly force any `float32` variable to reside on the GPU, so one can start the chain reaction of optimizations and use the GPU even in graphs with no shared variables. It is possible (though awkward, and discouraged) to specify exactly which computations to perform on the GPU by disabling the default GPU optimizations.

Tensors stored on the GPU use a special internal data type with an interface similar to the `ndarray`. This datatype fully supports strided tensors, and arbitrary numbers of dimensions. The support for strides means that several operations such as the transpose and simple slice indexing can be performed in constant time.

Code Generation

The code generation phase of the compilation process produces and loads dynamically-compiled Python modules with specialized implementations for the expressions in the computation graph. Not all expressions have C (technically C++) implementations, but many (roughly 80%) of Theano's expressions generate and compile C or CUDA code during `theano.function`. The majority of expressions that generate C code specialize the code based on the dtype, broadcasting pattern, and number of dimensions of their arguments. A few expressions, such as the small-filter convolution (`conv2d`), further specialize code based on the size the arguments will have.

Why is it so important to specialize C code in this way? Modern x86 architectures are relatively forgiving of code that does not make good use techniques such as loop unrolling and prefetching contiguous blocks of memory, and only the `conv2d` expression goes to any great length to generate many special case implementations for the CPU. By comparison, GPU architectures are much less forgiving of code that is not carefully specialized for the size and physical layout of function arguments. Consequently, the code generators for GPU expressions like `GpuSum`, `GpuElementwise`, and `GpuConv2d` generate a wider variety of implementations than their respective host expressions. With the current generation of graphics cards, the difference in speed between a naïve implementation and an optimal implementation of an expression as simple as matrix row summation can be an order of magnitude or more. The fact that Theano's GPU `ndarray`-like type supports strided tensors makes it even more important for the GPU code generators to support a variety of memory layouts. These compile-time specialized CUDA kernels are integral to Theano's GPU performance.

Limitations and Future Work

While most of the development effort has been directed at making Theano produce fast code, not as much attention has been paid to the optimization of the compilation process itself. At present, the compilation time tends to grow super-linearly with the size of the expression graph. Theano can deal with graphs up to a few thousand nodes, with compilation times typically on the order of seconds. Beyond that, it can be impractically slow, unless some of the more expensive optimizations are disabled, or pieces of the graph are compiled separately.

A Theano function call also requires more overhead (on the order of microseconds) than a native Python function call. For this reason, Theano is suited to applications where functions correspond to expressions that are not too small (see Figure 7).

The set of types and operations that Theano provides continues to grow, but it does not cover all the functionality of NumPy and covers only a few features of SciPy. Wrapping functions from these and other libraries is often straightforward, but implementing their gradients or related graph transformations can be more difficult. Theano does not yet have expressions for sparse or dense matrix inversion, nor linear algebra decompositions, although work on these is underway outside of the Theano trunk. Support for complex numbers is also not as widely implemented or as well-tested as for integers and floating point numbers. NumPy arrays with non-numeric dtypes (strings, Unicode, Python objects) are not supported at present.

We expect to improve support for advanced indexing and linear algebra in the coming months. Documentation online describes how to add new operations and new graph transformations. There is currently an experimental GPU version of the scan operation, used for looping, and an experimental lazy-evaluation scheme for branching conditionals.

The library has been tuned towards expressions related to machine learning with neural networks, and it is not as well tested outside of this domain. Theano is not a powerful computer algebra system, and it is an important area of future work to improve its ability to recognize numerical instability in complicated element-wise expression graphs.

Debugging Theano functions can require non-standard techniques and Theano specific tools. The reason is two-fold: 1) definition of Theano expressions is separate from their execution, and 2) optimizations can introduce many changes to the computation graph. Theano thus provides separate execution modes for Theano functions, which allows for automated debugging and profiling. Debugging entails automated sanity checks, which ensure that all optimizations and graph transformations are safe (Theano compares the results before and after their application), as well as comparing the outputs of both C and Python implementations.

We plan to extend GPU support to the full range of C data types, but only float32 tensors are supported as of this writing. There is also no support for sparse vectors or matrices on the GPU, although algorithms from the CUSPARSE package should make it easy to add at least basic support for sparse GPU objects.

Conclusion

Theano is a mathematical expression compiler for Python that translates high level NumPy-like code into machine language for efficient CPU and GPU computation. Theano achieves good performance by minimizing the use of temporary variables, minimizing pressure on fast memory caches, making full use of `gemm` and `gemv` BLAS subroutines, and generating fast C code that is specialized to sizes and constants in the expression graph. Theano implementations of machine learning algorithms related to neural networks on one core of an E8500 CPU are up to 1.8 times faster than implementations in NumPy, 1.6 times faster than MATLAB, and 7.6 times faster than a related C++ library. Using a Nvidia GeForce GTX285 GPU, Theano is an additional 5.8 times faster. One of Theano's greatest strengths is its ability to generate custom-made CUDA kernels, which can not only significantly outperform CPU implementations but alternative GPU implementations as well.

Acknowledgements

Theano has benefited from the contributions of many members of the machine learning group in the computer science department (Département d'Informatique et de Recherche Operationelle) at l'Université de Montréal, especially Arnaud Bergeron, Thierry Bertin-Mahieux, Olivier Delalleau, Douglas Eck, Dumitru Erhan, Philippe Hamel, Simon Lemieux, Pierre-Antoine Manzagol, and François Savard. The authors acknowledge the support of the following agencies for research funding and computing support: NSERC, RQCHP, CIFAR, SHARCNET and CLUMEQ.

REFERENCES

[theano] Theano. http://www.deeplearning.net/software/theano

[NumPy] T. E. Oliphant. "Python for Scientific Computing". *Computing in Science & Engineering* 9, 10 (2007).

[Bottou] L. Bottou. "Online Algorithms and Stochastic Approximations". In D. Saad, ed. *Online Learning and Neural Networks* (1998). Cambridge University Press, Cambridge, UK. Online: http://leon.bottou.org/papers/bottou-98x

[numexpr] D. Cooke *et al*. "numexpr". http://code.google.com/p/numexpr/

[Cython] S. Behnel, R. Bradshaw, and D. S. Seljebotn. "Cython: C-Extensions for Python". http://www.cython.org/

[scipy.weave] SciPy Weave module. http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html

[Alted] F. Alted. "Why Modern CPUs Are Starving And What Can Be Done About It". *Computing in Science and Engineering* 12(2):68-71, 2010.

[SymPy] SymPy Development Team. "SymPy: Python Library for Symbolic Mathematics". http://www.sympy.org/

[BLAS] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. "Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms". *ACM Trans. Math. Soft.*, 16:18-28, 1990. http://www.netlib.org/blas

[LAPACK] E. Anderson *et al*. "LAPACK Users' Guide, Third Edition". http://www.netlib.org/lapack/lug/index.html

[DLT] Deep Learning Tutorials. http://deeplearning.net/tutorial/

[dlb] Benchmarking code: http://github.com/pascanur/DeepLearningBenchmarks

[torch5] R. Collobert. "Torch 5". http://torch5.sourceforge.net

[EBL] "EBLearn: Energy Based Learning, a C++ Machine Learning Library". http://eblearn.sourceforge.net/

[gpumat] "GPUmat: GPU toolbox for MATLAB". http://gp-you.org

[Ecu] P. L'Ecuyer, F. Blouin, and R. Couture. "A Search for Good Multiple Recursive Generators". *ACM Transactions on Modeling and Computer Simulation*, 3:87-98, 1993.

A High Performance Robot Vision Algorithm Implemented in Python

Steven C. Colbert[§*], Gregor Franz[‡], Konrad Woellhaf[‡], Redwan Alqasemi[§], Rajiv Dubey[§]

Abstract—A crucial behavior for assistive robots that operate in unstructured domestic settings is the ability to efficiently reconstruct the 3D geometry of novel objects at run time using no *a priori* knowledge of the object. This geometric information is critical for the robot to plan grasping and other manipulation maneuvers, and it would be impractical to employ database driven or other prior knowledge based schemes since the number and variety of objects that system may be tasked to manipulate are large.

We have developed a robot vision algorithm capable of reconstructing the 3D geometry of a novel object using only three images of the object captured from a monocular camera in an eye-in-hand configuration. The reconstructions are sufficiently accurate approximations such that the system can use the recovered model to plan grasping and manipulation maneuvers. The three images are captured from disparate locations and the object of interest segmented from the background and converted to a silhouette. The three silhouettes are used to approximate the surface of the object in the form of a point cloud. The accuracy of the approximation is then refined by regressing an 11 parameter superquadric to the cloud of points. The 11 parameters of the recovered superquadric then serve as the model of the object.

The entire system is implemented in Python and Python related projects. Image processing tasks are performed with NumPy arrays making use of Cython for performance critical tasks. Camera calibration and image segmentation utilize the Python bindings to the OpenCV library which are available in the scikits.image project. The non-linear constrained optimization uses the fmin_l_bfgs_b algorithm in scipy.optimize. The algorithm was first vetted in a simulation environment built on top of Enthought Traits and Mayavi.

The hardware implementation utilizes the Python OpenOPC project to communicate with and control a Kuka KR 6/2 six axis industrial manipulator. Images are captured via an Axis 207MW wireless network camera by issuing cgi requests to the camera with the urllib2 module. The image data is converted from JPEG to RGB raster format with the Python Imaging Library. The core algorithm runs as a server on a standalone machine and is accessed using the XML-RPC protocol. Not including the time required for the robot to capture the images, the entire reconstruction process is executed, on average, in 300 milliseconds.

Index Terms—computer vision, real-time, geometry, robotics

1. Introduction

Recently, the robotics and automation literature has seen an increase in research focus on the autonomous pose and shape estimation of general objects. The intent of these studies is that the pose and shape information of objects can be used to plan grasping and manipulation maneuvers. In this context, such object recognition abilities have a plethora of applications that span multiple domains including, but not limited to: industrial automation, assistive devices, and rehabilitation robotics. Up to this point, a large portion of the research has focused on recognizing objects in which the system has some form of *a priori* knowledge; usually a 3D model or set of images of the object taken from various angles along with a database of information describing the objects. Recent examples of work in this area can be found in eg. [Kim09], [Effendi08], [Schlemmer07], [Liefhebber07], [Kragic05].

We approach this problem with the goal that the system need not have any prior knowledge of the object it wishes to manipulate. In the context of assistive or service robotics, requiring such 3D models or a database of information for every possible object would be prohibitively tedious and time consuming, thus severely limiting its usefulness and applicability. In order to achieve this goal, we attempt to describe generic objects in a bulk fashion. That is, to autonomously model an object's actual physical form at run time with a simplified shape that is an approximation; one which is also sufficiently accurate to allow for the planning and execution of grasping maneuvers. In our previous works [Colbert10_1], [Colbert10_2], we describe in detail the development of an algorithm that accomplishes just this. Only a brief overview of that theoretical work is presented here. Rather, the majority of this paper focuses on the implementation of that algorithm on an industrial manipulator and the accuracy of the reconstruction that results.

The paper progresses as follows: Section 2 provides the high level overview of the algorithm with some diagrams and a step-by-step visual example to ease conceptual understanding, Section 3 describes the software implementation of the algorithm, Section 4 describes the robotic hardware implementation to include networking and control, Section 5 elaborates on the testing and overall accuracy of the platform and algorithm under real-world conditions. We round out the paper with some conclusions in Section 6.

2. Algorithm Overview

This section provides a high-level overview of the theory behind our object reconstruction algorithm. No equations are presented. Rather the algorithm is explained qualitatively and the interested reader is directed to one of our previous works that develop the theory in detail: [Colbert10_1], [Colbert10_2].

* Corresponding author: sccolber@mail.usf.edu
§ University of South Florida
‡ University of Applied Sciences Ravensburg-Weingarten

2.1 Shape from Silhouettes

The first phase or our algorithm generates a rough approximation to the surface of an object using a method that falls under the category of *shape from silhouettes*. Algorithms of this class use a number of silhouettes of an object of interest captured from various vantage points and, by back-projecting the visual cones and finding their union, reconstruct the geometry of the object. As the number of available silhouettes increases to infinity, the reconstruction converges on the *visual hull* of the object [Laurentini94]. That is, the reconstruction will converge to the true shape of the object, minus any concavities. The method by which the visual cones are back projected varies from algorithm to algorithm, but most have typically used a variation of voxel coloring or space carving [Dyer01]. Our method is a modified version of a recently introduced new method of shape from silhouettes [Lippiello09]. Our modification to this algorithm utilizes projective geometry to eliminate the iteration step required in the original algorithm. The result is a shape from silhouettes algorithm that is conceptually easier to understand and computationally more efficient than historical methods.

Our algorithm begins by capturing three images of the object of interest from three disparate locations, and segmenting the object from the background. The segmented object is then converted to a silhouette. Then, using these silhouettes along with the known camera parameters, the 3D centroid of the object of interest is estimated. Along with the centroid, we estimate a *radius* of the object, which we define as a distance from the estimated centroid that would define the radius of a sphere that would fully encompass the object. Once this centroid and radius are determined, a virtual sphere of points can be constructed which fully encompasses the object. For each of the points in the sphere, the point is projected into the silhouette image and tested for intersection. If the point intersects the silhouette, nothing is done. However, if the point does not intersect the silhouette, its position in 3-space is modified such that its projected location in the image lies on the boundary of the silhouette. When this process is repeated for each silhouette, the resulting set of points will approximate the surface of the object. The geometry can be described with the following procedure and associated graphic:

1) Let the center of the camera be $\mathbf{c}_0$.
2) Let the center of the sphere be $\mathbf{x}_0$.
3) Let $\mathbf{x}_i$ be any point in the sphere other than $\mathbf{x}_0$.
4) Let $\mathbf{x}_{i_{new}}$ be the updated position of point $\mathbf{x}_i$.
5) Let the projection of the center of the sphere into the image be $\mathbf{x}'_0$.
6) Then, for each point $\mathbf{x}_i$:
    a) Project $\mathbf{x}_i$ into the silhouette image to get $\mathbf{x}'_i$.
    b) If $\mathbf{x}'_i$ does not intersect the silhouette:
        i) Find the pixel point $\mathbf{p}'$ that lies on the edge of the silhouette along the line segment $\mathbf{x}'_i\mathbf{x}'_0$.
        ii) Reproject $\mathbf{p}'$ into $\mathbb{R}^3$ to get the point $\mathbf{p}$.
        iii) Let the line $\mathbf{c}_0\mathbf{p}$ be $\mathbf{L}_1$.
        iv) Let the line $\mathbf{x}_0\mathbf{x}_i$ be $\mathbf{L}_2$.
        v) Let $\mathbf{x}_{i_{new}}$ be the point of intersection of lines $\mathbf{L}_1$ and $\mathbf{L}_2$.
7) Repeat steps 2-6 for each silhouette image.

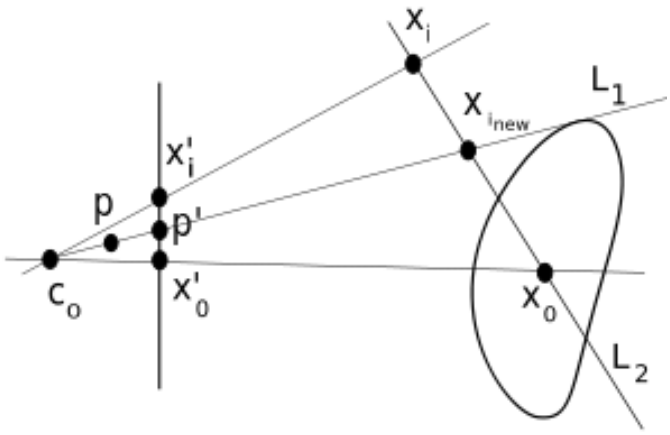


Fig. 1: *The geometry of point $\mathbf{x}_{i_{new}}$, which is the intersection of lines $\mathbf{L}_1$ and $\mathbf{L}_2$. The line $\mathbf{L}_2$ is defined by known points $\mathbf{x}_i$ and $\mathbf{x}_0$. The line $\mathbf{L}_1$ is defined by point $\mathbf{c}_0$, which is the camera center, and point $\mathbf{p}$, which is the reprojection of the image point $\mathbf{p}'$ into $\mathbb{R}^3$.*

2.2 Superquadrics

The resulting set of points will, in general, be only a rough approximation of the surface of the object of interest. As previously mentioned, as the number of captured images becomes large, this approximation will become ever more accurate, but at the expense of increasingly long computation times. Our aim is to achieve usable results with a minimum number of images. To achieve a more accurate representation of the object using just three images, we fit a superquadric to the set of points which approximate the surface in such a manner that the superquadric largely rejects disturbances due to perspective projection effects and localized noise. The fitted superquadric then serves as a parametrized description of the object which encodes its position, orientation, shape, and size.

Our fitting routine is based on the methods proposed in [Jaklic00], whose work on superquadrics is authoritative. We made a modification to their cost function which heavily penalizes points lying inside the boundaries of the superquadric. This modification has the effect of forcing the fitting routine to ignore disturbances caused by perspective projection effects. For a few number of images, these disturbances can be large, and thus this modification is crucial to achieving a satisfactory reconstruction with only three images.

The reconstruction of a simulated shape is shown in the following figure. From the figure, it is clear that the fitted superquadric provides a substantially better approximation to the original shape than what can be achieved from the point cloud alone, when only three images of the object are available.

3. Software Implementation

The algorithm was developed and implemented entirely in Python. Images take the form of NumPy arrays with FOR loop dependent geometric image calculations performed in Cython. The Cython bindings to the OpenCV library (available in the scikits.image project) were used to build up the image segmentation routine. The fmin_l_bfgs_b non-linear constrained optimization routine (available in SciPy) was adopted for purposes of finding the best fitting superquadric for the point cloud. The gradient of the

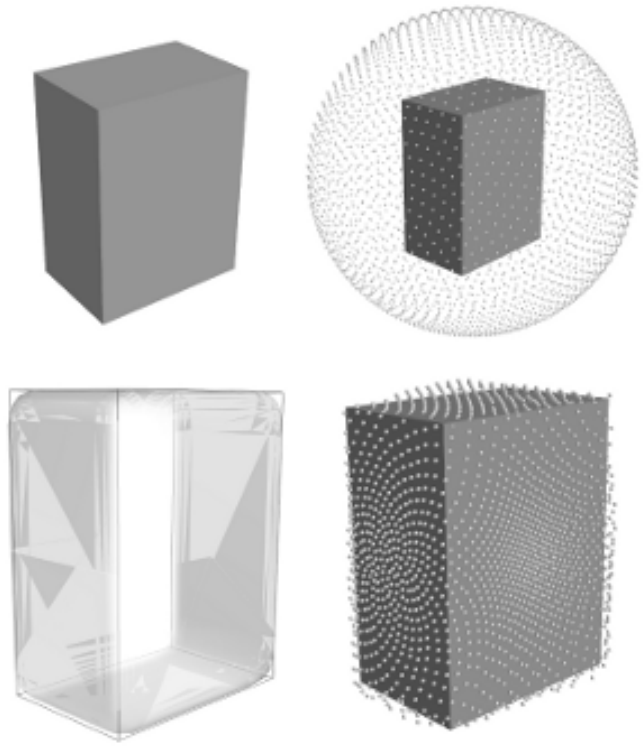


Fig. 2: *A simulated reconstruction. Clockwise from upper left: (1) The original shape. (2) The generated sphere of points. (3) The point cloud after the points have been shrunk to the silhouette boundaries. Error due to perspective projection is clearly seen. (4) The superquadric that was fit to the point cloud. Original shape shown as a wire frame. Notice the ability of the superquadric to ignore the perspective projection error.*

superquadric function (a hefty 296 SLOC) was implemented in Cython.

This software stack has proven to be quite performant. The average reconstruction time takes approximately 300 milliseconds. This includes image segmentation times but obviously does not include the time to actually capture the images. Compare this to the time taken for the reconstruction in [Yamazaki08] where a reconstruction using over 100 images required ~100 seconds of processing time for an equivalent accuracy.

A simulation environment was also developed in concert with the algorithm for testing purposes. The environment uses Mayavi as a rendering engine and TraitsUI for the GUI. The environment allows simulating a number of various shapes and modifying their parameters in real-time. It also allows the images of the object to be captured from any position. Once the images are captured, the simulator then performs the reconstruction and displays the recovered superquadric as an overlay on the current shape. The computed accuracy of the reconstruction, based on the recovered superquadric parameters versus the known ground truth, is shown in a sidebar. Various intermediate stages of the reconstruction process are also stored as hidden layers for debugging purposes. These layers can be turned on after the reconstruction via dialog options. All of the reconstruction images in this text were generated with either the simulator or the underlying Mayavi engine. A screenshot of the simulator is shown below.

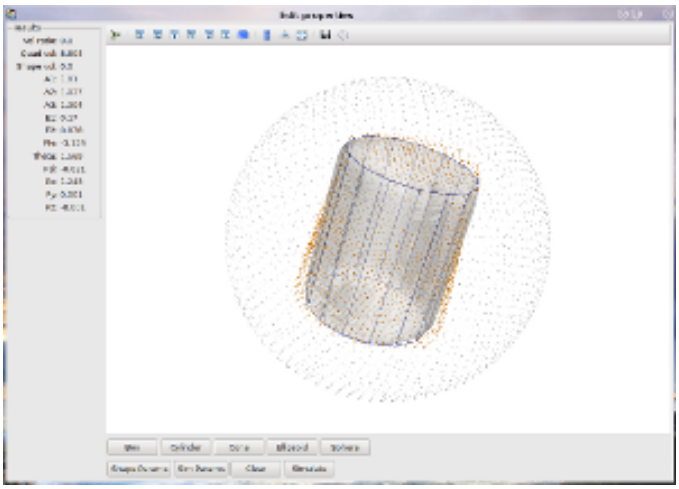


Fig. 3: *A screenshot of the simulator which is built on Mayavi and TraitsUI.*

4. Hardware Implementation

The implementation hardware consists of three main entities: the robotic manipulator which performs the required motions, the camera to capture the images, and the network which consists of the various components responsible for controlling the robot, the camera, and performing the actual object reconstruction computations.

It is desired to have these various systems interconnected in the most decoupled and hardware/operating system agnostic manner in order to facilitate software reuse on and with other platforms, robots, and cameras. Thus, portability was a chief goal behind the system design. The following sections describe each subsystem component in detail.

4.1 Robot

The robotic arm used for testing is a KUKA KR6/2, manufactured by KUKA Roboter GmbH. It is a six axis, low payload, industrial manipulator with high accuracy and a repeatability of <0.1mm. It's smaller size (though still too large for use on a mobile platform) and large workspace makes it well suited for laboratory use and a wide range of experiments. The robot setup, including the camera described in Section 4.2 is shown in the following figure.

The KUKA control software provides a proprietary user interface environment developed in Windows XP Embedded, which in turn runs atop the real time VxWorks operating system. The platform provides a programming interface to the robot utilizing the proprietary KUKA Robot Language (KRL) as well as an OPC server that allows for connections from outside computers and the reading and writing of OLE system variables. As KRL does not provide facilities for communicating with outside processes or computers, the OPC server connection was used in conjunction with a simple KRL program to export control to an outside machine. The details of this are delayed until Section 4.3.

4.2 Camera

The camera used for image acquisition is an Axis 207MW wireless network camera. It is relatively inexpensive and has megapixel resolution. The main beneficial feature of the camera is that it contains a built in HTTP web server with support for acquiring images via CGI requests. This means that the camera can be used

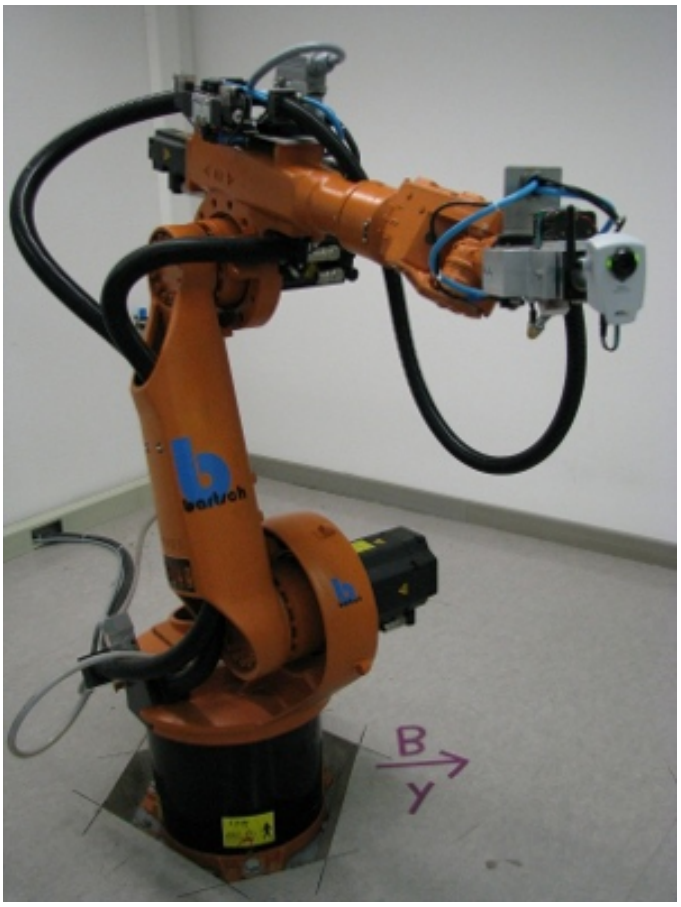


Fig. 4: *The robot platform with the camera mounted in the gripper.*

by any programming language with libraries supporting HTTP connections. Needless to say, the list of qualifying languages is extensive.

In order to transform the camera into a completely wireless component, a wireless power supply was developed. Namely, a custom voltage regulator was designed and fabricated to regulate the voltage of a battery pack down to the required 5V for the camera. The regulator will operate with any DC voltage from 7 - 25V, allowing interoperation with a wide variety of battery packs.

4.3 Network

In order to achieve our goal of portability, the network was designed around distributed components that use free and open source standards for interprocess communication. Each component in the network is capable of operating independently on its own machine from anywhere that has access to the central switch. In the case of our experiments, the central switch is a local 802.11 router providing WLAN access to the local computers in the laboratory. In our network setup, there are four components that share information across the LAN:

1) The KUKA robot computer running KRL programs and the OPC server
2) The Axis 207MW wireless network camera
3) The object reconstruction software
4) The external KUKA control software

The logical arrangement of these components, their interconnection, and the communication protocols used are illustrated

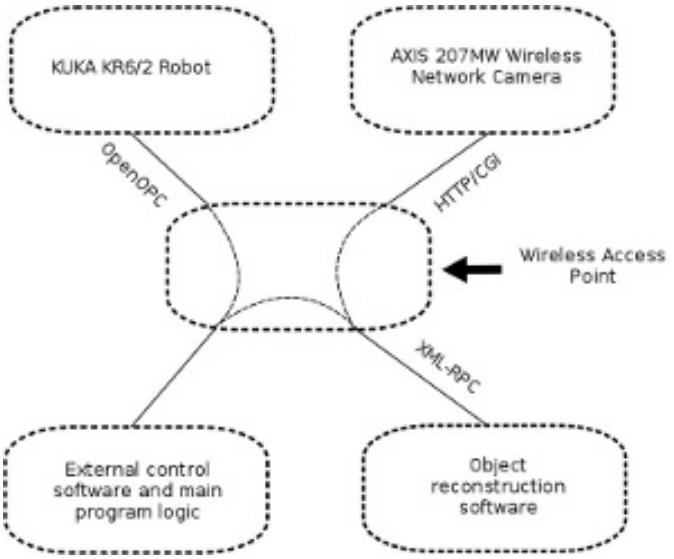


Fig. 5: *Network and communication layout.*

in following figure and are explained in detail in the following sections.

4.3.1 External KUKA Controller and the OPC Server: As previously mentioned, the KUKA robot software provides an OPC server that can be used to read and write system variables at run time. While OPC itself is an open standard, using it remotely requires extensive DCOM configuration which is both tedious and error prone, as well as limiting in that it requires the client machine to run a Microsoft Windows operating system. The OpenOPC project provides a solution to this problem. Built on Python, OpenOPC provides a platform agnostic method of making remote OPC requests. It runs a service on the host machine (in our case Windows XP embedded) which responds to requests from the client machine. The host service then proxies the OPC request to the (now local) OPC server, thus bypassing all DCOM related issues. The network communication transmits serialized Python objects ala the Pyro library.

A simple program was written in the KRL language and runs on the KUKA robot computer in parallel with the OPC server. This program sits in an idle loop monitoring the system variables until a command variable changes to True. At this point, the program breaks out of the loop and moves the robot to a position dictated by other system variables which are also set by the client machine. At the completion of the motion, the program re-enters the idle loop and the process repeats.

The external KUKA controller (the client) runs on a separate machine under Ubuntu Linux. This machine makes a connection to the OpenOPC service running on the KUKA computer and makes the appropriate requests to read and write the system variables. In this manner, this external machine is able to specify a desired robot position, either absolute or relative, and then, by setting the command variable to True, forces the robot to execute the motion. This machine also acts as the main control logic, synchronizing the robot motion with the image capturing and object reconstruction.

4.3.2 Wireless Camera and Object Reconstruction: The wireless camera presents itself on the network as an HTTP server where images can be obtained by making CGI requests. These requests are trivial to make using the Python urllib2 module. The data is received in the form of raw JPEG data which must be

Fig. 6: *The objects used for testing. Clockwise from upper-left: (1) A battery box. (2) A stack of cups. (3) A cardinal statue. (4) A ball of yarn.*

converted to RGB raster format for purposes of image processing. This conversion is done using the Python Imaging Library. So that the data need not traverse the network twice, the connection to the camera is made from the object reconstruction program and images are captured and converted upon request by the main control program.

The connection between the main controller and object reconstruction program utilizes the XML-RPC protocol. The object reconstruction programs exports the majority of its capability in the form of methods on a SimpleXMLRPCServer instance from the Python xmlrpclib module.

5. Testing and Results

After verifying the accuracy of the algorithm in simulation, it was implemented on the hardware platform and tested on a variety of real world objects: a prismatic battery box, an elongated cylinder composed of two stacked cups, a ball of yarn, and a small cardinal statue. The first three objects represent the range of geometric shapes frequently encountered in domestic settings: cylindrical, prismatic, and ellipsoidal. It was expected that the algorithm would achieve accurate reconstructions for these shapes. The last object is amorphous and was included to test the robustness of the algorithm when presented with data that is incapable of being accurately described by the superquadric model. In all cases, the test objects were red in color to ease the task of segmentation and facilitate reliable silhouette generation. The four objects tested are shown in the following figure.

As seen previously in the simulated reconstruction, the recovered superquadric models the original object to high a degree of accuracy. On the real world objects, the accuracy of the algorithm was seen to degrade only slightly. Indeed, most parameters were recovered to within few percent of known ground truth. It must be kept in mind, however, that there are several sources of error that

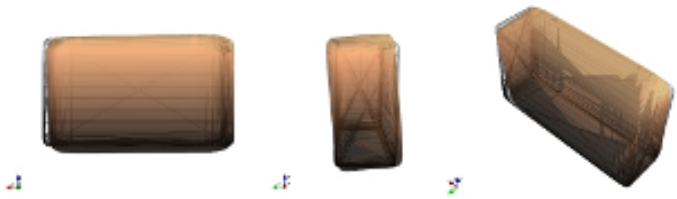


Fig. 7: *The reconstruction of the battery box.*

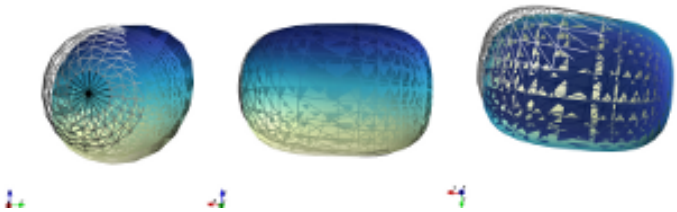


Fig. 8: *The reconstruction of the yarn ball.*

are compounded into these reconstructions which are not present in the simulation:

- Uncertain camera calibration: intrinsics and extrinsics
- Robot kinematic uncertainty
- Imperfect segmentation
- Ground truth measurement uncertainty

The last bullet is particularly noteworthy. Since the object is placed randomly in the robot's workspace, the only practical way of measuring the ground truth position and orientation is to use a measuring device attached to the end effector of the robot. Though more accurate than attempting to manually measure from the robot base, the error is compounded by both machine inaccuracy and human error.

In the following figures, the results of the reconstruction for each of the cases is shown by a rendering of the known ground truth of the object accompanied by an overlay of the calculated superquadric. The ground truth is shown as a wire frame and the reconstruction as an opaque surface.

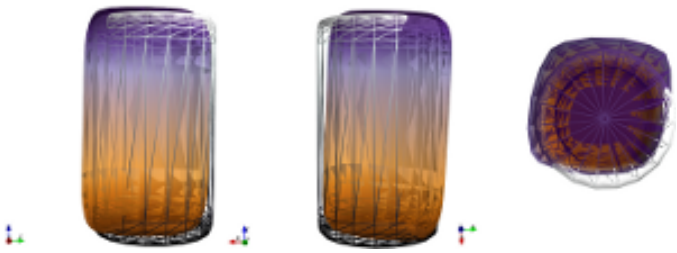


Fig. 9: *The reconstruction of the cup stack.*

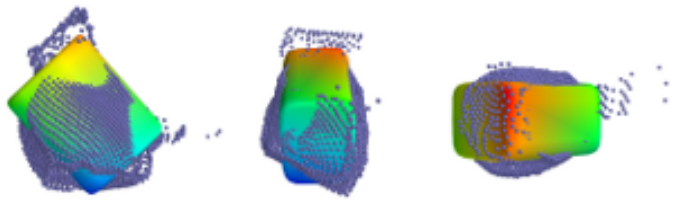


Fig. 10: *The reconstruction of the cardinal statue. This original object is shown in terms of the computed point cloud, given the difficulty of modeling the amorphous shape as a wireframe.*

We feel that the results of the cardinal statue reconstruction are due a bit of explanation. We included this case to test how our algorithm performs when provided with data that does not fit well with our reconstruction model and assumptions, e.g. that the original object can be modeled well by a superquadric. From the figure, it is clear that there would be no way to infer from the box shape that is the final reconstruction that the original object was a cardinal figurine. However, it is interesting to note that the reconstruction is very close to what a human would likely provide if asked to select a bounding box that best describes the object. That is, the reconstructed shape does an excellent job of capturing the bulk form of the statue despite the fact that the data is ill formed with respect to our modeling assumptions.

This example shows that, even when the object does not take a form that can be accurately modeled by a single superquadric, our proposed algorithm still generates useful results.

6. Conclusions

We have given an overview of our robotic vision algorithm that is implemented in Python. Our algorithm enables the recovery of the shape, pose, position and orientation of unknown objects using just three images of the object. The reconstructions have sufficient accuracy to allow for the planning of grasping and manipulation maneuvers.

Both the algorithm and software side of the hardware implementation are implemented entirely in Python and related projects. Notable libraries used include: NumPy, SciPy, Cython, OpenOPC, and scikits.image. This software stack was proven to provide high performance with our algorithm executing in less time than other implementations in the literature.

REFERENCES

[Kim09] D. Kim, R. Lovelett, and A. Behal, *Eye-in-Hand Stereo Visual Servoing on an Assistive Robot Arm in Unstructured Environments*, International Conference on Robotics and Automation, pp. 2326-2331, May 2009.

[Effendi08] S. Effendi, R. Jarvis, and D. Suter, *Robot Manipulation Grasping of Recognized Objects for Assistive Technology Support Using Stereo Vision*, Australasion Conference on Robotics and Automation, 2008.

[Schlemmer07] M. J. Schlemmer, G. Biegelbauer, and M. Vincze, *Rethinking Robot Vision - Combining Shape and Appearance*, International Journal of Advanced Robotic Systems, vol. 4, no. 3, pp. 259-270, 2007.

[Liefhebber07] F. Liefhebber and J. Sijs, *Vision-based control of the Manus using SIFT*, International Conference on Rehabilitation Robotics, June 2007.

[Kragic05] D. Kragic, M. Bjorkman, H. I. Christensen, and J. Eklundh, *Vision for Robotic Object Manipulation in Domesting Settings*, Robotics and Autonomous Systems, vol. 52, pp. 85-100, 2005.

[Colbert10_1] S. C. Colbert, R. Alqasemi, R. Dubey, *Efficient Shape and Pose Recovery of Unknown Objects from Three Camera Views*, International Symposium on Mechatronics and its Applications, April 2010.

[Colbert10_2] S. C. Colbert, R. Alqasemi, R. Dubey, G. Franz, K. Woellhaf, *Development and Evaluation of a Vision Algorithm for 3D Reconstruction of Novel Objects from Three Camera Views*, IEEE International Conference on Intelligent Robots and Systems, 2010. *in-press*.

[Laurentini94] A. Laurentini, *The Visual Hull Concept for Silhouette-Based Image Understanding*, Transactions of Pattern Analysis and Machine Intelligence, vol. 16, Feb. 1994.

[Dyer01] C. R. Dyer, *Volumetric Scene Reconstruction from Multiple Views*, Foundations of Image Understanding, Boston: Kluwer, pp. 469-489, 2001.

[Lippiello09] V. Lippiello and F. Ruggiero, *Surface Model Reconstruction of 3D Objects from Multiple Views*, International Conference on Robotics and Automation, pp. 2400-2405, May 2009.

[Jaklic00] A. Jaklic, A. Leonardis, and F. Solina, *Segmentation and Recovery of Superquadrics*, vol. 20 of Computational Imaging and Vision. Kluwer Academic Publishers, 2000.

[Yamazaki08] K. Yamazaki, M. Tomono, T. Tsubouchi, *Picking up an Unknown Object Through Autonomous Modeling and Grasp Planning by a Mobile Manipulator*, vol. 42/2008 of STAR. Springer Berlin / Heidelberg, 2008.

Unusual Relationships: Python and Weaver Birds

Kadambari Devarajan[‡*], Maria A. Echeverry-Galvis[§], Rajmonda Sulo[‡], Jennifer K. Peterson[§]

Abstract—As colonial birds, weaver birds nest in groups in very particular trees and face specific challenges in the selection and establishment of their nests. Socially-living individuals may organize themselves in particular configurations to decrease the probability of events that could be detrimental to their own fitness. This organization within a selected area could be dictated by biotic factors (such as predation, parasite invasion and/or thievery), or abiotic ones (like solar radiation, and protection from rain, among others), leading to a variety of arrangements. The parameters that individuals might evaluate while establishing/joining a colony help pick the main evolutionary drivers for colonial living. Here, the factors that determine the spatial relationships between the nests in a given tree are computationally modeled. We have built a computational model that explains the spatial arrangement of the nests with bird species, tree morphology, and the environment as factors.

Python has been used significantly in the construction of the model, particularly the machine learning libraries and visualization toolkits. Python is used for the initial data processing, based on which, statistical analysis and visualization are done. We use the PCA and regression tree algorithms to build a model that describes the main factors affecting the spatial arrangement of the nests and classify the nests based on these factors. Visualization is used for determining key attributes in the tree morphology, and nest characteristics, that might be better predictors of overall nest distribution. This aids in guiding other modeling questions. NumPy arrays are used extensively, during the visualization. Mayavi2 is used for the 3-D visualization and matplotlib is used for the representation of the results of statistical analysis.

Index Terms—ecology, evolution, biology, ornithology, machine learning, visualization

Introduction

Group living is a strategy that confers various advantages and disadvantages. These may include better access to mates and protection from predation, but also increases competition for resources and higher visibility. As an evolutionary response to these challenges, socially-living individuals have come to display certain patterns of organization, such as uniform or clumped distributions, to decrease the probability of events detrimental to their own fitness. However, each of these distributions can be modified or adjusted, depending on the scale at which the pattern is analyzed [Jovani07] and the physical space to which the group is confined. Of particular importance are nesting and/or roosting sites, whose spatial arrangement can be very informative about the type of biological and environmental pressures that can determine an organism's life history. Thus, the aggregation patterns of organisms living in groups should be carefully assessed from the individual's perspective as well as a communal point of view.

Determining the key factor(s) that drive the selection of nesting sites and their location poses a challenge because often the boundaries defining the group are unclear. However, birds nesting in a single tree present a unique opportunity to study a group that shares a single space with discrete edges. Some of the most extensive (and studied) colonies are found among sea birds, where it has been established that foraging and predation are the ultimate factors determining colony structure [Burger85]. In terrestrial birds however, it has been proposed that weather, competition, and predation may be the key factors for determining site selection and nest architecture [Crook62], [Schnell73]. Abiotic factors, such as weather and temperature, present challenges for the location of the nest [Ferguson89], while biotic factors like intra- and inter-specific competition and predator deterrence may also play a role [Pitmanc58], [Collias78].

The biological aim of this study was to determine if the location of each weaver bird nest was influenced more by structural or environmental (abiotic) factors, or by the social and behavioral interactions (biotic) of the weaver species studied. We have used computational tools for the analysis of a large dataset and elucidate the main factors determining the complex nest organization in weaver colonies. This provides a unique opportunity to incorporate many computational tools and techniques such as machine learning and visualization for analyzing various aspects of such problems. The approach discussed in this paper has widespread applications in a number of fields and can be modified or extrapolated easily for a range of similar ecological and biological problems involving analysis and modeling.

Python has been used extensively at every stage of computation, from the data processing to the analysis and visualizations. It was the programming language of choice for the data processing due to the simplicity and ease of use of the language. Similarly, for the visualization, Python was preferred, with Matplotlib's [Hunter07] usability and functionality from a plotting perspective, and Mayavi2's [Ramachandran08] scriptability, ease of use, and compatibility with NumPy, being the driving factors. On the other hand, R was initially used for the statistical analysis but later Orange [Orange] (a data mining tool that uses visual programming and Python scripting for the analysis and visualization), Rpy [RPy] (a Python interface to the R programming language), and Mayavi2 [Ramachandran08] (a Pythonic 3-D visualization toolkit) were used for the Principal Component Analysis and the Random Forest methods, since Python seemed to be the perfect umbrella for encompassing the data processing, analysis, and visualization of data. Moreover, Python's vast array of libraries and the different

* *Corresponding author: kadambari.devarajan@gmail.com*
‡ *University of Illinois at Chicago*
§ *Princeton University*

Python-based tools available are appropriate for the diverse set of techniques that were required for the scientific computing [Oliphant07] involved.

Study Area and Methods

Nest surveys were conducted at the 20,000 ha Mpala Research Center in the Laikipia District of central Kenya (0°20′ N, 36°53′ E).

We surveyed a total of sixteen trees and 515 nests. For each nest, its distance from the ground, distance from the trunk, estimated size, entrance direction (entrance facing north, south, west, east, down or so on), distance to its closest neighbor, and general condition (ranging from "dilapidated" to "in construction") were recorded. All measurements were taken with a telescopic graded bar and a decameter, along with compasses. We also determined the species of the bird for each nest, as several trees were cohabited by more than one species. When assessment by field observation was not possible, the bird species was determined based on the nest descriptions given by Zimmerman et al. [Zimmerman96]. Additionally, each tree was mapped and photographed for further reference.

Studied species

1) Grey-capped Weaver (*Pseudonigrita arnaudi*): a gregarious weaver that lives in small dense colonies. It is a common resident of African woodlands below 1400m elevation [Zimmerman96].
2) Black-capped Weaver (*Pseudonigrita cabanisi*): a gregarious species that feeds on the ground. They have strong nests, which are tightly anchored to a single branch. They are commonly found in grassy savannas below 1300m elevation [Zimmerman96].
3) White-browed Sparrow Weaver (*Plocepasser mahali*): a highly social species that lives in noisy flocks. It feeds on the ground, mainly on insects, but will also eat seeds. It is common in bush savannas and dry woodlands below 1400m elevation [Zimmerman96]. This species is known to have feeding grounds that are defended by the colony [Collias78].

Computational Methods

The Python programming language was used for the cleaning of collected data and also for the processing of this cleaned data to obtain the requisite features in a proper format. This cleaned, formatted data is used as input for the machine learning and statistical analysis tools applied. Analysis was done predominantly using the Principal Component Analysis (PCA) and the Random Forest (RF) methods, which were initially implemented in R. This was later completely converted to RPy, and subsequently implemented using Mayavi2. The process of conversion to RPy can be avoided in future studies. Since we want to completely Pythonize the tool suite that we use, we also implemented this using Orange and while Orange simplifies the obtaining of results using PCA and RF, our results are not as clean as in RPy, and require a lot of rework and a better understanding of Orange. Moreover, having the scripting power of Python combined with the statistical power of R was instrumental in the data analysis and speaks volumes of the extensibility of Python. On the other hand, Mayavi2 simplified process of analysis and all the visuals required for the analysis were later rewritten using Mayavi2 and Matplotlib, thereby completely Pythonising the implementation. The visualization was done using Mayavi2 as the primary tool for simulating the trees with the weaver bird nests. NumPy was essential for different aspects of the visualization generation and analysis, and NumPy arrays were crucial for this. All these helped bring the whole suite of tools required for scientific computing under the aegis of Python, where finding another umbrella language to incoporate all these different computational techniques and libraries would have been cumbersome.

Analyses and Interpretation

In order to identify the main factors that explained the local arrangement of the nests of the weaver birds, we applied two machine learning techniques: Principal Component Analysis and Random Forests.

Principal Component Analysis (PCA) is a method for dimensionality-reduction that identifies the underlying factors (or components) that explain most of the variance in data. One of the most widely used versions of this method is the Linear PCA, which assumes a linear relationship between the new factors and the original variables, such that

$$\begin{aligned} P_1 &= a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ P_2 &= a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \ldots & \\ P_d &= a_{d1}x_1 + a_{d2}x_2 + \ldots + a_{dn}x_n \end{aligned}$$

On the other hand, the Random Forest (RF) method constructs an ensemble of decision trees (non-linear models) and outputs an average of their results. Each decision tree uses a bootstrap sample from the original dataset. Also, each node in each tree considers a randomized subset of the original variables. Averaging and randomization are two critical components that make RF a very robust machine learning method [Breiman01]. One important feature of the RF is the computation of variable importance with respect to prediction.

In order to represent the local arrangement of the weaver nests, we used the following variables as the predicted (dependent) variables: normalized nest height with respect to the tree height, normalized nest height with respect to the height of highest nest, and normalized distance of nest with respect to distance of farthest nest.

Visualization

The objective of the visualization was to automate the visualization of each tree using the parameters from the dataset. This was implemented predominantly using the 3D visualization toolkit, Mayavi2, along with Python scripts.

The 3-D visualization of the scientific data was used to explore if any attributes of the tree morphology and nest characteristics could be predictors of the distribution of the nests in a tree and also analyze the distribution of nests among trees in a grove. It provided an opportunity to view the data from an alternative perspective and aided greatly in the analysis. Initially, an idealized high-level model of a tree was made using just the Mayavi2 user interface, with the canopy approximated to a hemisphere and the trunk approximated to a cylinder, with standardized tree coloring. In order to visualize the nests in the trees though, some

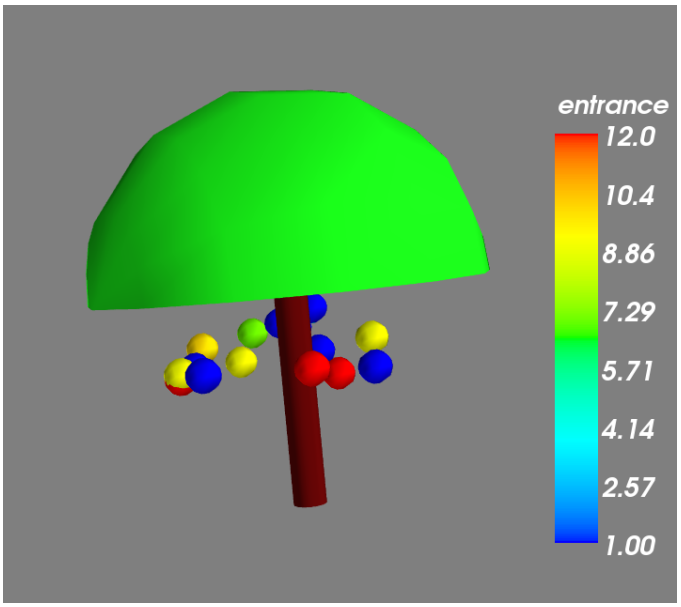


Fig. 1: *3-D visualization of a real tree with weaver bird nests studied at the Mpala Research Station, based on parameters recorded.*

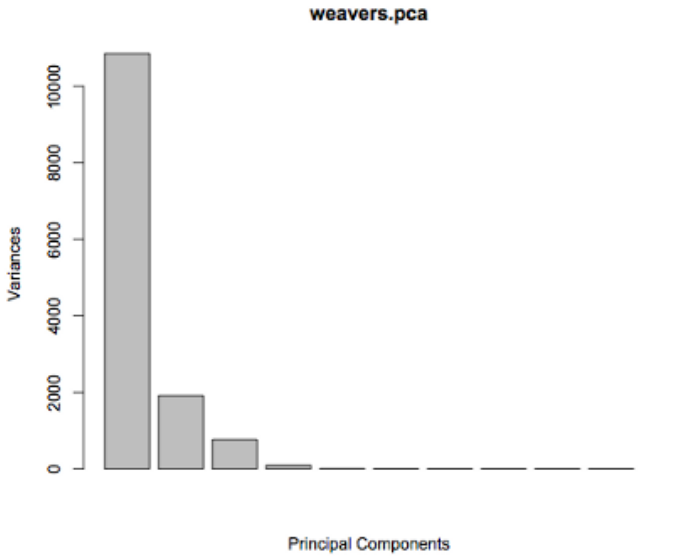


Fig. 2: *The proportion of variance explained by the top five principal components of the weavers nest arrangement at Mpala Research Station.*

scripting functionality was required and the user interface alone was insufficient.

The visualization obtained in Fig. 1 involves nests obtained using Python (and the IPython interface [Perez07]), NumPy, and Mayavi2's mlab interface along with a tree generated using mlab. This involves as input a file containing different parameters such as the number of nests, the height of the nest from the ground, the distance of the nests from the trunk, inter-nest distances, etc., and any tree with the requisite parameters can be simulated. This is automated so that each tree is simulated from the parameters in the dataset. This input from the dataset ensured a geometric view of a tree, with the trunk as a cylinder, the canopy represented as a hemisphere, and nests represented by spheres. As a result of this, we could see the relative position of the nests in each tree and some additional work included the color coding of the nests according to species, the climactic conditions, etc., in an attempt to extend the model for better evaluation and analysis.

Results and Discussion

As shown in Figure 2, there appear to be two or possibly three main factors explaining as much as 99% of the variance in the dataset gathered. Based on spatial constraints, our initial prediction was that tree architecture and requirements of the specific weaver birdspecies would play the most important roles in determining nest arrangement. To test this, we looked into the individual characteristics of the variables predicted by the PCA analysis.

When looking for variables that explain most of the variance, canopy size (total length of the farthest branches in 2 dimensions), number of nests (within each tree), and distance between closer nests emerge as the main variables determining the arrangement (Figure 3). These variables point to tree architecture and structure as the main drivers in the organization and nest disposition within a tree, since they are closely related to the actual space available for placing of the nests. It is important to notice that the bird species played no strong role with respect to the arrangement, alluding to the fact that regardless of the species identity the location of each nest is determined by common "rules" among weaver birds. In biological terms however, it would be interesting to further look into the availability of nesting materials and living resources to determine if the importance of tree architecture deals with its location in the landscape, or if the parameters hold true regardless of the proximity and availability of resources, and then to look at competition between individuals for them.

Of the species analyzed, Grey-capped Weavers and Black-capped Weavers show closest relations with respect to the importance of the variables (Fig. 3), which was also evident in the field since these two species tend to nest in tree together (85.7% of the trees examined with one species had the other present), while the White-browed Sparrow Weaver nests cluster independently in the PCA analysis (Fig. 3).

If we look at the species difference, we can see that the White-browed Sparrow Weaver clearly distinguishes itself from the other studied species (Fig. 4) by building nests closer to one another, in trees with smaller canopies and fewer nests. In contrast, Grey-capped Weavers and Black-capped Weavers present a wide variety of spatial conditions for the nest location (the scattered points in the tri-dimensional cube shown in Fig 4), with a lot of overlap between the data points representing the two species, indicating similar characteristics of the local arrangements of their nests colonies.

When analysing specific trees, 67% of the trees in which nests were found, are represented by *Acacia mellifera*, which generally has a bigger canopy than the other trees studied, that supports a larger number of nests (Fig 5). Another tree species widely surveyed (25% of the total trees) was the *Acacia xanthophloea*, where the canopy is taller but smaller than the former. However, due to its height, it allows for the establishment of nests in multiple levels, creating a different vertical distribution. Finally, *Acacia etbaica* presents a small canopy with reduced number of nests that are closer to each other, which was overall mostly occupied by the White-browed sparrow weaver.

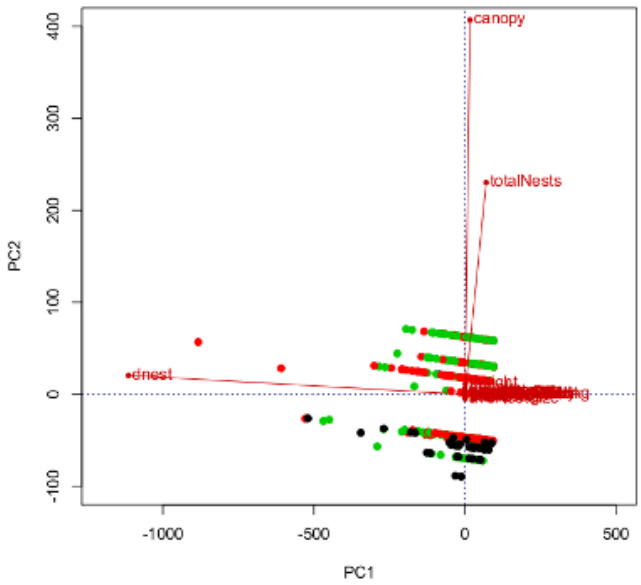


Fig. 3: *Projection of data on the top two principal component axes. Data points are colored by the bird species they represent.*

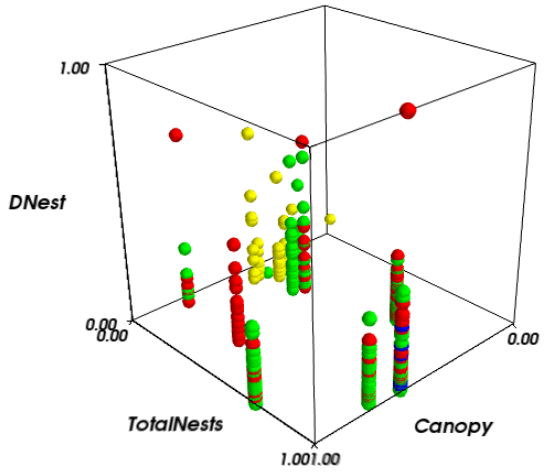


Fig. 4: *3-D plot of the canopy (Canopy), number of nests (TotalNests), and distance between nests (DNest) for each species of weaver bird. Data points are colored-coded for the bird species they represent, with Red denoting the Black-capped Weaver, Green denoting the Grey-capped Weaver, Blue denoting the Speke's Weaver, and Yellow denoting the White-browed Sparrow Weaver.*

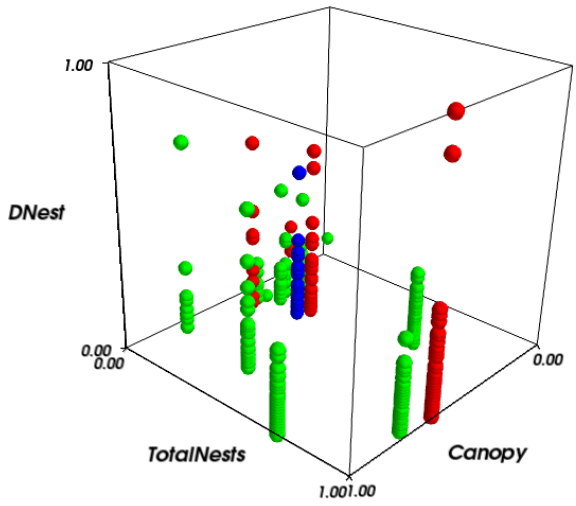


Fig. 5: *3-D plot of canopy (Canopy), number of nests (TotalNests), and distance between nests (DNest). Data points are colored-coded for the tree species in which they exist, with Red denoting* Acacia xanthophloea*, Green denoting* Acacia mellifera*, and Blue denoting* Acacia etbaica*.*

Summary and Future Work

The data collected from the Mpala Research Station was compiled into a table based on different parameters. Apart from this data set, a working database of digital pictures from all trees, rough visualizations and maps, sketches of the trees, and a bibliography was also created. This data was used as input for computational analysis so as to solve different problems such as finding key predictors of the spatial arrangment of the weaver bird nests and evaluating if there exists an "algorithm" that weaver bird follows in choosing a nesting site and building a nest. Machine learning and statistical analysis techniques were used for this. Visualization of the scientific data was also done.

Python was used significantly for the cleaning and pre-processing of the data, the machine learning, and the visualization. The Python programming language and packages associated with it, such as Mayavi2, Orange, RPy, IPython [Perez07], NumPy, etc., were involved in various stages of the scientific computing. Python's power as a general-purpose glue language is also brought out by the variety of tasks it was used for, and also by it's ability to interface easily with R. Under the aegis of Python, the data was visualized, and models for the analysis were built. The visualization is also used to summarize the results obtained visually, apart from aid model the tree-bird-nest system along with other parameters.

A number of features can be built on top of this base model. For instance, a thermal model can be built using the sun's azimuth, wind, rain, and other factors, similar to weather visualization. From a biological perspective, these results grant further research on the specific location of each tree. This might help elucidate if selected trees present specific characteristics within the landscape that grant them as more suitable for the weavers. It would also be interesting to be able to differentiate temporal patterns of occupation in a given tree. It would be informative to determine if nests are located based on the space available or an active preference for clustering. From a computational angle, ongoing

work involves the construction of 3D visualizations of the trees with the nests, with information on orientation to the sun, wind, and other climate data, to determine if any of the variation in the nest arrangement could be due to environmental artifacts. Moreover, one of the goals of the visualization is to automate generation of the trees and nests using a user interface with simply some standard parameters from the dataset. As more data flows in, different problems will be addressed and additional functionality required and Python is thus the perfect environment for a bulk of the computation considering it's extensibility, ability to interface with a variety of packages, the variety of packages available, and it's extensive documentation.

Acknowledgements

We would like to extend out gratitude to professors Tanya Berger-Wolf (the University of Illinois at Chicago, IL), Daniel Rubenstein (Princeton University, Princeton, NJ), and Iain Couzin (Princeton University, Princeton, NJ) for all their input, ranging from the field setup to the computer analysis in this research. We would also like to thank our fellow graduate students in the Department of Computer Science at the University of Illinois at Chicago and the Department of Ecology and Evolutionary Biology at Princeton University. Additionally, the authors would like to thank Prof. Prabhu Ramachandran of the Indian Institute of Technology Bombay and author of Mayavi2 for his help in using Mayavi2, and input (and troubleshooting) for all things Pythonic. Funding for this project was granted by the NSF (CAREER Grant No. 0747369) and by the Department of Ecology and Evolutionary Biology at Princeton University.

REFERENCES

[Breiman01] Breiman, L. Random forests. Machine Learning 45, 5–32.

[Burger85] Burger, J. & Gochfeld, M. Nest site selection by laughing gulls: comparison of tropical colonies (Culebra, Puerto Rico) with temperate colonies (New Jersey). Condor 87: 364-373.

[Collias78] Collias, N. & Collias E. 1978. Nest building and nesting behaviour of the Sociable Weaver (*Philetairus socius*). Ibis 120: 1-15.

[Collias80] Collias, N. & Collias E. 1980. Behavior of the Grey-capped social weaver (*Pseudonigrita arnaudi*) in Kenya. Auk 97: 213-226

[Crook62] Crook, J. H. 1962. A Comparative Analysis of Nest Structure in the Weaver Birds (*Ploceinae*)

[Ferguson89] Ferguson, J.W. & Siegfried, W. 1989. Environmental factors influencing nest-site preference in White-Browed Sparrow-Weavers (*Plocepasser mahali*). The Condor 91: 100-107

[Hunter07] Hunter, J. D. Matplotlib: A 2D Graphics Environment#, Computing in Science & Engineering, vol. 9, 2007, pp. 90-95.

[Jovani07] Jovani, R. & Tella, J. L. 2007. Fractal bird nest distribution produces scale-free colony sizes. Proc. R. Soc. B 274: 2465-2469

[Oliphant07] Oliphant, T. Python for Scientic Computing, Computing in Science & Engineering, vol. 9, 2007, pp 10-20.

[Orange] Orange – Open source data visualization, mining and analysis using visual progamming and Python scripting. http://www.ailab.si/orange/

[Perez07] Pérez, F. and Granger, B.E. IPython: A System for Interactive Scientific Computing, Computing in Science & Engineering, vol. 9, 2007, pp. 21-29.

[Picman88] Picman, J. 1988. Experimental-study of predation on eggs of ground-nesting birds - effects of habitat and nest distribution. The Condor 90: 124-131.

[Pitmanc58] Pitmanc, R. S. 1958. Snake and lizard predators of birds. Bull. Brit. Om. Club. 78: 82-86.

[Pringle07] Pringle, R. M., Young, T. P., Rubenstein, D. I. & McCauley, D. J. 2007. Herbivore-initiated interaction cascades and their modulation by productivity in an African savanna. PNAS 104: 193-197

[Ramachandran08] Ramachandran, P., Varoquaux, G., 2008. Mayavi: Making 3D data visualization reusable. In: Varoquaux, G., Vaught, T., Millman, J. (Eds.), Proceedings of the 7th Python in Science Conference. Pasadena, CA USA, pp. 51-56.

[RPy] RPy – A Python interface to the R programming language. http://rpy.sourceforge.net/

[Schnell73] Schnell, G. D. 1973. Reanalysis of nest structure in weavers (Ploceinae) using numerical taxonomic techniques. Ibis 115: 93-106

[Zimmerman96] Zimmerman, D. A., Turner, D. A. Y Pearson, D. J. 1996. Birds of Kenya and Northern Tanzania. Princeton University Press, New Jersey

Weather Forecast Accuracy Analysis

Eric Floehr[‡*]

Abstract—ForecastWatch is a weather forecast verification and accuracy analysis system that collects over 70,000 weather forecasts per day. The system consists of data capture, verification, aggregation, audit and display components. All components are written in Python. The display component consists of two websites, ForecastWatch.com, the commercial offering, and ForecastAdvisor.com, the free consumer version, both implemented using the Django web application framework. In addition to providing comparative forecast accuracy analysis to commercial weather forecast providers like The Weather Channel, ForecastWatch data and systems have been instrumental in a number of research endeavors. Dr. Eric Bickel, of the University of Texas at Austin, Dr. Bruce Rose of The Weather Channel, and Jungmin Lee of Florida International University have published research using data from ForecastWatch and software written by Intellovations.

Index Terms—weather, forecasting, web

Introduction

The author of this paper has been interested in the weather for all of his life. In 2003, he was curious if there was any difference between forecasts from Accuweather.com, weather.gov (National Weather Service), and weather.com (The Weather Channel). He was also interested in just how accurate weather forecasts were, as there was very little comprehensive data available on the subject. There were a few small studies, comprising a single location or a few locations, and usually only for a period of a few weeks or months. The National Weather Service had comprehensive data on accuracy of their own forecasts, but not others, and the data was not easy to retrieve. At the same time, the author was looking for a new project, and was just exploring new dynamic programming languages like Ruby and Python, after having spent most of career programming in C, C++, and Java. Thus, ForecastWatch was born.

John Hunter, creator of the popular matplotlib library, mentioned in a talk to the Chicago Python group that there is a "great divide" within the people who use Python, with the scientific and financial programming people on one side, and the web application people on the other [Hun09]. I'd like to think my company, Intellovations, through products like ForecastWatch [FW], helps bridge that "great divide". ForecastWatch consists of much back-end calculations, calculating metrics like bias, absolute error, RMS error, odds-ratio skill scores, Brier, and Brier skill scores. However, it also consists of front-end web components. ForecastWatch.com is the commercial front-end to all the calculated quality, verification, and accuracy data, while ForecastAdvisor.com [FA] provides some limited statistics to help consumers make better sense of weather forecasts and their expected accuracy.

* *Corresponding author: eric@intellovations.com*
‡ *Intellovations, LLC*

Architectural Overview

All front-end and back-end processes are in Python. Forecasts are collected through FTP or SFTP feeds from the providers (usually pull), or via scraping public forecasts from provider websites. Observations are pulled from a product by the National Climatic Data Center, consisting of quality-controlled daily and hourly observations from the ASOS and AWOS observation networks. Both forecasts and observations are stored in a MySQL database, along with station meta-data and information that describes how to collect forecasts from each provider and for each station. All data is saved in the form it was collected, so that audits can trace the data back to the source file. Additionally, if errors in parsing are discovered, the source data can be re-parsed and the data corrected in place.

Forecasts are pulled from provider feeds and the web at the same time to ensure that no provider has an unfair advantage. Each provider forecast request is a separate thread, so that all forecasts requests occur at the same moment. This also considerably improves collection times for public forecasts, as network requests are made in parallel. Once the raw file has been collected, either via HTTP, FTP, or SOAP, it is parsed. Text forecasts are normalized and forecast measurements standardized by unit. At time of insertion, a number of sanity checks are performed (low temperature greater than -150 degrees Fahrenheit, for example) and the forecast flagged as invalid if any check fails.

Actual observation files are pulled from an FTP site (U.S.) or via HTTP (Canada) once per month and inserted into the database. Both daily and hourly observations are inserted. The hourly values are used to generate values that do not fall in the 24-hour local time window of the daily observations. For example, some weather forecast providers POP forecasts are for 24-hour local time, others are for 7am to 7pm local time, and the National Weather Service precipitation forecast is for 1200-0000 UTC. The hourly data is also used and as a audit check against the daily values. For example, if the high temperature for the 24-hour local day derived from the hourly data is not within five degrees of recorded daily high temperature, a flag is raised. If there are not enough hourly observations, the entire record is flagged invalid. All this is done in Python using Django's ORM.

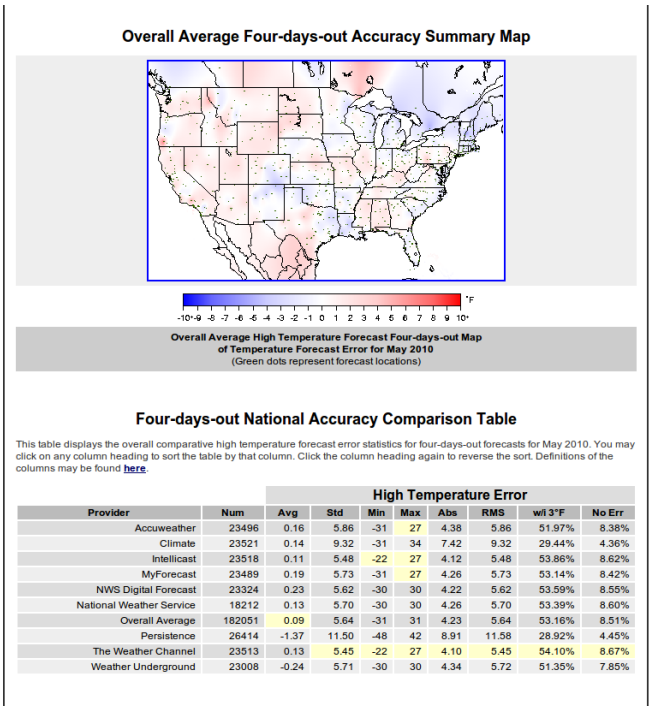
Once the monthly actual observations are in the database, each forecast is scored. Error values are calculated for high temperature, low temperature, probability of precipitation, icon and text forecasts, as well as wind and sky condition forecasts (if available). Once the scores are calculated a second set of audit

checks are performed. Outlier checks are performed on forecast errors, to determine the validity of the forecast. Forecasts with high error are not automatically flagged as invalid, but outliers are often a good indication of an invalid forecast. It has been argued that invalid forecasts should remain, as these forecasts did go out and were potentially viewed, but keeping them severely reduces the utility of the aggregated statistics, as invalid outliers unnecessarily skew the statistics. For example, queries to the National Digital Forecast Database (NDFD) [NDFD] via the National Weather Service web service interface occasionally return a temperature of three degrees Fahrenheit for both high and low. While that is indeed the forecast that was retrieved, it is obviously invalid for most locations and dates. Unfortunately, outlier checking does not catch invalid forecasts that do not result in outlier error. In the three degree forecast example above, it would be difficult to determine an invalid three degree forecast from a valid three degree forecast in far northern U.S. climates in the winter.

Outlier checking is also used to uncover invalid actual observations that were not flagged in the initial sanity checks. It is assumed that forecasts are "reasonable" approximations of observations, with one day out high temperature forecasts, for example, averaging only about 3 degrees Fahrenheit of error. Large one-day out forecast error for any particular observation is flagged as suspect and checked. Sometimes, large forecast error is just a blown model that affected every forecast provider, but other times it is a bad observation. If an observation with large one-day-out forecast error is flagged, it is checked against observations on days before and after, as well as nearby observation sites. One must be careful however, because it is often the outliers that have the most economic value if they can be better predicted. An energy utility, for example, is far more interested in days that fall outside the norm, than the days that are near-normal. Once the audit is complete, aggregations are performed on the raw scores. The scores are aggregated by month, and sliced by nation, state/province, and location, as well as by days out and type. This is performed with raw SQL generated from Python code. The complexity is such that an ORM does not provide any benefit, and in most cases is incapable of generating the queries at all. These aggregations are then used to generate surface plots of error and skill using the mapping tools in the GMT (Generic Mapping Tools) package [GMT].

The aggregate data and maps are primarily displayed in a web application, used by a number of forecast providers, such as The Weather Channel, Telvent DTN, The Weather Network, and CustomWeather. Figure 1 shows a screenshot of how the aggregated data and generated maps are used in a web interface. Not shown in the screenshot for space reasons are navigation tabs above and drill-down links below the screen capture. The user can click on the map to drill-down to the state (or province) level, or view the state summary table (not shown) and click on an individual state in the table to view a list of locations within the state that can be viewed.

Django [DJ] is the web front-end for both ForecastWatch and ForecastAdvisor.com. It can be used to quickly build robust, dynamic websites. For example, Dr. Bruce Rose, Principal Scientist and Vice President at The Weather Channel, is studying snowfall forecast accuracy [Ros10]. There is a common perception that snowfall forecasts are "overdone". Specifically, that forecasts of snowfall generally predict more snowfall than actually occurs. Despite this common perception, little scientific research has been done to verify snowfall forecasts. Dr. Rose wanted a public site



		High Temperature Error							
Provider	Num	Avg	Std	Min	Max	Abs	RMS	w/i 3°F	No Err
Accuweather	23496	0.16	5.86	-31	27	4.38	5.86	51.97%	8.38%
Climate	23521	0.14	9.32	-31	34	7.42	9.32	29.44%	4.36%
Intellicast	23518	0.11	5.48	-22	27	4.12	5.48	53.86%	8.62%
MyForecast	23489	0.19	5.73	-31	27	4.26	5.73	53.14%	8.42%
NWS Digital Forecast	23324	0.23	5.62	-30	30	4.22	5.62	53.59%	8.55%
National Weather Service	18212	0.13	5.70	-30	30	4.26	5.70	53.39%	8.60%
Overall Average	182051	0.09	5.64	-31	31	4.23	5.64	53.16%	8.51%
Persistence	26414	-1.37	11.50	-48	42	8.91	11.58	28.92%	4.45%
The Weather Channel	23513	0.13	5.45	-22	27	4.10	5.45	54.10%	8.67%
Weather Underground	23008	-0.24	5.71	-30	30	4.34	5.72	51.35%	7.85%

Fig. 1: *Figure 1. screenshot of portion of ForecastWatch web interface.*

that would collect the snowfall forecasts and observations, and provide an intuitive, easy-to-use, dynamic data-driven site that updated automatically when data came in. One of the big challenges in science and scientific research is the increasingly large amounts of data research is based on. Challenges of curation, storage, and accessibility are becoming more frequent. "Climategate" brought the issue of reproducibility of research when large amounts of data are used, as the raw data on which several papers were based was found to have been deleted. While this does not invalidate the research, it does present a credibility issue, and puts roadblocks in one of the tenets of the scientific method: that of reproducibility. Python and Django were used to create a data-driven site that allowed all the data to be navigated and explored.

Some Findings

ForecastWatch started as an answer to the question "Is there any difference between weather forecasts from different providers?" It turns out there is a difference. As an amateur scientist, it has been interesting to look at all the data in a number of different ways. While many forecast providers perform continuous internal verification of forecasts, and the National Weather Service has an entire group devoted to it, there has been little information communicated at the popular level regarding weather forecast accuracy. One of the goals of ForecastWatch is to help meteorologists educate their customers as to their accuracy, and begin to help dissipate some of the skepticism that is reflected in comments such as "I wish I could have a job where I'm wrong half the time an still keep my job".

Figure 2 shows a histogram of one-day-out and four-day-out high temperature forecast forecast error against 24-hour high observations from all providers over all of 2009. There are nearly two million forecasts represented in each day's histogram. As expected, but nice to confirm, the histogram of high temperature forecast error follows a normal distribution. As also might be

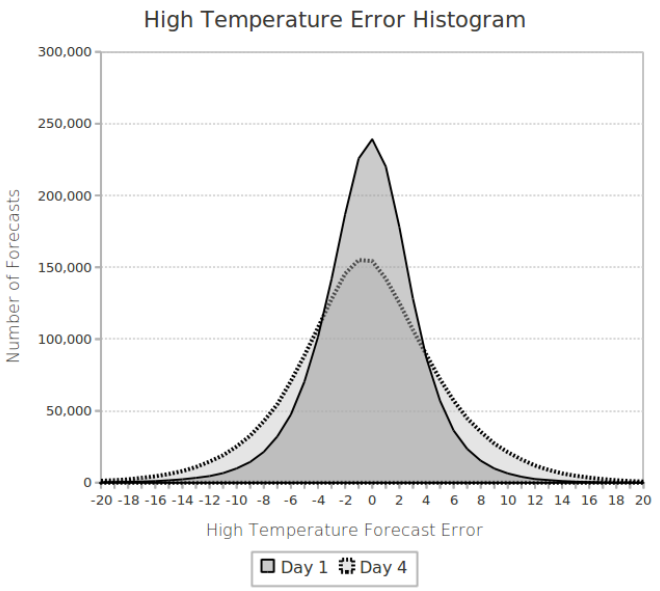


Fig. 2: *Figure 2. High temperature error histogram.*

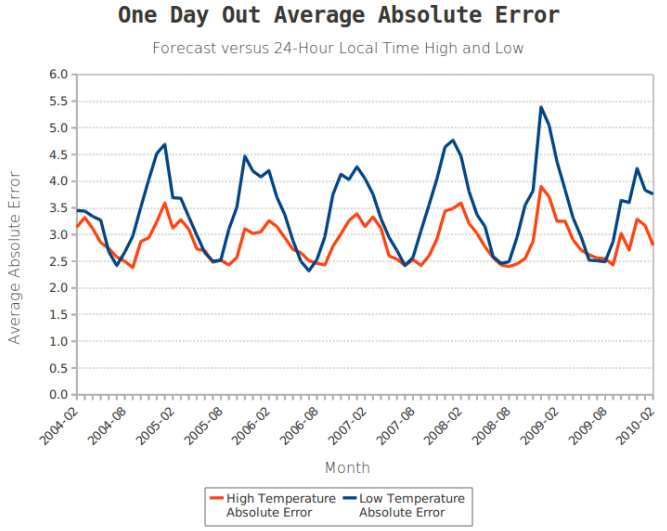


Fig. 3: *Figure 3. High and low temperature forecast error by month.*

expected, the histogram for four-day-out forecasts is more spread out than that of the one-day-out forecasts. The further out the forecast is for, the greater the standard deviation of error. Eagle-eyed readers may notice that the histogram "leans" slightly negative, meaning that average error has a light negative bias. The reason for this is subtle, and demonstrates the care that must be taken when interpreting results.

This histogram represents the error of forecasts when compared against the 24-hour high temperature reported in the daily observations. However, some forecasters' valid time for high temperature is 7am to 7pm local standard time. While nearly all high temperatures for the day fall in this period, very rarely they do not. In this case, the 24-hour high observation will be higher than the high temperature between 7am and 7pm. Thus, the forecast will under-predict the high from the perspective of the 24-hour high temperature verification. This leads to the slight negative bias. In general, short-term temperature forecasts are well-calibrated and bias corrected. Generating a high or low temperature observation between an hourly range (for example, 7am to 7pm) also results in a slight error bias. This is because hourly observations are taken at a specific time. The odds are high that the true high or low temperature in a span will occur intra-hour. The probability that a single observation each hour will capture the true high temperature is small, and thus the generated high or low temperature will be lower than the actual high. The 24-hour high and low temperature observations are nearly continuous and reflect the true high and low temperatures of the day.

One fact of weather forecasts that consistently surprises people, even people using weather forecasts in quantitative modeling and decision-making is that weather forecast accuracy is seasonal, and varies greatly geographically. There are many people using weather forecasts as input to risk and prediction models that do not factor in seasonality or location along with the temperature forecast. Figure 3 shows the accuracy of U.S. and Canadian temperature forecasts for the past six years. Temperatures are more accurate in the summer than winter, with high temperature accuracy swinging by one degree and low temperature accuracy even more. Additionally, a high temperature forecast for Atlanta in July has less error on average than a high temperature forecast for Chicago in December.

ForecastWatch also generates skill measures, by comparing unskilled forecasts with skilled predictions. An unskilled forecast is a forecast that requires no skill to produce. The two unskilled forecasts that are used by ForecastWatch are persistence forecasts and climatology forecasts. A persistence forecast is a forecast that says "tomorrow, and the next day, and the next, etc. will be exactly like today". If the high temperature is 95 degrees Fahrenheit today, the persistence forecast will be for 95 degrees Fahrenheit tomorrow. If it is raining today, the prediction will be that it will be raining tomorrow. The climatology forecast will predict that the high and low temperature will be exactly "average". Specifically, the ForecastWatch climatology forecast uses the daily climatic normals (CLIM84) from the National Climatic Data Center [NCDC] which are statistically fitted daily temperatures smoothed through monthly values.

Figure 4 shows high temperature forecast accuracy by days-out for 2009 between the two unskilled forecasts, and the average accuracy of all providers' forecasts. The climatic unskilled forecast is a straight line because the climatic forecast for a given day never changes. It is always the calculated 30-year average temperature as expressed by the nearest station in the CLIM84 product. The two intersections between the forecast error lines are the most interesting features of this figure. The first intersection, between the unskilled persistence forecast and the climatology forecast, occurs between the one- and two-day-out forecasts. This means that a persistence forecast is only better than climatology at predicting high temperature one day out. After one day out, climatology has more influence than local weather perturbations.

Possibly the more interesting intersection is between skilled forecast providers and climatology forecasts between eight and nine days out. What this graph is saying is that weather forecasts from weather forecast providers are **worse** than an unskilled climatology forecast beyond eight days out. The American Meteorological Society said in 2007 that "the current skill in forecasting daily weather conditions beyond eight days is relatively low" [AMS07] in a statement on weather analysis and forecasting. This graphs shows how "relatively low" the skill really is. One

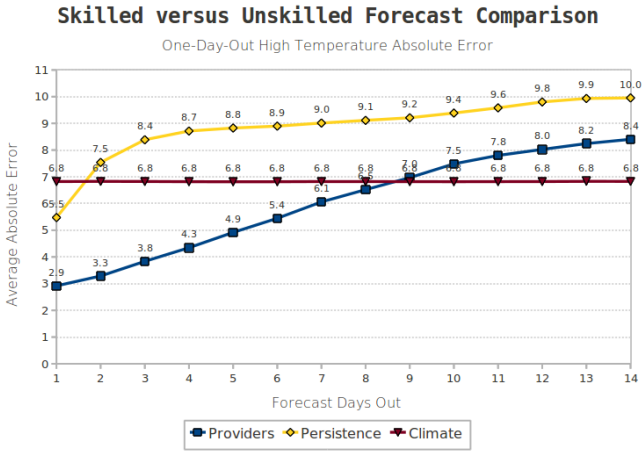


Fig. 4: *Figure 4. High temperature skilled versus unskilled forecasts.*

question that is asked about this is why do forecasters not replace their forecast with the climatology forecast for their nine-day and beyond forecasts? One reason is that these extended forecasts might be skillful in forecasting temperature trends (above or below normal) which the climatology forecast cannot do. Research is ongoing on this aspect of longer-term forecasts.

REFERENCES

[FW] ForecastWatch Website
[FA] ForecastAdvisor Website
[GMT] Generic Mapping Tools
[DJ] Django Web Application Framework
[NDFD] NWS National Digitial Forecast Database
[NCDC] National Climatic Data Center
[Hun09] "Matplotlib, the popular 2D Plotting Library." blip.tv. 04 September 2009
[Tim09] "Climate change data dumped" The Times Online. 29 November 2009
[Ros10] "Are Snow Accumulation Forecasts Generally Overdone?"
[Bic08] Bickel, J. Eric and Eric Floehr. 2008. "Calibration of Probability of Precipitation Forecasts", INFORMS Annual Conference, Decision Analysis Track, Washington, DC.
[Bik10] "Comparing NWS POP forecasts to third-party providers"
[Lee08] "Economic Bias of Weather Forecasting: A Spatial Modeling Approach"
[AMS07] "AMS Statement on Weather Analysis and Forecasting"

Rebuilding the Hubble Exposure Time Calculator

Perry Greenfield‡*, Ivo Busko‡, Rosa Diaz‡, Vicki Laidler‡, Todd Miller‡, Mark Sienkiewicz‡, Megan Sosey‡

Abstract—An Exposure Time Calculator (ETC) is an invaluable web tool for astronomers wishing to submit proposals to use the Hubble Space Telescope (HST). It provide a means of estimating how much telescope time will be needed to observe a specified source to the required accuracy. The current HST ETC was written in Java and has been used for several proposing cycles, but for various reasons has become difficult to maintain and keep reliable. Last year we decided a complete rewrite—in Python, of course—was needed and began an intensive effort to develop a well-tested replacement before the next proposing cycle this year.

This paper will explain what the ETC does and outline the challenges involved in developing a new implementation that clearly demonstrates that it gets the right answers and meet the needed level of reliability (astronomers get cranky when the calculator stops working on the day before the proposal deadline). The new ETC must be flexible enough to enable quick updates for new features and accommodate changing data about HST instruments. The architecture of the new system will allow Python-savvy astronomers to use the calculation engine directly for batch processing or science exploration.

Testing is a very large component of this effort, and we discuss how we use existing test cases, as well as new systematic test generators to properly explore parameter space for doing test comparisons, and a locally developed test management system to monitor and efficiently analyze thousands of complex test cases.

Index Terms—astronomy, telescope, Java, NASA

Introduction

Observing time on the Hubble Space Telescope (HST) is quite valuable. One orbit of observing time (typically 45 or fewer minutes of on-target time) costs on the order of $100K. Understandably, no one would like to waste that. As a result, understanding how much time is needed to accomplish the planned science is important. Asking for too little or too much will result in wasted telescope resources. Hence, the need for exposure time calculators.

ETCs are used to answer important questions such as: how long must one observe to achieve a desired signal-to-noise ratio, or what signal-to-noise ratio can one expect for a planned observation? Computing these quantities requires specifying many input parameters. These fall into few basic categories.

- What instrument will be used? What kind of observing mode (e.g., imaging, spectroscopy, coronography, or target acquisition). What filters or gratings? What detector parameters?
- What kind of source? In particular, what are the spectral details of the source. Any dust extinction or redshifts involved?
- How bright is the source? How is the brightness to be specified?
- How much area around an image or spectral feature do you plan to use to extract signal? Is it a point source or is it extended?
- What kinds of sky background levels do you expect?

All of these choices affect the answer. Some of them involve many possible options.

In the past, peak ETC request rates averaged 10 calculations per minute. Although much computation is involved for each request, the request rate is never terribly high.

History

Despite the clear utility of ETCs they were not part of the original HST software plan, and it was several years before they were officially supported. A number of precursors arose from essentially grass-roots efforts. By the late 1990s web-based CGI ETCs, most written in C, had begun appearing for some of the HST instruments. Around 2000 an effort to provide an integrated GUI-based proposing environment was started. Since ETCs share many of the input parameters with the proposing tools, it was believed that it made sense to integrate these tools into one GUI application, and a large effort was begun to do just that. It was based on a prototype application developed in Java by NASA (The Science Expert Assistant [Gro00]) and became the Astronomers Proposal Tool (APT, http://apt.stsci.edu).

Despite the original intent to develop an integrated application, it was eventually judged that a web-based interface was much more important, and at that point the ETC functionality began a voyage out of the APT GUI and eventually became an entirely separate application again. In the process, it had become much more complicated because of the intrinsic complexity of the design driven by the GUI interface. Over time much of the GUI heritage was removed, but it still had many lingering effects on the code and its design.

The addition of support for more instruments and modes, along with pressure to meet proposal deadlines, led to increasing complexity and inconsistencies, particularly with regard to testing and installations. Unrealized to many, it was quickly becoming unmaintainable. For the Cycle 16 proposal period (HST runs a roughly annual proposal cycle), it worked reasonably well and was quite reliable. That all changed in Cycle 17, where we experienced server hangs approximately every 5 minutes when under peak load. Many months of work to improve the reliability of the server

* Corresponding author: perry@stsci.edu
‡ Space Telescope Science Institute

actually ended up with worse reliability. Even worse, the cycle of making a change, and then delivering and testing the change could take weeks or months leading to an unacceptably long iteration cycle.

At this point an evaluation was made of the underlying problems. These consisted of:

- Large code base. The number of Java lines of code had grown to over 130K and was viewed as unnecessarily complex.
- Database inflexibility. Information was being stored in it that made it impossible to run more than one instance or version of an ETC on a given machine, leading to testing and installation complications.
- Lack of automated build and testing. Installations involved many manual, context-dependent steps
- It was difficult to run tests and to examine results.
- Tests were represented in many different formats and require much massaging by the software to run.
- Test results varied from one system to another because of database issues.
- Lack of tests of javascript functionality.
- Resource leaks could hang server.
- Uncaught exceptions would hang server.
- Instrument parameters located in many different locations in different kinds of files.

Rebuilding

Rather than try to fix these with the existing code base, we decided to re-implement the ETCs in Python. This was partly because we (Science Software Branch) write relatively little software in Java now and have comparatively little expertise in it, Additionally, one of the key tools used by the ETC (pysynphot [Lai08]) is written in Python, so interactions with the ETC would be simplified. Rewriting the entire code base solely in Python also dramatically decreased the overall length of the code.

A rewrite was begun in April 2009 with a proof of concept computational prototype. After approval to go ahead in June, a major effort began to design and implement a new system. The new design had a number of requirements it had to meet:

- One-step install
- Ability to support multiple installations on the same computer
- Consistent test scheme
- Nightly regression testing
- Separation of web and compute functionality
- Ability to script ETC calculations from Python without a web server
- Use of standard Apache/Database server schemes to handle failover and load balancing
- Simple database structure
- Concentrate instrument information in one place
- Use automatic test generation for better parameter space coverage
- No XML
- No cached results
- It had to be ready for Cycle 19 proposal preparation
- Minimal changes to the user interface
- Dispense with interactive form features that weren't working well in the old ETCs

Django was used for the web framework and user interface. Our use of its features is fairly light, but even so, it made the web side of the system fairly easy. Taking a lesson from the past ETC, we made the use of Django's database as simple as possible. One goal was to minimize the need to change the database schemas during operations. Since ETCs take many parameters for the sources and the instruments, there are many potential fields for a database, and it is likely that many of these would change or be added. Yet there is rarely any need to query the database for values of these fields. For those occasions, it would probably be best to specially create a new database for such queries. All the input and output information is encapsulated in a string which is then stored in the database.

Validation Testing

The validation of the new ETCs is simpler in one aspect: we only need match the results of the previous ETC, even if we believe the previous results are incorrect. Any identified discrepancies believed to be errors in the original code were identified as such and noted for later work. If there is time for the instrument groups to address the issue, waivers for differences can be obtained.

It might seem counter-intuitive to use this approach, but it works well in our environment. The software developers cannot always authoritatively answer scientific questions, so we often rely on the appropriate instrument group. But they are not always available to answer our questions quickly due to other priorities.

By using the old ETC as a reference, we can remove the instrument group from our work flow. This reduces their workload, because they are not directly involved in the new development. As software developers, it reduces our cycle time to test a new feature: Instead of asking a scientist to manually perform a detailed analysis of a result, we can simply compare it to the same result from a system that has previously been accepted as correct.

Our target for the maximum difference was generally 1%, though we were permitted to allow differences as much as 5% from the HST project if helpful for meeting the schedule.

On the other hand, migrating the existing tests proved more work than expected because of the many forms such tests took, and the many issues in determining the proper mapping of test parameters to the old and new ETCs. The typical test migration process was to start with custom code to handle any special cases for parameter migration, run a batch test case migration, run the tests, and from the errors, fix migration errors and iterate until all remaining errors were purely computational issues.

The reference results from the old ETC were obtained by running it through its web interface using the mechanize module. The most important information on the result was the ID of the request, which was then used to retrieve the very extensive log files that were generated on the server side which contained the values of the final results and many intermediate values. These also proved invaluable in tracking down where results diverged between the old and the new.

The old ETC had tests in two basic forms (with many variations in details). Some appeared as XML files with one test per file. Others as CSV files, with one test per row. In both cases most were generated manually. We desired a more systematic way of testing parameter space, so a special module was written to generate test cases automatically. In this way we can define

Daily (2010-05-13)	count	pass	fail	error
	8705	7234	865	606
engine/*	7068	5794	668	606
server/*	1626	1429	197	0
web/*	11	11	0	0
Engine Only	count	pass	fail	error
	7068	5794	668	606
engine.*	2	2	0	0
migrated/*	6963	5690	668	605
spider/*	103	102	0	1

TABLE 1: *The report of test results from one night's test run. Count refers to the number of tests in that category; pass refers to the number that run and match the expected results to within the specified threshold; fail refers to the number of tests that produce results but do not match all results to the required threshold; and error indicates the number of tests that fail to produce all necessary results.*

whole sets of tests by providing specific lists of parameter values for specific parameters and construct combination of parameter sets by using tools to generate specific test sets by varying one parameter at a time (akin to traveling along parameter axes), or by generating all combinations (filling the parameter space with a grid of points). One can combine subspaces of parameters in analogous ways. There is a mechanism to make concatenating disjoint sets of parameters that correspond to radio button subsets easy.

We have nightly regression tests running more than 8000 cases a night. Initially the reference results are those taken from the old ETC. Once commissioning is complete, the reference results will be a snapshot of the new ETC results to ensure that future software updates do not change the results in unexpected ways. Table 1 shows an example of a single night's run.

Current Status

To date all of the supported instrument modes have been implemented as far as the calculation engine goes. Most reporting and plotting functionality is in place. Nearly all migrated tests run, though there are still discrepancies being resolved for a few modes. These discrepancies are expected to be understood within a month. The new ETC has approximately 22K lines of code in the web and engine components. A further 5K lines of code were written to support the testing effort. This includes conversion of test data, running tests of the old ETC, comparing results, etc. The new ETC uses a similar form interface, and generates output pages similar (though not identical) to that of the previous ETC.

Figure 1 shows an example of an input form. Figure 2 shows the results obtained from that form, and Figure 3 shows plots of related information associated with those results.

Plans

The ETC must be operational by December 2010. Future activities include web security analysis, load testing, through-the-browser tests (manual and automatic), and documentation.

This ETC framework will be the basis of the James Webb Space Telescope ETCs. JWST is expected to be launched in 2015. Work has begun on understanding what features will be needed for JWST that don't already exist for the HST ETCs. Besides providing the instrument performance information, it is already

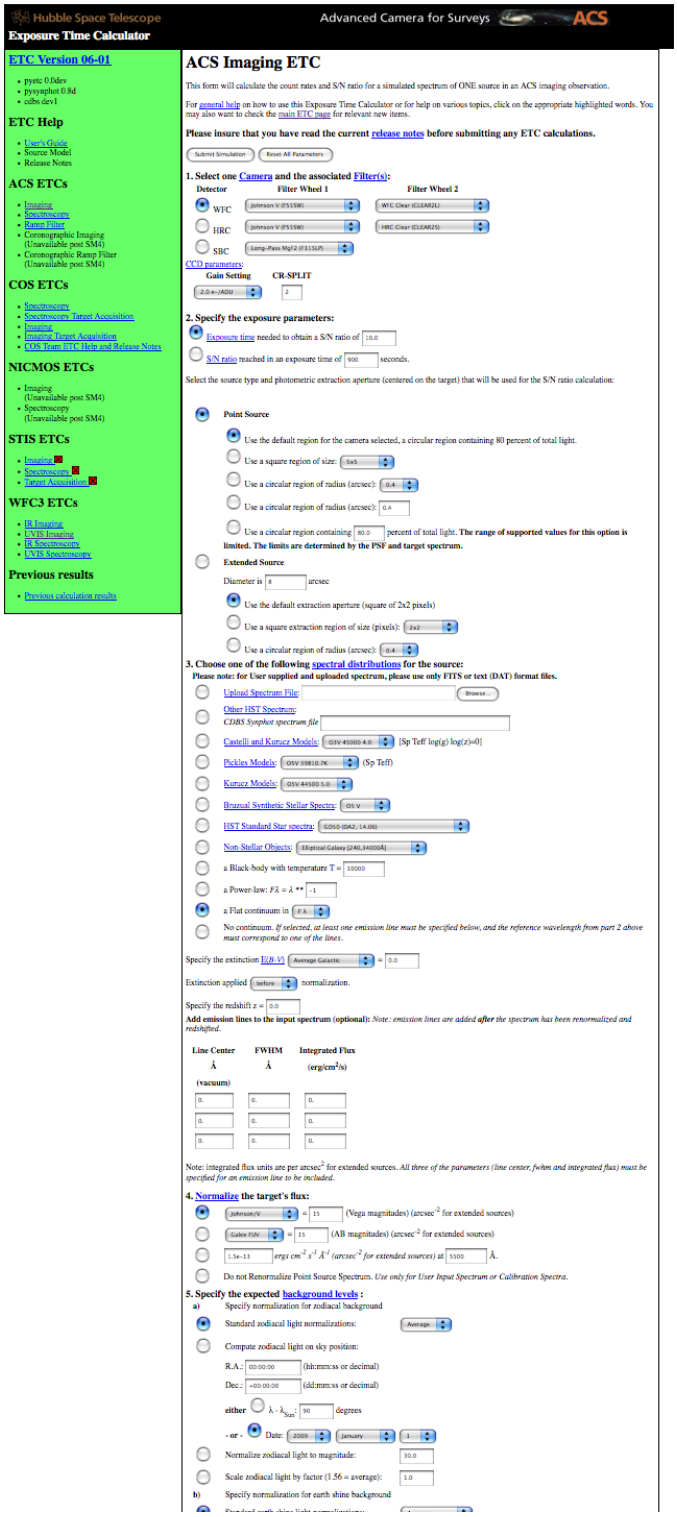
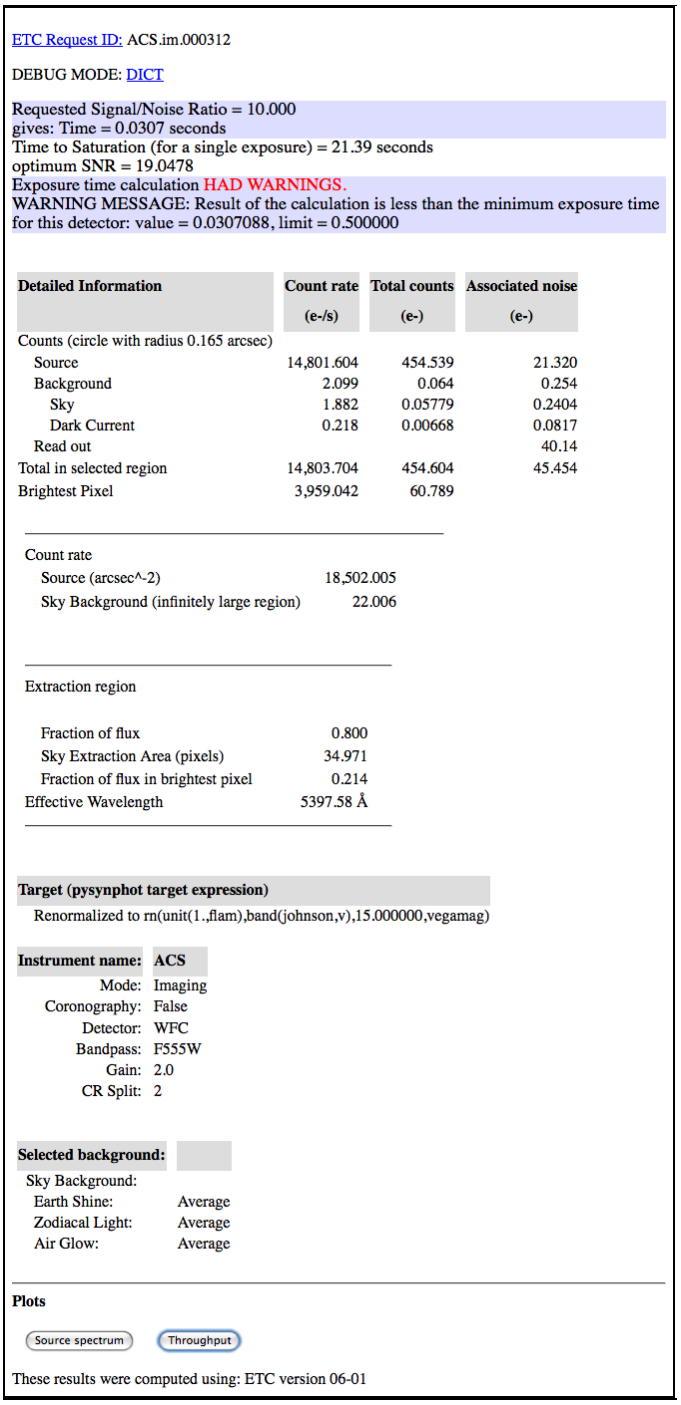


Fig. 1: *Part of the input form for the Advanced Camera for Surveys. This shows most of the choices available to users.*



ETC Request ID: ACS.im.000312

DEBUG MODE: DICT

Requested Signal/Noise Ratio = 10.000
gives: Time = 0.0307 seconds
Time to Saturation (for a single exposure) = 21.39 seconds
optimum SNR = 19.0478
Exposure time calculation HAD WARNINGS.
WARNING MESSAGE: Result of the calculation is less than the minimum exposure time for this detector: value = 0.0307088, limit = 0.500000

Detailed Information	Count rate (e-/s)	Total counts (e-)	Associated noise (e-)
Counts (circle with radius 0.165 arcsec)			
Source	14,801.604	454.539	21.320
Background	2.099	0.064	0.254
Sky	1.882	0.05779	0.2404
Dark Current	0.218	0.00668	0.0817
Read out			40.14
Total in selected region	14,803.704	454.604	45.454
Brightest Pixel	3,959.042	60.789	

Count rate
Source (arcsec^-2) 18,502.005
Sky Background (infinitely large region) 22.006

Extraction region
Fraction of flux 0.800
Sky Extraction Area (pixels) 34.971
Fraction of flux in brightest pixel 0.214
Effective Wavelength 5397.58 Å

Target (pysynphot target expression)
Renormalized to rn(unit(1.,flam),band(johnson,v),15.000000,vegamag)

Instrument name: ACS
Mode: Imaging
Coronography: False
Detector: WFC
Bandpass: F555W
Gain: 2.0
CR Split: 2

Selected background:
Sky Background:
Earth Shine: Average
Zodiacal Light: Average
Air Glow: Average

Plots
Source spectrum | Throughput

These results were computed using: ETC version 06-01

Fig. 2: *The results page shown corresponding to the input parameters shown in Figure 1.*

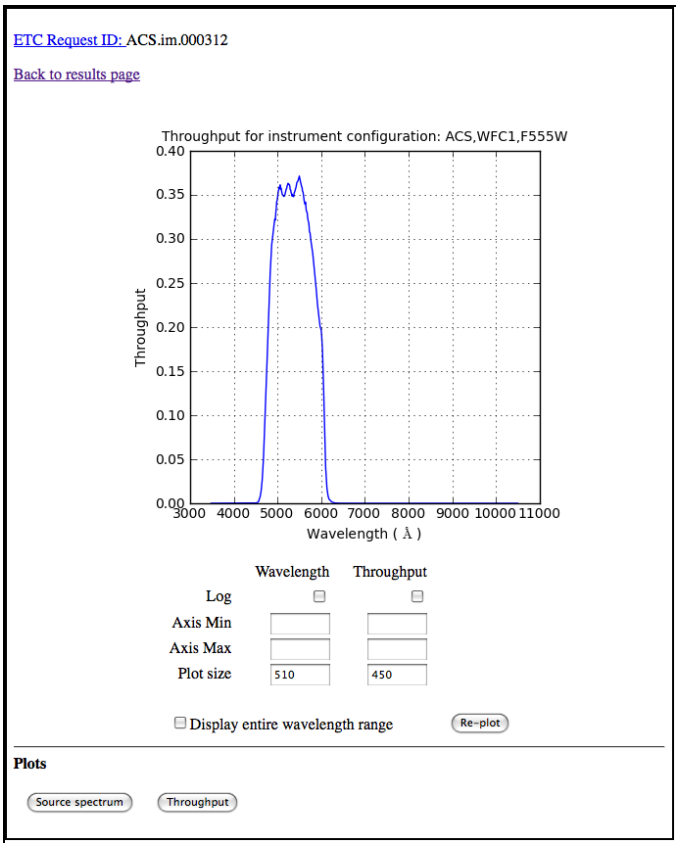


Fig. 3: *One of the plot options for the results shown in Figure 2. In this case the instrument throughput is shown as a function of wavelength for the selected observing mode.*

clear that much more sophisticated sky background models will be needed to be developed to determine which of several detector operations modes will yield the best signal-to-noise ratio.

Furthermore, JWST has requirements to schedule observations at times that do not degrade signal-to-noise too much (due to varying sky background levels that depend on the time of year the target is observed). As such, the scheduling system will need to obtain this information from the ETC. There is also a desire for the proposal preparation tool to be able to use the ETC to determine the optimal detector operating mode for each exposure.

We will be importing all the data regarding instrument performance as it relates to ETC calculations into our Calibration Data tracking system (not possible with the older ETC because of the dispersed nature of the data).

The ETC also provides tables of results to the observatory scheduling system which helps detect when bright sources may pose a health and safety hazard to the instrument in use.

The ETC computational engine will be made available with an Open Source License (BSD/MIT) when the production version is completed.

Conclusions

The rewrite has resulted in a far smaller and consistent code base. More importantly, we can test on the same system that is used operationally. The cycle of building, delivering, and testing the software now can be done in hours instead of weeks giving us far greater ability to fix problems and add enhancements. Django, and our pre-existing tools (matplotlib, pysynphot) greatly facilitated this effort. We will be in a much better position to adapt to JWST ETC requirements.

There were certainly general lessons to be learned from this experience and other work we've done. In coming up with this list, we are generalizing about some issues that didn't necessarily affect this project. Among them:

- There is a big difference between scientific programming as most scientists do it, and what is needed for operational purposes. Table 2 contrasts some of the differences in approach that

Scientist	Operations
Ad-hoc changes to handle various needs	One code base to handle all needed alternatives
Corner cases often ignored	Special cases given more attention
Little attention to user interface	Much more attention to user interface
Minimal error checking	Extensive error checking
No version control	Version Control
No unit or regression tests	Extensive tests
Minimal documentation	More extensive documentation
Refactoring rare	Hopefully not...

TABLE 2: *Comparison of attributes of software developed by researchers to those of software developed for widespread or operational use.*

one usually sees. This isn't to say that scientists couldn't benefit from some of the approaches and tools for operational software (often they could), it's just that that they usually don't use them. These differences result in important management issues discussed later.

- Databases are a double-edged sword. They clearly have important uses, particularly for web applications. On the other hand, they introduce a number of strong constraints on flexibility and ease of distribution. Think carefully about what you use them for and when you really need it.
- Resist temptation to continually put new features over internal coherence. Refactor when needed.
- Routine builds and testing are extremely important. The installation process needs to be as automatic as possible.
- Test on the same machine (or as identical an environment as possible) to be used for operations (at least a subset of the full tests).
- No matter how much analysis you do up front about the design, you probably won't get it right. Be ready to redo it when you face the real world.
- It has to work for all cases, not just the common ones. Even crazy input parameters must at least give a useful error message that will help the user identify the problem.

Complicating the interface between the astronomers and developers is the fact that many astronomers have written programs for their research purposes, but have never had to write programs for distribution or operational settings, and have never had to support software they have written. As a result many astronomers do not appreciate the effort required to produce reliable and distributable software that can be used by individuals or complex systems. That effort is typically up to an order of magnitude more than needed to get software that works for their particular need. It is not unusual to see astronomers become frustrated at the effort required for implementation when they think they could have done it in one fifth the time. As important as any programming, software engineering, or management technique, is the management of the expectations of such customers, and resistance against such expectations driving software into an unmaintainable state.

REFERENCES

[Gro00] S. R. Grosvenor, C. Burkhardt, A. Koratkar, M. Fishman, K. R. Wolf, J. E. Jones, L. Ruley. *The Scientist's Expert Assistant Demonstration*, Astronomical Data Analysis Software and Systems, IV, 216, 695-698.

[Lai08] V. Laidler, P. Greenfield, I. Busko, R. Jedrzejewski. *Pysynphot: A Python Re-Implementation of a Legacy App in Astronomy*, Proceedings of the 7th Python in Science Conference, 2008, 36-38.

Using Python with Smoke and JWST Mirrors

Warren J. Hack‡*, Perry Greenfield‡, Babak Saif‡, Bente Eegholm‡

Abstract—We will describe how the Space Telescope Science Institute is using Python in support of the next large space telescope, the James Webb Space Telescope (JWST). We will briefly describe the 6.5 meter segmented-mirror infra-red telescope, currently planned for a 2014 launch, and its science goals. Our experience with Python has already been employed to study the variation of the mirror and instrument support structures during cyrogenic cool-down from ambient temperatures to 30 Kelvin with accuracies better than 10 nanometers using a speckle interferometer. Python was used to monitor, process (initially in near real-time) and analyze over 15 TB of data collected. We are currently planning a metrology test that will collect 10 TB of data in 7 minutes. We will discuss the advantages of using Python for each of these projects.

Index Terms—astronomy, telescope, NASA, measure, real-time, big data

Introduction

The James Webb Space Telescope (JWST) will be NASA's next Great Observatory. It will be an infrared-optimized telescope with a 6.5m primary mirror made up of 18-separate segments which will be launched no sooner than 2015 by an Ariane 5 into an orbit at the second Langragian (L2) point. This orbital position, about 1.5 million km from the Earth, keeps the Sun behind the Earth at all times making it easier to shield the telescope and keep it cool. The Hubble Space Telescope (HST), by comparison, is a telescope in a 570km high orbit with a solid 2.4m primary mirror optimized for UV and optical observations. A lot of effort will go into building and testing JWST, as it did with HST, to get it to work as desired and as reliably as possible once launched. However, unlike HST, there will not be any possibility of performing a repair mission. The primary structure of JWST will be made of carbon-fiber composites in order to be lightweight enough for launch while still providing the necessary strength and rigidity to support such a large set of mirrors and instrument packages. The primary mirror itself will be composed of 18 separate hexagonal segments. These segments will be mounted onto a backplane with actuators that will allow the segments to be aligned to match one common optical surface that represents a single mirror with a diameter of 6.5m.

A test article, the Backplane Stability Test Article (BSTA), was manufactured using the same materials, techniques, and design principles being developed for constructing the entire telescope. Extensive thermal-vacuum chamber testing was conducted on the BSTA to verify that it will meet the stringent requirements necessary for JWST to work; specifically, that it will remain stable to within 38nm over the orbital operational temperature range of 30-60K. These tests required the development of specialized software to collect and process all the necessary data. Such software was written in Python and is described in the following sections.

Fig. 1: *An Ariane 5 launch similar to the launcher that will be used to place JWST into orbit about the L2 point, the orbital position that keeps the Sun behind the Earth all the time as viewed from JWST. Photo: ESA.*

Testing the BSTA

NASA required a test that demonstrated the stability of this engineering article and which verified that the design and construction

* *Corresponding author: hack@stsci.edu*
‡ *Space Telescope Science Institute*

Fig. 2: Engineers move the Backplane Stability Test Article (BSTA) into position for the thermal vacuum test. This test piece represents a section of the backplane that will support only 3 of the 18 mirror segments of the primary mirror for JWST.

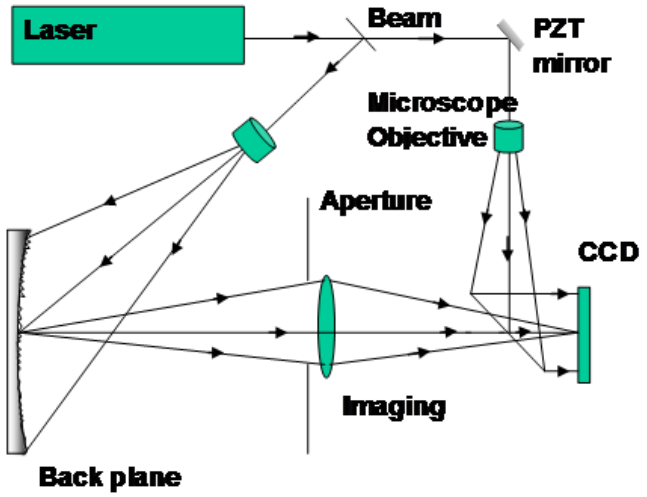


Fig. 3: Schematic of ESPI showing how the ESPI measures the change in the object due to thermal or mechanical stress by tracking the speckles' phase change on the surface.

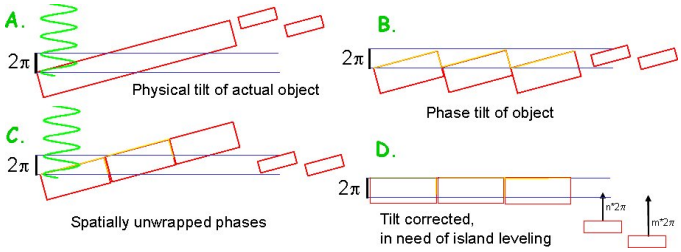


Fig. 4: Schematic showing how bulk tilt introduces multiple 2π variations across the structure and how it gets corrected in processing, allowing for relative variations to be measured across the surface as described in "Testing the BSTA".

techniques will work to meet the requirements of the telescope: namely, that it will remain stable to within 68nm from 30-50K (-405 to -370 °F). They also wished to track the structural changes from ambient (~295K) to cryogenic temperatures (~30K) in order to better understand the accuracy of their structural models. The primary test equipment, a special interferometer, did have software to take measurements, but that software was not suited for the needs of this test. This required the development of specialized software to support the test.

Normal interferometry is used with optical elements where the reflected or transmitted laser signal remains spatially coherent over large areas. Speckle interferometry is intended to be used with non- optical surfaces, that is, surfaces that are optically rough on very small spatial scales. When illuminated by a laser, such surfaces typically show "speckles" that result from random points where the reflections from the surface are relatively coherent (as compared to the darker areas where the reflections mostly cancel out through interference). While the phase of speckles varies randomly from one spot on the article to the next, and thus cannot be used for any spatial comparison from a single image, how the phase for a specific speckle changes, does indicate how the surface is moving relative to the laser; in this way speckle interferometers can be used to determine how surfaces are changing in time. The BSTA, although intended to hold JWST mirror segments, has no optical surfaces of its own. In order to understand how the structure changed with temperature it was necessary to use the Electronic Speckle Pattern Interferometer (ESPI).

The ESPI laser illuminates the surface of the BSTA, then recombines the reflected signal with a reference beam split from the same laser pulse, to create an interferometric map of the surface speckles. As the structure moves, the speckles change phase reflecting the change in interference between the incident and reference laser pulses. However, those phases cycle from $-\pi$ to π and back as the surface continues to move. This required the use of phase unwrapping across the surface, spatial phase unwrapping, using an algorithm developed by the manufacturers of the ESPI.

As the surface tilted during the test, it produced fringes where the phases across the structure would transition from π to $-\pi$. This tilt needed to be removed in order to allow us to measure the relative changes from one region of the structure to another.

Since the measured phase is ambiguous by multiples of 2π, special techniques are required to remove these ambiguities. One is to presume that continuous surfaces have continuous phase, and any discontinuities on continuous surfaces are due to phase wrapping. Thus such discontinuities can be "unwrapped" to produce spatially continuous phase variations. Another presumption is that even though the general position and tilt of the entire structure may change greatly from one exposure to another, the relative phase shape of the structure will not change rapidly in time once bulk tilt and displacement are removed.

The following figures show the consequent phase wraps when a surface has any significant tilt. One can perform spatial phase unwrapping on spatially contiguous sections. Gross tilts are fit to the largest contiguous sections, and then the average tilt is removed (as well as the average displacement). However, there are areas of interest (the mirror pad supports) which are discontiguous and as a result possibly several factors of 2π offset in reality as a result of the tilt, and thus improperly corrected when tilts are removed. Since these areas are assumed to change slowly in time, temporal phase unwrapping is applied to these areas.

The entire ESPI system, hardware and software, was built by 4D Technologies under the guidance of one of our team members,

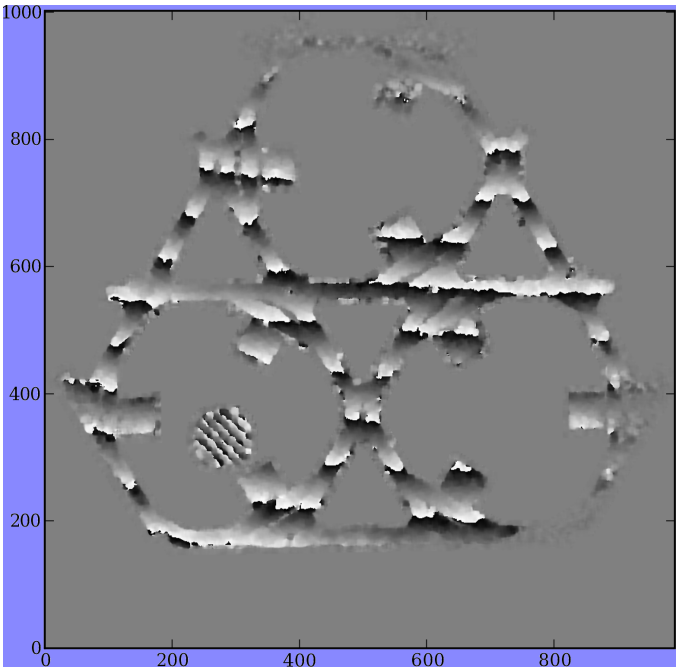


Fig. 5: A sample ESPI image illustrating the fringes that build up due to bulk tilts. These fringes get "unwrapped" to produce spatially contiguous phase variations across the surface of the object.

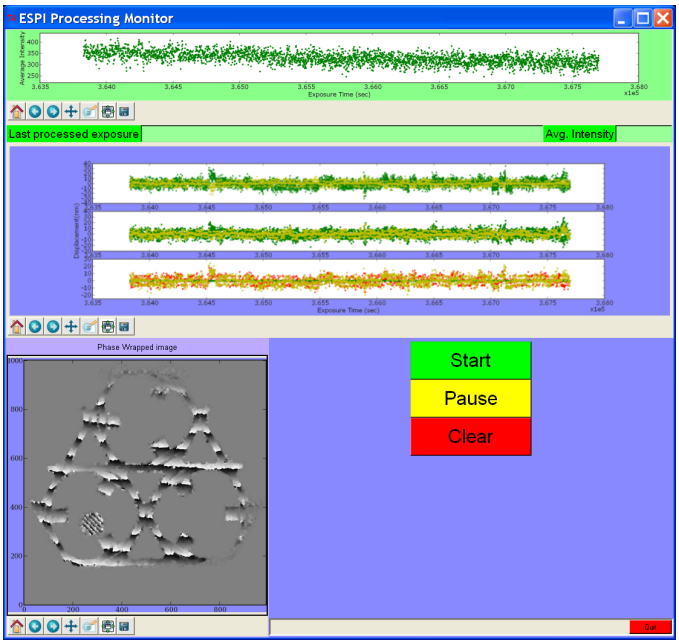


Fig. 6: This snapshot of the ESPI Monitoring GUI in operation illustrates the near-real-time monitoring plots and image display used to track the health of the laser and quality of the data and subsequent processing.

Babak. The commercial software from 4D Technologies that came with the ESPI hardware had algorithms for performing the spatial unwrapping using a GUI interface for interactive operation. This interface, though, was unable to support the needs of the test; namely, that it would need to continuously take 5 images/second for 24 hours/day for up to 6 weeks at a time. Thus, we needed to write our own specialized software to support the test.

Python to the Rescue

Many of the requirements for any software that needed to be written were unknowable, not just unknown, for a number of reasons. No test had ever been conducted like this before, so there was no experience to draw upon to foresee what problems may arise during the test. Concerns ranged from whether the laser output could be maintained at a stable level over such a long period of time given that the output was dependent on the ambient temperature of the test facility. This drove the requirement to monitor in near-real-time the laser intensity as measured from the observations themselves. These results were compared with occasional checks of the laser output using burn paper in the laser path, creating a bit of smoke in the process, to insure that the monitoring was accurately tracking the health of the laser.

We also had no certainty about what phase-unwrapping algorithms were going to work until the test actually started. Test conditions such as residual vibrations in the test rig could seriously impact our ability to measure the surface changes we were after and potentially require changes to how the phase-unwrapping algorithms needed to be applied. It was only after the test started that these effects would be known, requiring the ability to update the data acquisition and processing code on the fly to accommodate the quality of the test data.

Finally, the code had to be easily adaptable and capable of handling massive amounts of data in as close to real time as possible! Python offered the best possible choice for addressing these challenges in supporting this test. It allowed us to develop code rapidly to adjust for the test conditions during the test with minimal impact. The plotting and array-handling libraries, specifically matplotlib and numpy, proved robust and fast enough to keep up with the near-real-time operations. The commercial software that came with ESPI hardware had also been written in Python and C, so Python allowed us to interface to that code to run our own custom processing code using the commercial algorithms for data acquisition and phase-unwrapping.

Our data acquisition system used custom code to automate the operation of the commercial software used to interface with the ESPI camera. This module was run under the commercial software's own Python environment in order to most easily access their camera's API and stored the images in real time on a storage server. The remainder of the processing required the use of the Python API to the commercial software's functions to perform the phase unwrapping. As a result of this extended processing, the remainder of the code could only process and monitor the results of every 5th image taken during the test. This monitoring was performed using a custom Tkinter GUI which provided plots of a couple of key processing results, and an image display of the latest processed image, all using matplotlib.

This data processing pipeline was set up using 4 PCs and a 15Tb storage server. A separate PC was dedicated to each of the processing steps; namely, data acquisition, initial phase unwrapping, measuring of regions, and monitoring of the processing. This distributed system was required in order to support the data acquisition rate for the test: 5 1004x996 pixel images per second for 24 hours a day for 6 uninterrupted weeks. A total of approximately 11Tb of raw data was eventually acquired during the test. These raw observations were later reprocessed several times using the original set of 4 PCs from the test as well as additional PCs all running simultaneously to refine the results in much less than real time using all the lessons learned while the test

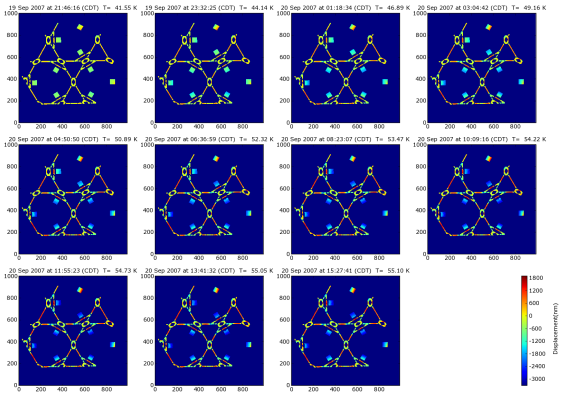


Fig. 7: *Mosaic of sample processed measurements of the BSTA as the temperature changed from 40K to 60K, matching the operational temperature range of JWST. This mosaic illustrates how the structure was measured to change as the temperature changed.*

was in progress. This reprocessing effort represented the simplest possible case of parallel processing, where separate sets of data could be processed independently on separate systems. No other use of parallel processing techniques was implemented for the test or subsequent reprocessing.

Results

BSTA data analysis measured the slope of the data, expansion due to temperature, with an RMS of 25.2nm/K, well within the 36.8nm/K requirement for meeting NASA's goals. These measurements were based on calibrations which had RMS values less than 5 nm around the measured slope.

Python allowed for rapid development of a near-real-time processing pipeline spread across multiple systems which we were able to revise quickly as needed during the test. The fact that the commercial software was written using Python also allowed us to interface with it to use their C-based algorithms for data acquisition and phase-unwrapping. Equally importantly, we were able to implement changes in the processing algorithms while the test was underway to address aspects of the data quality that were not expected when the test began. This software, though, can not be distributed as it was designed explicitly to support the JWST tests alone. The success of this test, though, resulted in establishing the ESPI as a resource for later tests, and this software will be used as the framework for supporting additional tests of JWST in the coming years.

Future Tests

The development of the software for the ESPI tests validated its utility to measure the shape of structures to nanometer accuracies. Additional testing of the actual structure built for use in supporting all 18 segments of the primary mirror for JWST will require this level of accuracy, albeit under very different testing conditions. A new test to map the actual positions and orientations of each of the mirror segments will use an upgraded version of the ESPI to monitor the mirror segments after they have been mounted on the backplane of the telescope. This test will validate that the actuators controlling the position of each mirror segment can be controlled sufficiently to align all the segments to create a single optical surface.

This test will require adjusting the mirror positions, then taking up to a thousand images a second for a short period of time to verify the newly updated positions. Such a test can easily generate 10Tb of imaging data in only 7 minutes. The Python software we developed for previous ESPI tests will be used as the basis for the data acquisition and data processing systems for this new test, including synthesizing data from additional measuring devices. The only way to keep up with this test will be to use multiple systems processing data in parallel to process the data quickly enough to allow the test to proceed as needed, much as we did with the reprocessing of the original ESPI data. In short, Python's rapid development capabilities, fast array handling, and ability to run the same code on multiple systems in parallel will be critical to the success of this new test.

Modeling Sudoku Puzzles with Python

Sean Davis‡*, Matthew Henderson‡, Andrew Smith‡

Abstract—The popular Sudoku puzzles which appear daily in newspapers the world over have, lately, attracted the attention of mathematicians and computer scientists. There are many, difficult, unsolved problems about Sudoku puzzles and their generalizations which make them especially interesting to mathematicians. Also, as is well-known, the generalization of the Sudoku puzzle to larger dimension is an NP-complete problem and therefore of substantial interest to computer scientists.

In this article we discuss the modeling of Sudoku puzzles in a variety of different mathematical domains. We show how to use existing third-party Python libraries to implement these models. Those implementations, which include translations into the domains of constraint satisfaction, integer programming, polynomial calculus and graph theory, are available in an open-source Python library `sudoku.py` developed by the authors and available at http://bitbucket.org/matthew/scipy2010

Index Terms—sudoku, mathematics, graph theory

Introduction

Sudoku puzzles

A Sudoku puzzle is shown near the top of the second column on this page.

To complete this puzzle requires the puzzler to fill every empty cell with an integer between 1 and 9 in such a way that every number from 1 up to 9 appears once in every row, every column and every one of the small 3 by 3 boxes highlighted with thick borders.

Sudoku puzzles vary widely in difficulty. Determining the hardness of Sudoku puzzles is a challenging research problem for computational scientists. Harder puzzles typically have fewer prescribed symbols. However, the number of prescribed cells is not alone responsible for the difficulty of a puzzle and it is not well-understood what makes a particular Sudoku puzzle hard, either for a human or for an algorithm to solve.

The Sudoku puzzles which are published for entertainment invariably have unique solutions. A Sudoku puzzle is said to be *well-formed* if it has a unique solution. Another challenging research problem is to determine how few cells need to be filled for a Sudoku puzzle to be well-formed. Well-formed Sudoku with 17 symbols exist. It is unknown whether or not there exists a well-formed puzzle with only 16 clues. In this paper we consider all Sudoku puzzles, as defined in the next paragraph, not only the well-formed ones.

* *Corresponding author: Sean_Davis@berea.edu*
‡ *Berea College*

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

By *Sudoku puzzle of boxsize n*, in this paper, is meant a partial assignment of values from $\{1,\ldots,n^2\}$ to the cells of an $n^2 \times n^2$ grid in such a way that at most one of each symbols occurs in any row, column or box. A *solution* of a Sudoku puzzle is a complete assignment to the cells, satisfying the same conditions on row, columns and boxes, which extends the original partial assignment.

sudoku.py

With `sudoku.py`, the process of building models of Sudoku puzzles, which can then be solved using algorithms for computing solutions of the models, is a simple matter. In order to understand how to build the models, first it is necessary to explain the two different representations of Sudoku puzzles in `sudoku.py`.

The dictionary representation of a puzzle is a mapping between cell labels and cell values. Cell values are integers in the range $\{1,\ldots,n^2\}$ and cell labels are integers in the range $\{1,\ldots,n^4\}$. The labeling of a Sudoku puzzle of boxsize n starts with 1 in the top-left corner and moves along rows, continuing to the next row when a row is finished. So, the cell in row i and column j is labeled $(i-1)n^2+j$.

For example, the puzzle from the introduction can be represented by the dictionary

```
>>> d = {1: 2, 2: 5, 5: 3, 7: 9, 9: 1,
```

```
...      11: 1, 15: 4, 19: 4, 21: 7, 25: 2,
...      27: 8, 30: 5, 31: 2, 41: 9, 42: 8,
...      43: 1, 47: 4, 51: 3, 58: 3, 59: 6,
...      62: 7, 63: 2, 65: 7, 72: 3, 73: 9,
...      75: 3, 79: 6, 81: 4}
```

A Sudoku puzzle object can be built from such a dictionary. Note that the boxsize is a parameter of the `Puzzle` object constructor.

```
>>> from sudoku import Puzzle
>>> p = Puzzle(d, 3)
>>> p
 2  5  .  .  3  .  9  .  1
 .  1  .  .  .  4  .  .  .
 4  .  7  .  .  .  2  .  8
 .  .  5  2  .  .  .  .  .
 .  .  .  .  9  8  1  .  .
 .  4  .  .  .  3  .  .  .
 .  .  .  3  6  .  .  7  2
 .  7  .  .  .  .  .  .  3
 9  .  3  .  .  .  6  .  4
```

In practice, however, the user mainly interacts with `sudoku.py` either by creating specific puzzles instances through input of puzzle strings, directly or from a text file, or by using generator functions.

The string representation of a Sudoku puzzle of boxsize n is a string of ascii characters of length n^4. In such a string a period character represents an empty cell and other ascii characters are used to specify assigned values. Whitespace characters and newlines are ignored when `Puzzle` objects are built from strings.

A possible string representation of the puzzle from the introduction is:

```
>>> s = """
... 2 5 . . 3 . 9 . 1
... . 1 . . . 4 . . .
... 4 . 7 . . . 2 . 8
... . . 5 2 . . . . .
... . . . . 9 8 1 . .
... . 4 . . . 3 . . .
... . . . 3 6 . . 7 2
... . 7 . . . . . . 3
... 9 . 3 . . . 6 . 4
    """
```

A `Puzzle` object can be built from a puzzle string by providing the keyword argument `format = 's'`

```
>>> p = Puzzle(s, 3, format = 's')
```

Random puzzles can be created in `sudoku.py` by the `random_puzzle` function.

```
>>> from sudoku import random_puzzle
>>> q = random_puzzle(15, 3)
>>> q
 .  .  .  .  5  .  .  .  1
 .  5  .  .  .  .  .  .  7
 .  .  1  9  .  7  .  .  .
 .  .  .  .  .  .  .  .  .
 .  .  5  .  .  .  7  .  .
 .  .  6  .  .  .  .  9  .
 .  .  .  .  .  5  .  .  .
 5  .  .  .  .  .  4  .  .
 1  .  .  .  .  .  .  .  .
```

The first argument to `random_puzzle` is the number of prescribed cells in the puzzle.

Solving of puzzles in `sudoku.py` is handled by the `solve` function. This function can use a variety of different algorithms, specified by an optional `model` keyword argument, to solve the puzzle. Possible values are `CP` for constraint propagation, `lp` for linear programming, `graph` to use a node coloring algorithm on a graph puzzle model and `groebner` to solve a polynomial system model via a Groebner basis algorithm. The default behavior is to use constraint propagation.

```
>>> from sudoku import solve
>>> s = solve(q)
>>> s
 7  3  2  8  5  6  9  4  1
 8  5  9  4  2  1  6  3  7
 6  4  1  9  3  7  8  5  2
 9  7  8  5  4  3  1  2  6
 3  2  5  6  1  9  7  8  4
 4  1  6  7  8  2  5  9  3
 2  9  4  1  6  5  3  7  8
 5  6  3  2  7  8  4  1  9
 1  8  7  3  9  4  2  6  5
```

Sudoku puzzles of boxsize other than 3 can also be modeled with `sudoku.py`. Puzzles of boxsize 2 are often called Shidoku.

```
>>> q2 = random_puzzle(7, 2)
>>> q2
 4  .  .  .
 2  1  .  .
 .  4  .  2
 .  .  3  4
>>> solve(q2)
 4  3  2  1
 2  1  4  3
 3  4  1  2
 1  2  3  4
```

Sudoku puzzles of boxsize greater than three are less commonly studied in the literature. In `sudoku.py` we use printable characters (from `string.printable`) for the symbols of puzzles with boxsize greater than 3

```
>>> q4 = random_puzzle(200, 4)
>>> q4
 .  .  e  d  .  .  a  9  8  .  .  5  .  3  2  1
 c  b  a  9  4  .  2  1  g  .  e  d  8  7  6  .
 8  .  6  5  g  f  e  d  4  3  2  1  c  b  a  9
 .  .  2  1  8  7  6  5  c  .  a  .  g  f  e  d
 f  d  g  .  9  8  7  c  3  6  .  b  .  2  .  .
 2  6  .  .  1  d  g  b  f  4  c  .  9  .  8  7
 .  4  1  8  3  6  .  2  9  e  7  .  .  .  5  c
 9  c  7  b  e  a  5  .  2  1  .  8  f  g  3  6
 e  g  9  f  7  .  8  a  6  d  3  4  5  1  b  .
 b  a  .  7  .  2  9  e  5  .  1  f  .  8  c  .
 3  8  .  6  5  1  4  f  .  9  b  2  7  a  d  g
 .  .  4  .  d  g  b  3  7  a  8  c  e  6  9  f
 .  e  f  c  2  9  3  8  a  5  g  7  6  4  .  b
 7  9  .  4  a  .  1  6  d  8  .  e  2  c  g  3
 6  2  8  g  b  .  d  .  .  c  9  3  .  .  f  .
 5  1  3  a  f  e  c  g  b  2  4  6  .  .  7  8
```

Solving puzzles of this size is still feasible by constraint propogation

```
>>> solve(q4)
g  f  e  d  c  b  a  9  8  7  6  5  4  3  2  1
c  b  a  9  4  3  2  1  g  f  e  d  8  7  6  5
8  7  6  5  g  f  e  d  4  3  2  1  c  b  a  9
4  3  2  1  8  7  6  5  c  b  a  9  g  f  e  d
f  d  g  e  9  8  7  c  3  6  5  b  1  2  4  a
2  6  5  3  1  d  g  b  f  4  c  a  9  e  8  7
a  4  1  8  3  6  f  2  9  e  7  g  b  d  5  c
9  c  7  b  e  a  5  4  2  1  d  8  f  g  3  6
e  g  9  f  7  c  8  a  6  d  3  4  5  1  b  2
b  a  d  7  6  2  9  e  5  g  1  f  3  8  c  4
3  8  c  6  5  1  4  f  e  9  b  2  7  a  d  g
1  5  4  2  d  g  b  3  7  a  8  c  e  6  9  f
d  e  f  c  2  9  3  8  a  5  g  7  6  4  1  b
7  9  b  4  a  5  1  6  d  8  f  e  2  c  g  3
6  2  8  g  b  4  d  7  1  c  9  3  a  5  f  e
5  1  3  a  f  e  c  g  b  2  4  6  d  9  7  8
```

Models

In this section we introduce several models of Sudoku and show how to use existing Python components to implement these models. The models introduced here are all implemented in `sudoku.py`. Implementation details are discussed in this section and demonstrations of the components of `sudoku.py` corresponding to each of the different models are given.

Constraint models

Constraint models for Sudoku puzzles are discussed in [Sim05]. A simple model uses the AllDifferent constraint.

A constraint program is a collection of constraints. A constraint restricts the values which can be assigned to certain variables in a solution of the constraint problem. The AllDifferent constraint restricts variables to having mutually different values.

Modeling Sudoku puzzles is easy with the AllDifferent constraint. To model the empty Sudoku puzzle (i.e. the puzzle with no clues) a constraint program having an AllDifferent constraint for every row, column and box is sufficient.

For example, if we let $x_i \in \{1,\ldots,n^2\}$ for $1 \leq i \leq n^4$, where $x_i = j$ means that cell i gets value j then the constraint model for a Sudoku puzzle of boxsize $n = 3$ would include constraints:

$$\text{AllDifferent}(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9)$$

$$\text{AllDifferent}(x_1,x_{10},x_{19},x_{28},x_{37},x_{46},x_{55},x_{64},x_{73})$$

$$\text{AllDifferent}(x_1,x_2,x_3,x_{10},x_{11},x_{12},x_{19},x_{20},x_{21})$$

These constraints ensure that, respectively, the variables in the first row, column and box get different values.

The Sudoku constraint model in `sudoku.py` is implemented using `python-constraint v1.1` by Gustavo Niemeyer. This open-source library is available at http://labix.org/python-constraint.

With `python-constraint` a `Problem` having variables for every cell $\{1,\ldots,n^4\}$ of the Sudoku puzzle is required. The list of cell labels is given by the function `cells` in `sudoku.py`. Every variable has the same domain $\{1,\ldots,n^2\}$ of symbols. The list of symbols in `sudoku.py` is given by the `symbols` function.

The `Problem` member function `addVariables` provides a convenient method for adding variables to a constraint problem object.

```
>>> from constraint import Problem
>>> from sudoku import cells, symbols
>>> cp = Problem()
>>> cp.addVariables(cells(n), symbols(n))
```

The AllDifferent constraint in `python-constraint` is implemented as `AllDifferentConstraint()`. The `addConstraint(constraint, variables)` member function is used to add a `constraint` on `variables` to a constraint `Problem` object. So, to build an empty Sudoku puzzle constraint model we can do the following.

```
>>> from constraint import AllDifferentConstraint
>>> from sudoku import \
...   cells_by_row, cells_by_col, cells_by_box
>>> for row in cells_by_row(n):
...   cp.addConstraint(AllDifferentConstraint(), row)
>>> for col in cells_by_col(n):
...   cp.addConstraint(AllDifferentConstraint(), col)
>>> for box in cells_by_box(n):
...   cp.addConstraint(AllDifferentConstraint(), box)
```

Here the functions `cells_by_row`, `cells_by_col` and `cells_by_box` give the cell labels of a Sudoku puzzle ordered, respectively, by row, column and box. These three loops, respectively, add to the constraint problem object the necessary constraints on row, column and box variables.

To extend this model to a Sudoku puzzle with clues requires additional constraints to ensure that the values assigned to clue variables are fixed. One possibility is to use an ExactSum constraint for each clue.

The ExactSum constraint restricts the sum of a set of variables to a precise given value. We can slightly abuse the ExactSum constraint to specify that certain individual variables are given certain specific values. In particular, if the puzzle clues are given by a dictionary `d` then we can complete our model by adding the following constraints.

```
>>> from constraint import ExactSumConstraint as Exact
>>> for cell in d:
...   cp.addConstraint(Exact(d[cell]), [cell])
```

To solve the Sudoku puzzle now can be done by solving the constraint model `cp`. The constraint propogation algorithm of `python-constraint` can be invoked by the `getSolution` member function.

```
>>> s = Puzzle(cp.getSolution(), 3)
>>> s
 2  5  8  7  3  6  9  4  1
 6  1  9  8  2  4  3  5  7
 4  3  7  9  1  5  2  6  8
 3  9  5  2  7  1  4  8  6
 7  6  2  4  9  8  1  3  5
 8  4  1  6  5  3  7  2  9
 1  8  4  3  6  9  5  7  2
 5  7  6  1  4  2  8  9  3
 9  2  3  5  8  7  6  1  4
```

The general `solve` function of `sudoku.py` knows how to build the constraint model above, find a solution via the propogation algorithm of `python-constraint` and translate the solution into a completed Sudoku puzzle.

```
>>> s = solve(p, model = 'CP')
```

Here, `p` is a `Puzzle` instance. In fact, the `model = 'CP'` keyword argument in this case is redundant, as `'CP'` is the default value of `model`.

Graph models

A graph model for Sudoku is presented in [Var05]. In this model, every cell of the Sudoku grid is represented by a node of the graph. The edges of the graph are given by the dependency relationships between cells. In other words, if two cells lie in the same row, column or box, then their nodes are joined by an edge in the graph.

In the graph model, a Sudoku puzzle is given by a partial assignment of colors to the nodes of the graph. The color assigned to a node corresponds to a value assigned to the corresponding cell. A solution of the puzzle is given by a coloring of the nodes with colors $\{1,\ldots,n^2\}$ which extends the original partial coloring. A node coloring of the Sudoku graph which corresponds to a completed puzzle has the property that adjacent vertices are colored differently. Such a node coloring is called *proper*.

The Sudoku graph model in `sudoku.py` is implemented using `networkx v1.1`. This open-source Python graph library is available at http://networkx.lanl.gov/

Modeling an empty Sudoku puzzle as a `networkx.Graph` object requires nodes for every cell and edges for every pair of dependent cells. To add nodes (respectively, edges) to a graph, `networkx` provides member functions `add_nodes_from` (respectively, `add_edges_from`). Cell labels are obtained from `sudoku.py`'s `cells` function.

```
>>> import networkx
>>> g = networkx.Graph()
>>> g.add_nodes_from(cells(n))
```

Dependent cells are computed using the `dependent_cells` function. This function returns the list of all pairs (x,y) with $x < y$ such that x and y either lie in the same row, same column or same box.

```
>>> from sudoku import dependent_cells
>>> g.add_edges_from(dependent_cells(n))
```

To model a Sudoku puzzle, we have to be able to assign colors to nodes. Graphs in `networkx` allow arbitrary data to be associated with graph nodes. To color nodes according to the dictionary `d` of puzzle clues.

```
>>> for cell in d:
...   g.node[cell]['color'] = d[cell]
```

There are many node coloring algorithms which can be used to find a solution of a puzzle. In `sudoku.py`, a generic node coloring algorithm is implemented. This generic coloring algorithm can be customized to provide a variety of different specific coloring algorithms. However, none of these algorithms is guaranteed to find a soolution which uses only symbols from $\{1,\ldots,n^2\}$. In general, these algorithms use too many colors

```
>>> from sudoku import node_coloring, n_colors
>>> cg = node_coloring(g)
>>> n_colors(cg)
13
>>> from sudoku import graph_to_dict
>>> s = Puzzle(graph_to_dict(cg), 3)
>>> s
 2  5  6  7  3  a  9  4  1
 3  1  8  5  2  4  7  6  a
 4  9  7  6  b  c  2  3  8
 6  3  5  2  4  7  8  9  b
 7  2  a  b  9  8  1  5  6
 8  4  9  a  5  3  c  2  7
 5  8  4  3  6  9  a  7  2
 a  7  b  4  8  5  d  c  3
 9  c  3  d  7  b  6  8  4
```

To solve a Sudoku `Puzzle` instance `p`, call the `solve` function, with `model = graph` as a keyword argument.

```
>>> s = solve(p, model = 'graph')
```

Polynomial system models

The graph model above is introduced in [Var05] as a prelude to modeling Sudoku puzzles as systems of polynomial equations. The polynomial system model in [Var05] involves variables x_i for $i \in \{1,\ldots,n^4\}$ where $x_i = j$ is interpreted as the cell with label i being assigned the value j.

The Sudoku polynomial-system model in sudoku.py is implemented using `sympy v0.6.7`. This open-source symbolic algebra Python library is available at http://code.google.com/p/sympy/

Variables in `sympy` are `Symbol` objects. A `sympy.Symbol` object has a name. So, to construct the variables for our model, first we map symbol names onto each cell label.

```
>>> from sudoku import cell_symbol_name
>>> def cell_symbol_names(n):
...   return map(cell_symbol_name, cells(n))
```

Now, with these names for the symbols which represent the cells of our Sudoku puzzle, we can construct the cell variable symbols themselves.

```
>>> from sympy import Symbol
>>> def cell_symbols(n):
...   return map(Symbol, cell_symbol_names(n))
```

Finally, with these variables, we can build a Sudoku polynomial system model. This model is based on the graph model of the previous section. There are polynomials in the system for every node in the graph model and polynomials for every edge.

The role of node polynomial $F(x_i)$ is to ensure that every cell i is assigned a number from $\{1,\ldots,n^2\}$:

$$F(x_i) = \prod_{j=1}^{n^2}(x_i - j)$$

Node polynomials, for a `sympy.Symbol` object `x` are built as follows.

```
>>> from operator import mul
>>> from sudoku import symbols
>>> def F(x,n):
...   return reduce(mul,[(x-s) for s in symbols(n)])
```

The edge polynomial $G(x_i,x_j)$ for dependent cells i and j, ensures that cells i and j are assigned different values. These polynomials have the form. :

$$G(x_i,x_j) = \frac{F(x_i) - F(x_j)}{x_i - x_j}$$

In `sympy`, we build edge polynomials from the node polynomial function `F`.

```
>>> from sympy import cancel, expand
>>> def G(x,y,n):
...   return expand(cancel((F(x,n)-F(y,n))/(x-y)))
```

The polynomial model for the empty Sudoku puzzle consists of the collection of all node polynomials for nodes in the Sudoku graph and all edge polynomials for pairs `(x,y)` in `dependent_symbols(n)`. The `dependent_symbols` function is simply a mapping of the `sympy.Symbol` constructor onto the list of dependent cells.

Specifying a Sudoku puzzle requires extending this model by adding polynomials to represent clues. According to the model from [Var05], if D is the set of fixed cells (i.e. cell label, value pairs) then to the polynomial system we need to add polynomials

$$D(x_i, j) = x_i - j$$

Or, with `sympy`:

```
>>> def D(i, j):
...   return Symbol(cell_symbol_name(i)) - j
```

To build the complete polynomial system, use the `puzzle_as_polynomial_system` function of `sudoku.py`:

```
>>> from sudoku import puzzle_as_polynomial_system
>>> g = puzzle_as_polynomial_system(d, 3)
```

The `sympy` implementation of a Groebner basis algorithm can be used to find solutions of this polynomial system. The Groebner basis depends upon a variable ordering, here specified as lexicographic. Other orderings, such as degree-lexicographic, are possible.

```
>>> from sympy import groebner
>>> h = groebner(g, cell_symbols(n), order = 'lex')
```

The solution of the polynomial system `g` is a system of linear equations in the symbols x_i which can be solved by the linear solver from `sympy`.

```
>>> from sympy import solve as lsolve
>>> s = lsolve(h, cell_symbols(n))
```

To use the polynomial-system model to find a solution to `Puzzle` instance `p` call the solve function with the keyword argument `model = groebner`.

```
>>> s = solve(p, model = 'groebner')
```

Integer programming models

In [Bar08] a model of Sudoku as an integer programming problem is presented. In this model, the variables are all binary.

$$x_{ijk} \in \{0,1\}$$

Variable x_{ijk} represents the assignment of symbol k to cell (i,j) in the Sudoku puzzle.

$$x_{ijk} = \begin{cases} 1 & \text{if cell } (i,j) \text{ contains symbol } k \\ 0 & \text{otherwise} \end{cases}$$

The integer programming (IP) model has a set of equations which force the assignment of a symbol to every cell.

$$\sum_{k=1}^{n} x_{ijk} = 1, \quad 1 \le i \le n, 1 \le j \le n$$

Other equations in the IP model represent the unique occurence of every symbol in every column:

$$\sum_{i=1}^{n} x_{ijk} = 1, \quad 1 \le j \le n, 1 \le k \le n$$

every symbol in every row:

$$\sum_{j=1}^{n} x_{ijk} = 1, \quad 1 \le i \le n, 1 \le k \le n$$

and every symbol in every box:

$$\sum_{j=mq-m+q}^{mq} \sum_{i=mp-m+1}^{mp} x_{ijk} = 1$$

$$1 \le k \le n, 1 \le p \le m, 1 \le q \le m$$

The Sudoku IP model is implemented in `sudoku.py` using `pyglpk v0.3` by Thomas Finley. This open-source mixed integer/linear programming Python library is available at http://tfinley.net/software/pyglpk/

In `pyglpk`, an integer program is represented by the matrix of coefficients of a system of linear equations. Two functions of `sudoku.py` provide the correct dimensions of the coefficient matrix.

```
>>> from glpk import LPX
>>> from sudoku import \
...   lp_matrix_ncols, lp_matrix_nrows
>>> lp = LPX()
>>> lp.cols.add(lp_matrix_ncols(n))
>>> lp.rows.add(lp_matrix_nrows(n))
```

Columns of the matrix represent different variables. All our variables are binary and so their bounds are set appropriately, between 0 and 1.

```
>>> for c in lp.cols:
...   c.bounds = 0.0, 1.0
```

Rows of the coefficient matrix represent different linear equations. We require all our equations to have a value of 1, so we set both the lower and upper bound of every equation to be 1.

```
>>> for r in lp.rows:
...   r.bounds = 1.0, 1.0
```

With appropriate dimensions and bounds fixed, the coefficient matrix itself is provided by `sudoku.py`'s `lp_matrix` function.

```
>>> from sudoku import lp_matrix
>>> lp.matrix = lp_matrix(n)
```

To extend the IP model to a Sudoku puzzle with fixed clues requires further equations. Fixed elements in the puzzle, given by a set F of triples (i,j,k), are each represented by an equation in the system:

$$x_{ijk} = 1, \quad \forall (i,j,k) \in F$$

To add these equations to the `pyglpk.LPX` object `lp`:

```
>>> from sudoku import lp_col_index
>>> for cell in d:
...   lp.rows.add(1)
...   r = lp_matrix_ncols(n)*[0]
...   r[lp_col_index(cell, d[cell], n)] = 1
...   lp.rows[-1].matrix = r
...   lp.rows[-1].bounds = 1.0, 1.0
```

To solve the `LPX` instance `lp` requires first solving a linear relaxation via the simplex algorithm implementation of `pyglpk`

```
>>> lp.simplex()
```

Once the linear relaxation is solved, the original integer program can be solved.

```
>>> for col in lp.cols:
...   col.kind = int
>>> lp.integer()
```

Finally, we need to extract the solution as a dictionary from the model via the `lp_to_dict` function from `sudoku.py`.

```
>>> from sudoku import lp_to_dict
>>> d = lp_to_dict(lp, n)
>>> s = Puzzle(d, 3)
>>> s
 2  5  8  7  3  6  9  4  1
 6  1  9  8  2  4  3  5  7
 4  3  7  9  1  5  2  6  8
 3  9  5  2  7  1  4  8  6
 7  6  2  4  9  8  1  3  5
 8  4  1  6  5  3  7  2  9
 1  8  4  3  6  9  5  7  2
 5  7  6  1  4  2  8  9  3
 9  2  3  5  8  7  6  1  4
```

To use the IP model to solve a `Puzzle` instance, specify the keyword argument `model = lp`.

```
>>> s = solve(p, model = 'lp')
```

Experimentation

In this section we demonstrate the use of `sudoku.py` for creating Python scripts for experimentation with Sudoku puzzles. For the purposes of demonstration, we discuss, briefly, enumeration of Shidoku puzzles, coloring the Sudoku graph and the hardness of random puzzles.

Enumerating Shidoku

Enumeration of Sudoku puzzles is a very difficult computational problem, which has been solved by Felgenhauer and Jarvis in [Fel06]. The enumeration of Shidoku, however, is easy. To solve the enumeration problem for Shidoku, using the constraint model implemented in `sudoku.py`, takes only a few lines of code and a fraction of a second of computation.

```
>>> s = "from sudoku import Puzzle, count_solutions"
>>> e = "print count_solutions(Puzzle({}, 2))"
>>> from timeit import Timer
>>> t = Timer(e, s)
>>> print t.timeit(1)
288
0.146998882294
```

Coloring the Sudoku graph

As discussed above in the section on "Graph models", a completed Sudoku puzzle is equivalent to a minimal proper node coloring of the Sudoku graph. We have experimented with several different node coloring algorithms to see which are more effective, with respect to minimizing the number of colors, at coloring the Sudoku graph.

Initially, we used Joseph Culberson's graph coloring programs (http://webdocs.cs.ualberta.ca/~joe/Coloring/index.html) by writing Sudoku puzzle graphs to a file in Dimacs format (via the `dimacs_string` function of `sudoku.py`).

Of those programs we experimented with, the program implementing the saturation degree algorithm (DSatur) of Brelaz from [Bre79] seemed most effective at minimizing the number of colors.

Motivated to investigate further, with `sudoku.py` we implemented a general node coloring algorithm directly in Python which can reproduce the DSatur algorithm as well as several other node coloring algorithms.

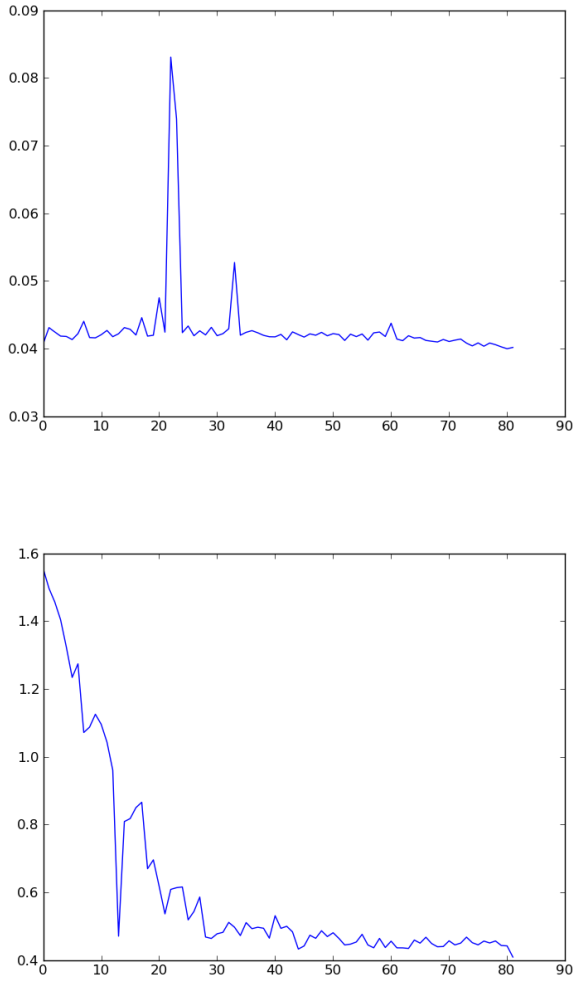
Our node coloring function allows for customization of a quite general scheme. The behavior of the algorithm is specialized by two parameters. The `nodes` parameter is an iterable object giving a node ordering. The `choose_color` parameter is a visitor object which is called every time a node is visited by the algorithm.

Several node orderings and color choice selection schemes have been implemented. The simplest sequential node coloring algorithm can be reproduced, for example, by assigning `nodes = InOrder` and `choose_color = first_available_color`. A random ordering on nodes can be acheived instead by assigning `nodes = RandomOrder`. Importantly for our investigations, the node ordering is given by an iterable object and so, in general, can reflect upon to current graph state. This mean that online algorithms like the DSatur algorithm can be realized by our general node coloring scheme. The DSatur algorithm is obtained by assigning `nodes = DSATOrder` and `choose_color = first_available_color`.

Hardness of random puzzles

We introduced the `random_puzzle` function in the introduction. The method by which this function produces a random puzzle is fairly simple. A completed Sudoku puzzle is first generated by solving the empty puzzle via constraint propagation and then from this completed puzzle the appropriate number of clues is removed.

An interesting problem is to investigate the behavior of different models on random puzzles. A simple script, available in the `investigations` folder of the source code, has been written to time the solution of models of random puzzles and plot the timings via `matplotlib`.



Two plots produced by this script highlight the different behavior of the constraint model and the integer programming model.

The first plot has time on the vertical axis and the number of clues on the horizontal axis. From this plot it seems that the constraint propogation algorithm finds puzzles with many or few clues easy. The difficult problems for the constraint solver appear to be clustered in the range of 20 to 35 clues.

A different picture emerges with the linear programming model. With the same set of randomly generated puzzles it appears that the more clues the faster the solver finds a solution.

Conclusions and future work

In this article we introduced `sudoku.py`, an open-source Python library for modeling Sudoku puzzles. We discussed several models of Sudoku puzzles and demonstrated how to implement these models using existing Python libraries. A few simple experiments involving Sudoku puzzles were presented.

Future plans for `sudoku.py` are to increase the variety of models. Both by allowing for greater customization of currently

implemented models and by implementing new models. For example, we can imagine several different Sudoku models as constraint programs beyond the model presented here. Another approach is to model Sudoku puzzles as exact cover problems and investigate the effectiveness of Knuth's dancing links algorithm. Also important to us is to compare all our models with models [Lyn06] from satisfiability theory. In [Kul10] a general scheme is presented which is highly effective for modeling Sudoku.

There are great many interesting, unsolved scientific problems involing Sudoku puzzles. Our hope is that `sudoku.py` can become a useful tool for scientists who work on these problems.

REFERENCES

[Bar08] A. Bartlett, T. Chartier, A. Langville, T. Rankin. *An Integer Programming Model for the Sudoku Problem*, J. Online Math. & Its Appl., 8(May 2008), May 2008

[Bre79] Brelaz, D., *New methods to color the vertices of a graph*, Communications of the Assoc. of Comput. Machinery 22 (1979), 251-256.

[Fel06] B. Felgenhauer, F. Jarvis. *Enumerating possible Sudoku grids* Online resource 2006 http://www.afjarvis.staff.shef.ac.uk/sudoku/

[Kul10] O. Kullmann, *Green-Tao numbers and SAT* in LNCS (Springer), "Theory and Applications of Satisfiability Testing - SAT 2010", editors O. Strichman and S. Szeider

[Lew05] R. Lewis. *Metaheuristics can solve Sudoku puzzles*, Journal of Heuristics (2007) 13: 387-401

[Lyn06] Lynce, I. and Ouaknine. *Sudoku as a SAT problem*, Proceedings of the 9th Symposium on Artificial Intelligence and Mathematics, 2006.

[Sim05] H. Simonis. *Sudoku as a Constraint Problem*, Proceedings of the 4th International Workshop on Modelling and Reformuulating Constraint Satisfaction Problems. pp.13-27 (2005)

[Var05] J. Gago-Vargas, I. Hartillo-Hermosa, J. Martin-Morales, J. M. Ucha-Enriquez, *Sudokus and Groebner Bases: not only a Divertimento*, In: Lecture Notes in Computer Science, vol. 4194. pp. 155-165. 2005

Data Structures for Statistical Computing in Python

Wes McKinney‡*

Abstract—In this paper we are concerned with the practical issues of working with data sets common to finance, statistics, and other related fields. **pandas** is a new library which aims to facilitate working with these data sets and to provide a set of fundamental building blocks for implementing statistical models. We will discuss specific design issues encountered in the course of developing **pandas** with relevant examples and some comparisons with the R language. We conclude by discussing possible future directions for statistical computing and data analysis using Python.

Index Terms—data structure, statistics, R

Introduction

Python is being used increasingly in scientific applications traditionally dominated by [R], [MATLAB], [Stata], [SAS], other commercial or open-source research environments. The maturity and stability of the fundamental numerical libraries ([NumPy], [SciPy], and others), quality of documentation, and availability of "kitchen-sink" distributions ([EPD], [Pythonxy]) have gone a long way toward making Python accessible and convenient for a broad audience. Additionally [matplotlib] integrated with [IPython] provides an interactive research and development environment with data visualization suitable for most users. However, adoption of Python for applied statistical modeling has been relatively slow compared with other areas of computational science.

A major issue for would-be statistical Python programmers in the past has been the lack of libraries implementing standard models and a cohesive framework for specifying models. However, in recent years there have been significant new developments in econometrics ([StaM]), Bayesian statistics ([PyMC]), and machine learning ([SciL]), among others fields. However, it is still difficult for many statisticians to choose Python over R given the domain-specific nature of the R language and breadth of well-vetted open-source libraries available to R users ([CRAN]). In spite of this obstacle, we believe that the Python language and the libraries and tools currently available can be leveraged to make Python a superior environment for data analysis and statistical computing.

In this paper we are concerned with data structures and tools for working with data sets *in-memory*, as these are fundamental building blocks for constructing statistical models. **pandas** is a new Python library of data structures and statistical tools initially developed for quantitative finance applications. Most of our examples here stem from time series and cross-sectional data arising in financial modeling. The package's name derives from *panel data*, which is a term for 3-dimensional data sets encountered in statistics and econometrics. We hope that **pandas** will help make scientific Python a more attractive and practical statistical computing environment for academic and industry practitioners alike.

* *Corresponding author: wesmckinn@gmail.com*
‡ *AQR Capital Management, LLC*

Statistical data sets

Statistical data sets commonly arrive in tabular format, i.e. as a two-dimensional list of *observations* and names for the fields of each observation. Usually an observation can be uniquely identified by one or more values or *labels*. We show an example data set for a pair of stocks over the course of several days. The NumPy `ndarray` with structured dtype can be used to hold this data:

```
>>> data
array([('GOOG', '2009-12-28', 622.87, 1697900.0),
       ('GOOG', '2009-12-29', 619.40, 1424800.0),
       ('GOOG', '2009-12-30', 622.73, 1465600.0),
       ('GOOG', '2009-12-31', 619.98, 1219800.0),
       ('AAPL', '2009-12-28', 211.61, 23003100.0),
       ('AAPL', '2009-12-29', 209.10, 15868400.0),
       ('AAPL', '2009-12-30', 211.64, 14696800.0),
       ('AAPL', '2009-12-31', 210.73, 12571000.0)],
      dtype=[('item', '|S4'), ('date', '|S10'),
             ('price', '<f8'), ('volume', '<f8')])

>>> data['price']
array([622.87, 619.4, 622.73, 619.98, 211.61, 209.1,
       211.64, 210.73])
```

Structured (or record) arrays such as this can be effective in many applications, but in our experience they do not provide the same level of flexibility and ease of use as other statistical environments. One major issue is that they do not integrate well with the rest of NumPy, which is mainly intended for working with arrays of homogeneous dtype.

R provides the `data.frame` class which can similarly store mixed-type data. The core R language and its 3rd-party libraries were built with the `data.frame` object in mind, so most operations on such a data set are very natural. A `data.frame` is also flexible in size, an important feature when assembling a collection of data. The following code fragment loads the data stored in the CSV file `data` into the variable `df` and adds a new column of boolean values:

```
> df <- read.csv('data')
  item       date  price   volume
1 GOOG 2009-12-28 622.87  1697900
2 GOOG 2009-12-29 619.40  1424800
3 GOOG 2009-12-30 622.73  1465600
4 GOOG 2009-12-31 619.98  1219800
5 AAPL 2009-12-28 211.61 23003100
```

```
6 AAPL 2009-12-29 209.10 15868400
7 AAPL 2009-12-30 211.64 14696800
8 AAPL 2009-12-31 210.73 12571000

> df$ind <- df$item == "GOOG"
> df
  item       date  value   volume   ind
1 GOOG 2009-12-28 622.87  1697900  TRUE
2 GOOG 2009-12-29 619.40  1424800  TRUE
3 GOOG 2009-12-30 622.73  1465600  TRUE
4 GOOG 2009-12-31 619.98  1219800  TRUE
5 AAPL 2009-12-28 211.61 23003100 FALSE
6 AAPL 2009-12-29 209.10 15868400 FALSE
7 AAPL 2009-12-30 211.64 14696800 FALSE
8 AAPL 2009-12-31 210.73 12571000 FALSE
```

pandas provides a similarly-named `DataFrame` class which implements much of the functionality of its R counterpart, though with some important enhancements (namely, built-in data alignment) which we will discuss. Here we load the same CSV file as above into a `DataFrame` object using the `fromcsv` function and similarly add the above column:

```
>>> data = DataFrame.fromcsv('data', index_col=None)
     date           item     value      volume
0    2009-12-28     GOOG     622.9      1.698e+06
1    2009-12-29     GOOG     619.4      1.425e+06
2    2009-12-30     GOOG     622.7      1.466e+06
3    2009-12-31     GOOG     620        1.22e+06
4    2009-12-28     AAPL     211.6      2.3e+07
5    2009-12-29     AAPL     209.1      1.587e+07
6    2009-12-30     AAPL     211.6      1.47e+07
7    2009-12-31     AAPL     210.7      1.257e+07
>>> data['ind'] = data['item'] == 'GOOG'
```

This data can be reshaped into a different form for future examples by means of the `DataFrame` method `pivot`:

```
>>> df = data.pivot('date', 'item', 'value')
>>> df
              AAPL           GOOG
2009-12-28    211.6          622.9
2009-12-29    209.1          619.4
2009-12-30    211.6          622.7
2009-12-31    210.7          620
```

Beyond observational data, one will also frequently encounter *categorical* data, which can be used to partition identifiers into broader groupings. For example, stock tickers might be categorized by their industry or country of incorporation. Here we have created a `DataFrame` object `cats` storing country and industry classifications for a group of stocks:

```
>>> cats
       country   industry
AAPL   US        TECH
IBM    US        TECH
SAP    DE        TECH
GOOG   US        TECH
C      US        FIN
SCGLY  FR        FIN
BAR    UK        FIN
DB     DE        FIN
VW     DE        AUTO
RNO    FR        AUTO
F      US        AUTO
TM     JP        AUTO
```

We will use these objects above to illustrate features of interest.

pandas data model

The **pandas** data structures internally link the axes of a `ndarray` with arrays of unique labels. These labels are stored in instances of the `Index` class, which is a 1D `ndarray` subclass implementing an *ordered set*. In the stock data above, the row labels are simply sequential observation numbers, while the columns are the field names.

An `Index` stores the labels in two ways: as a `ndarray` and as a `dict` mapping the values (which must therefore be unique and hashable) to the integer indices:

```
>>> index = Index(['a', 'b', 'c', 'd', 'e'])
>>> index
Index([a, b, c, d, e], dtype=object)
>>> index.indexMap
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

Creating this `dict` allows the objects to perform lookups and determine membership in constant time.

```
>>> 'a' in index
True
```

These labels are used to provide alignment when performing data manipulations using differently-labeled objects. There are specialized data structures, representing 1-, 2-, and 3-dimensional data, which incorporate useful data handling semantics to facilitate both interactive research and system building. A general *n*-dimensional data structure would be useful in some cases, but data sets of dimension higher than 3 are very uncommon in most statistical and econometric applications, with 2-dimensional being the most prevalent. We took a pragmatic approach, driven by application needs, to designing the data structures in order to make them as easy-to-use as possible. Also, we wanted the objects to be idiomatically similar to those present in other statistical environments, such as R.

Data alignment

Operations between related, but differently-sized data sets can pose a problem as the user must first ensure that the data points are properly aligned. As an example, consider time series over different date ranges or economic data series over varying sets of entities:

```
>>> s1              >>> s2
AAPL   0.044       AAPL   0.025
IBM    0.050       BAR    0.158
SAP    0.101       C      0.028
GOOG   0.113       DB     0.087
C      0.138       F      0.004
SCGLY  0.037       GOOG   0.154
BAR    0.200       IBM    0.034
DB     0.281
VW     0.040
```

One might choose to explicitly align (or *reindex*) one of these 1D `Series` objects with the other before adding them, using the `reindex` method:

```
>>> s1.reindex(s2.index)
AAPL    0.0440877763224
BAR     0.199741007422
C       0.137747485628
DB      0.281070058049
F       NaN
GOOG    0.112861123629
IBM     0.0496445829129
```

However, we often find it preferable to simply ignore the state of data alignment:

```
>>> s1 + s2
AAPL     0.0686791008184
BAR      0.358165479807
C        0.16586702944
```

```
DB       0.367679872693
F        NaN
GOOG     0.26666583847
IBM      0.0833057542385
SAP      NaN
SCGLY    NaN
VW       NaN
```

Here, the data have been automatically aligned based on their labels and added together. The result object contains the union of the labels between the two objects so that no information is lost. We will discuss the use of `NaN` (Not a Number) to represent missing data in the next section.

Clearly, the user pays linear overhead whenever automatic data alignment occurs and we seek to minimize that overhead to the extent possible. Reindexing can be avoided when `Index` objects are shared, which can be an effective strategy in performance-sensitive applications. [Cython], a widely-used tool for easily creating Python C extensions, has been utilized to speed up these core algorithms.

Handling missing data

It is common for a data set to have missing observations. For example, a group of related economic time series stored in a `DataFrame` may start on different dates. Carrying out calculations in the presence of missing data can lead both to complicated code and considerable performance loss. We chose to use `NaN` as opposed to using NumPy MaskedArrays for performance reasons (which are beyond the scope of this paper), as `NaN` propagates in floating-point operations in a natural way and can be easily detected in algorithms. While this leads to good performance, it comes with drawbacks: namely that `NaN` cannot be used in integer-type arrays, and it is not an intuitive "null" value in object or string arrays.

We regard the use of `NaN` as an implementation detail and attempt to provide the user with appropriate API functions for performing common operations on missing data points. From the above example, we can use the `valid` method to drop missing data, or we could use `fillna` to replace missing data with a specific value:

```
>>> (s1 + s2).valid()
AAPL    0.0686791008184
BAR     0.358165479807
C       0.16586702944
DB      0.367679872693
GOOG    0.26666583847
IBM     0.0833057542385

>>> (s1 + s2).fillna(0)
AAPL     0.0686791008184
BAR      0.358165479807
C        0.16586702944
DB       0.367679872693
F        0.0
GOOG     0.26666583847
IBM      0.0833057542385
SAP      0.0
SCGLY    0.0
VW       0.0
```

Common `ndarray` methods have been rewritten to automatically exclude missing data from calculations:

```
>>> (s1 + s2).sum()
1.3103630754662747

>>> (s1 + s2).count()
6
```

Similar to R's `is.na` function, which detects `NA` (Not Available) values, **pandas** has special API functions `isnull` and `notnull` for determining the validity of a data point. These contrast with `numpy.isnan` in that they can be used with dtypes other than `float` and also detect some other markers for "missing" occurring in the wild, such as the Python `None` value.

```
>>> isnull(s1 + s2)
AAPL     False
BAR      False
C        False
DB       False
F        True
GOOG     False
IBM      False
SAP      True
SCGLY    True
VW       True
```

Note that R's `NA` value is distinct from `NaN`. While the addition of a special `NA` value to NumPy would be useful, it is most likely too domain-specific to merit inclusion.

Combining or joining data sets

Combining, joining, or merging related data sets is a quite common operation. In doing so we are interested in associating observations from one data set with another via a *merge key* of some kind. For similarly-indexed 2D data, the row labels serve as a natural key for the `join` function:

```
>>> df1                          >>> df2
            AAPL    GOOG                      MSFT    YHOO
2009-12-24  209     618.5     2009-12-24      31      16.72
2009-12-28  211.6   622.9     2009-12-28      31.17   16.88
2009-12-29  209.1   619.4     2009-12-29      31.39   16.92
2009-12-30  211.6   622.7     2009-12-30      30.96   16.98
2009-12-31  210.7   620

>>> df1.join(df2)
            AAPL    GOOG    MSFT    YHOO
2009-12-24  209     618.5   31      16.72
2009-12-28  211.6   622.9   31.17   16.88
2009-12-29  209.1   619.4   31.39   16.92
2009-12-30  211.6   622.7   30.96   16.98
2009-12-31  210.7   620     NaN     NaN
```

One might be interested in joining on something other than the index as well, such as the categorical data we presented in an earlier section:

```
>>> data.join(cats, on='item')
     country  date        industry item   value
0    US       2009-12-28  TECH     GOOG   622.9
1    US       2009-12-29  TECH     GOOG   619.4
2    US       2009-12-30  TECH     GOOG   622.7
3    US       2009-12-31  TECH     GOOG   620
4    US       2009-12-28  TECH     AAPL   211.6
5    US       2009-12-29  TECH     AAPL   209.1
6    US       2009-12-30  TECH     AAPL   211.6
7    US       2009-12-31  TECH     AAPL   210.7
```

This is akin to a SQL join operation between two tables.

Categorical variables and "Group by" operations

One might want to perform an operation (for example, an aggregation) on a subset of a data set determined by a categorical variable. For example, suppose we wished to compute the mean value by industry for a set of stock data:

```
>>> s              >>> ind
AAPL   0.044       AAPL   TECH
IBM    0.050       IBM    TECH
```

```
SAP     0.101       SAP     TECH
GOOG    0.113       GOOG    TECH
C       0.138       C       FIN
SCGLY   0.037       SCGLY   FIN
BAR     0.200       BAR     FIN
DB      0.281       DB      FIN
VW      0.040       VW      AUTO
                    RNO     AUTO
                    F       AUTO
                    TM      AUTO
```

This concept of "group by" is a built-in feature of many data-oriented languages, such as R and SQL. In R, any vector of non-numeric data can be used as an input to a grouping function such as `tapply`:

```
> labels
[1] GOOG GOOG GOOG GOOG AAPL AAPL AAPL AAPL
Levels: AAPL GOOG
> data
[1] 622.87 619.40 622.73 619.98 211.61 209.10
211.64 210.73

> tapply(data, labels, mean)
   AAPL    GOOG
210.770 621.245
```

pandas allows you to do this in a similar fashion:

```
>>> data.groupby(labels).aggregate(np.mean)
AAPL    210.77
GOOG    621.245
```

One can use `groupby` to concisely express operations on relational data, such as counting group sizes:

```
>>> s.groupby(ind).aggregate(len)
AUTO    1
FIN     4
TECH    4
```

In the most general case, `groupby` uses a function or mapping to produce groupings from one of the axes of a **pandas** object. By returning a `GroupBy` object we can support more operations than just aggregation. Here we can subtract industry means from a data set:

```
demean = lambda x: x - x.mean()

def group_demean(obj, keyfunc):
    grouped = obj.groupby(keyfunc)
    return grouped.transform(demean)

>>> group_demean(s1, ind)
AAPL     -0.0328370881632
BAR       0.0358663891836
C        -0.0261271326111
DB        0.11719543981
GOOG      0.035936259143
IBM      -0.0272802815728
SAP       0.024181110593
SCGLY    -0.126934696382
VW        0.0
```

Manipulating panel (3D) data

A data set about a set of individuals or entities over a time range is commonly referred to as *panel data*; i.e., for each entity over a date range we observe a set of variables. Such data can be found both in *balanced* form (same number of time observations for each individual) or *unbalanced* (different numbers of observations). Panel data manipulations are important for constructing inputs to statistical estimation routines, such as linear regression. Consider the Grunfeld data set [Grun] frequently used in econometrics (sorted by year):

```
>>> grunfeld
       capita    firm      inv       value     year
0      2.8       1         317.6     3078      1935
20     53.8      2         209.9     1362      1935
40     97.8      3         33.1      1171      1935
60     10.5      4         40.29     417.5     1935
80     183.2     5         39.68     157.7     1935
100    6.5       6         20.36     197       1935
120    100.2     7         24.43     138       1935
140    1.8       8         12.93     191.5     1935
160    162       9         26.63     290.6     1935
180    4.5       10        2.54      70.91     1935
1      52.6      1         391.8     4662      1936
21     50.5      2         355.3     1807      1936
41     104.4     3         45        2016      1936
61     10.2      4         72.76     837.8     1936
81     204       5         50.73     167.9     1936
101    15.8      6         25.98     210.3     1936
121    125       7         23.21     200.1     1936
141    0.8       8         25.9      516       1936
161    174       9         23.39     291.1     1936
181    4.71      10        2          87.94     1936
...
```

Really this data is 3-dimensional, with *firm*, *year*, and *item* (data field name) being the three unique keys identifying a data point. Panel data presented in tabular format is often referred to as the *stacked* or *long* format. We refer to the truly 3-dimensional form as the *wide* form. **pandas** provides classes for operating on both:

```
>>> lp = LongPanel.fromRecords(grunfeld, 'year',
                               'firm')
>>> wp = lp.toWide()
>>> wp
<class 'pandas.core.panel.WidePanel'>
Dimensions: 3 (items) x 20 (major) x 10 (minor)
Items: capital to value
Major axis: 1935 to 1954
Minor axis: 1 to 10
```

Now with the data in 3-dimensional form, we can examine the data items separately or compute descriptive statistics more easily (here the `head` function just displays the first 10 rows of the `DataFrame` for the `capital` item):

```
>>> wp['capital'].head()
    1935      1936      1937      1938      1939
1   2.8       265       53.8      213.8     97.8
2   52.6      402.2     50.5      132.6     104.4
3   156.9     761.5     118.1     264.8     118
4   209.2     922.4     260.2     306.9     156.2
5   203.4     1020      312.7     351.1     172.6
6   207.2     1099      254.2     357.8     186.6
7   255.2     1208      261.4     342.1     220.9
8   303.7     1430      298.7     444.2     287.8
9   264.1     1777      301.8     623.6     319.9
10  201.6     2226      279.1     669.7     321.3
```

In this form, computing summary statistics, such as the time series mean for each (item, firm) pair, can be easily carried out:

```
>>> wp.mean(axis='major')
      capital     inv         value
1     140.8       98.45       923.8
2     153.9       131.5       1142
3     205.4       134.8       1140
4     244.2       115.8       872.1
5     269.9       109.9       998.9
6     281.7       132.2       1056
7     301.7       169.7       1148
8     344.8       173.3       1068
9     389.2       196.7       1236
10    428.5       197.4       1233
```

As an example application of these panel data structures, consider constructing dummy variables (columns of 1's and 0's identifying

dates or entities) for linear regressions. Especially for unbalanced panel data, this can be a difficult task. Since we have all of the necessary labeling data here, we can easily implement such an operation as an instance method.

Implementing statistical models

When applying a statistical model, data preparation and cleaning can be one of the most tedious or time consuming tasks. Ideally the majority of this work would be taken care of by the model class itself. In R, while `NA` data can be automatically excluded from a linear regression, one must either align the data and put it into a `data.frame` or otherwise prepare a collection of 1D arrays which are all the same length.

Using **pandas**, the user can avoid much of this data preparation work. As a exemplary model leveraging the **pandas** data model, we implemented ordinary least squares regression in both the standard case (making no assumptions about the content of the regressors) and the panel case, which has additional options to allow for entity and time dummy variables. Facing the user is a single function, `ols`, which infers the type of model to estimate based on the inputs:

```
>>> model = ols(y=Y, x=X)
>>> model.beta
AAPL         0.187984100742
GOOG         0.264882582521
MSFT         0.207564901899
intercept    -0.000896535166817
```

If the response variable `Y` is a `DataFrame` (2D) or `dict` of 1D `Series`, a panel regression will be run on stacked (pooled) data. The `x` would then need to be either a `WidePanel`, `LongPanel`, or a `dict` of `DataFrame` objects. Since these objects contain all of the necessary information to construct the design matrices for the regression, there is nothing for the user to worry about (except the formulation of the model).

The `ols` function is also capable of estimating a *moving window* linear regression for time series data. This can be useful for estimating statistical relationships that change through time:

```
>>> model = ols(y=Y, x=X, window_type='rolling',
                window=250)
>>> model.beta
<class 'pandas.core.matrix.DataFrame'<>
Index: 1103 entries , 2005-08-16 to 2009-12-31
Data columns:
AAPL         1103  non-null values
GOOG         1103  non-null values
MSFT         1103  non-null values
intercept    1103  non-null values
dtype: float64(4)
```

Here we have estimated a moving window regression with a window size of 250 time periods. The resulting regression coefficients stored in `model.beta` are now a `DataFrame` of time series.

Date/time handling

In applications involving time series data, manipulations on dates and times can be quite tedious and inefficient. Tools for working with dates in MATLAB, R, and many other languages are clumsy or underdeveloped. Since Python has a built-in `datetime` type easily accessible at both the Python and C / Cython level, we aim to craft easy-to-use and efficient date and time functionality. When the NumPy `datetime64` dtype has matured, we will, of course, reevaluate our date handling strategy where appropriate.

For a number of years **scikits.timeseries** [SciTS] has been available to scientific Python users. It is built on top of MaskedArray and is intended for fixed-frequency time series. While forcing data to be fixed frequency can enable better performance in some areas, in general we have found that criterion to be quite rigid in practice. The user of **scikits.timeseries** must also explicitly align data; operations involving unaligned data yield unintuitive results.

In designing **pandas** we hoped to make working with time series data intuitive without adding too much overhead to the underlying data model. The **pandas** data structures are *datetime-aware* but make no assumptions about the dates. Instead, when frequency or regularity matters, the user has the ability to generate date ranges or conform a set of time series to a particular frequency. To do this, we have the `DateRange` class (which is also a subclass of `Index`, so no conversion is necessary) and the `DateOffset` class, whose subclasses implement various general purpose and domain-specific time increments. Here we generate a date range between 1/1/2000 and 1/1/2010 at the "business month end" frequency `BMonthEnd`:

```
>>> DateRange('1/1/2000', '1/1/2010',
              offset=BMonthEnd())
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessMonthEnd>
[2000-01-31 00:00:00, ..., 2009-12-31 00:00:00]
length: 120
```

A `DateOffset` instance can be used to convert an object containing time series data, such as a `DataFrame` as in our earlier example, to a different frequency using the `asfreq` function:

```
>>> monthly = df.asfreq(BMonthEnd())
             AAPL      GOOG      MSFT      YHOO
2009-08-31   168.2     461.7     24.54     14.61
2009-09-30   185.3     495.9     25.61     17.81
2009-10-30   188.5     536.1     27.61     15.9
2009-11-30   199.9     583       29.41     14.97
2009-12-31   210.7     620       30.48     16.78
```

Some things which are not easily accomplished in scikits.timeseries can be done using the `DateOffset` model, like deriving custom offsets on the fly or shifting monthly data forward by a number of business days using the `shift` function:

```
>>> offset = Minute(12)
>>> DateRange('6/18/2010 8:00:00',
              '6/18/2010 12:00:00',
              offset=offset)
<class 'pandas.core.daterange.DateRange'>
offset: <12 Minutes>
[2010-06-18 08:00:00, ..., 2010-06-18 12:00:00]
length: 21

>>> monthly.shift(5, offset=Bay())
             AAPL    GOOG    MSFT    YHOO
2009-09-07   168.2   461.7   24.54   14.61
2009-10-07   185.3   495.9   25.61   17.81
2009-11-06   188.5   536.1   27.61   15.9
2009-12-07   199.9   583     29.41   14.97
2010-01-07   210.7   620     30.48   16.78
```

Since **pandas** uses the built-in Python `datetime` object, one could foresee performance issues with very large or high frequency time series data sets. For most general applications financial or econometric applications we cannot justify complicating `datetime` handling in order to solve these issues; specialized tools will need to be created in such cases. This may be indeed be a fruitful avenue for future development work.

Related packages

A number of other Python packages have appeared recently which provide some similar functionality to **pandas**. Among these, **la** ([Larry]) is the most similar, as it implements a labeled `ndarray` object intending to closely mimic NumPy arrays. This stands in contrast to our approach, which is driven by the practical considerations of time series and cross-sectional data found in finance, econometrics, and statistics. The references include a couple other packages of interest ([Tab], [pydataframe]).

While **pandas** provides some useful linear regression models, it is not intended to be comprehensive. We plan to work closely with the developers of **scikits.statsmodels** ([StaM]) to generally improve the cohesiveness of statistical modeling tools in Python. It is likely that **pandas** will soon become a "lite" dependency of **scikits.statsmodels**; the eventual creation of a *superpackage* for statistical modeling including **pandas**, **scikits.statsmodels**, and some other libraries is also not out of the question.

Conclusions

We believe that in the coming years there will be great opportunity to attract users in need of statistical data analysis tools to Python who might have previously chosen R, MATLAB, or another research environment. By designing robust, easy-to-use data structures that cohere with the rest of the scientific Python stack, we can make Python a compelling choice for data analysis applications. In our opinion, **pandas** represents a solid step in the right direction.

REFERENCES

[pandas] W. McKinney, AQR Capital Management, *pandas: a python data analysis library*, http://pandas.sourceforge.net

[Larry] K. Goodman. *la / larry: ndarray with labeled axes*, http://larry.sourceforge.net/

[SciTS] M. Knox, P. Gerard-Marchant, *scikits.timeseries: python time series analysis*, http://pytseries.sourceforge.net/

[StaM] S. Seabold, J. Perktold, J. Taylor, *scikits.statsmodels: statistical modeling in Python*, http://statsmodels.sourceforge.net

[SciL] D. Cournapeau, et al., *scikits.learn: machine learning in Python*, http://scikit-learn.sourceforge.net

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

[Tab] D. Yamins, E. Angelino, *tabular: tabarray data structure for 2D data*, http://parsemydata.com/tabular/

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

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

[matplotlib] J. Hunter, et al., *matplotlib: Python plotting*, http://matplotlib.sourceforge.net/

[EPD] Enthought, Inc., *EPD: Enthought Python Distribution*, http://www.enthought.com/products/epd.php

[Pythonxy] P. Raybaut, *Python(x,y): Scientific-oriented Python distribution*, http://www.pythonxy.com/

[CRAN] *The R Project for Statistical Computing*, http://cran.r-project.org/

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn, et al., *The Cython compiler*, http://cython.org

[IPython] F. Perez, et al., *IPython: an interactive computing environment*, http://ipython.scipy.org

[Grun] Batalgi, *Grunfeld data set*, http://www.wiley.com/legacy/wileychi/baltagi/

[nipy] J. Taylor, F. Perez, et al., *nipy: Neuroimaging in Python*, http://nipy.sourceforge.net

[pydataframe] A. Straw, F. Finkernagel, *pydataframe*, http://code.google.com/p/pydataframe/

[R] R Development Core Team. 2010, *R: A Language and Environment for Statistical Computing*, http://www.R-project.org

[MATLAB] The MathWorks Inc. 2010, *MATLAB*, http://www.mathworks.com

[Stata] StatCorp. 2010, *Stata Statistical Software: Release 11* http://www.stata.com

[SAS] SAS Institute Inc., *SAS System*, http://www.sas.com

Protein Folding with Python on Supercomputers

Jan H. Meinke‡*

Abstract—Today's supercomputers have hundreds of thousands of compute cores and this number is likely to grow. Many of today's algorithms will have to be rethought to take advantage of such large systems. New algorithms must provide fine grained parallelism and excellent scalability. Python offers good support for numerical libraries and offers bindings to MPI that can be used to develop parallel algorithms for distributed memory machines.

PySMMP provides bindings to the protein simulation package SMMP. Combined with mpi4py, PySMMP can be used to perform parallel tempering simulations of small proteins on the supercomputers JUGENE and JuRoPA. In this paper, the performance of the Fortran implementation of parallel tempering in SMMP is compared with the Python implementation in PySMMP. Both codes use the same Fortran code for the calculation of the energy.

The performance of the implementations is comparable on both machines, but some challenges remain before the Python implementation can replace the Fortran implementation for all production runs.

Index Terms—parallel, MPI, biology, protein structure

Introduction

Many of the problems well known to high-performance computing (HPC) are becoming main stream. Processors add more and more cores, but the performance of a single core does not improve as drastically as it used to. Suddenly everybody has to deal with tens or hundreds of processes or threads. To take advantage of graphics hardware another factor of 100 in the number of threads is needed. Issues such as task management, load balancing, and race conditions are starting to become known to everybody who wants to write efficient programs for PCs. But things work the other way around, too. High-level programming languages such as Python that were not developed to get peak performance but to make good use of the developers time are becoming increasingly popular in HPC.

The Simulation Laboratory Biology at the Juelich Supercomputing Centre (JSC) uses Python to script workflows, implement new algorithms, perform data analysis and visualization, and to run simulations on the supercomputers JUGENE and JuRoPA. Often, we combine existing packages such as Biopython [BioPy], Modeller [MOD], matplotlib [PyLab], or PyQt [PyQt] with our own packages to tackle the scientific problems we are interested in. In this paper I will focus on using Python for protein folding studies on JuRoPA and JUGENE.

* *Corresponding author: j.meinke@fz-juelich.de*
‡ *Jülich Supercomputing Centre*

Proteins

Proteins bind oxygen and carbon dioxide, transport nutrients and waste products. They catalyze reactions, transfer information, and perform many other important functions. Proteins don't act in isolation. They are part of an interaction network that allows a cell to perform all the necessary operations of life. A very important feature of a protein is its shape. Only when it obtains its correct three dimensional structure does it provide the right interface for its reaction partners. In fact, changing the interface is a way to turn proteins on and off and regulate their activity.

Proteins are long chains of amino acids. The sequence of amino acids determines a protein's native shape. The sequence is encoded in the genome and assembled by the ribosome—itself a complex of RNA and proteins—amino acid by amino acid.

Proteins need anywhere from a few micro seconds to several minutes to obtain their native structure. This process is called protein folding. It occurs reliably in our body many times each second yet it is still poorly understood.

For some globular proteins it has been shown that they can unfold and refold in a test tube. At least for these proteins folding is driven purely by classical physical interactions. This is the basis for folding simulations using classical force fields.

The program package SMMP first released in 2001 [SMMP] implements several Monte Carlo algorithms that can be used to study protein folding. It uses a simplified model of a protein that keeps the bond angles and lengths fixed and only allows changes of the dihedral angles. To calculate the energy of a given conformation of a protein, SMMP also implements several energy functions, the so called force fields. A force field is defined by a functional form of the energy function and its parametrization.

In 2007, we released version 3 of SMMP [SMMP3]. With this version we provided Python bindings PySMMP that made the properties of the proteins, the algorithms, and the calculation of energy available from Python. In addition to the wrapper library created with f2py, we included three modules: universe, protein, and algorithms that make setting up a simulation and accessing the properties of a protein much more convenient. The wrapper modules were inspired by the Molecular Modeling Toolkit [MMTK], but implement a flat hierarchy. We did not, however, include the parallelized energy functions, which requires MPI to work.

For the work described in this paper, I decided to use mpi4py as MPI bindings for the Python code for its completeness and its integration with Scientific Python and Cython. An important feature of mpi4py is that it provides easy access to communicators in a way that can be passed to the Fortran subroutine called.

Compiling the Modules

JUGENE is a 72-rack IBM Blue Gene/P (BG/P) system installed at JSC. Each rack consists of 1024 compute nodes. Each compute node has a 4-core PowerPC 450 processor running at 850 MHz and 2 GB of memory for a total of 294912 cores and 147 TB of memory. The nodes are connected via a three dimensional torus network. Each node is linked to its six neighbors. In addition to the torus network, BG/P features a tree network that is used for global communication. The nodes are diskless. They forward IO requests to special IO nodes, which in turn talk to the GPFS file system. JUGENE's peak performance is about one petaflop and it reaches about 825 teraflops in the Linpack benchmark. This makes it Europe's fastest computer and the number 5 in the world [Top500]. While the slow clock rate makes the system very energy efficient (364 MFlops/W), it also makes code that scales well a must, since each individual core provides only about one third of the peak performance of an Intel Nehalem core and the performance gap is even larger in many applications. Production runs on JUGENE should use at least one rack.

Programs that run on JUGENE are usually cross-compiled for the compute nodes. The compute nodes run a proprietary 32-bit compute node kernel with reduced functionality whereas the login nodes use Power6 processors with a full 64-bit version of SUSE Linux Enterprise Server 10. Cross compiling can be tricky. It is important to set all the environment variables and paths correctly. First, we need to make sure to use the correct compiler

```
export BGPGNU=/bgsys/drivers/ppcfloor/gnu-linux
export F90=$BGPGNU/powerpc-bgp-linux/bin/gfortran
```

Then we need to use f2py with the correct Python interpreter, for example

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$BGPGNU/lib
$BGPGNU/bin/python /bgsys/local/numpy/1.2.1/bin/f2py
```

Now, f2py produces libraries that can be loaded on the compute nodes.

Launching Python on thousands of cores

A first step for any Python program is to load the interpreter and the default modules. While this is usually not a problem if we start a few instances, it can become troublesome on a large system such as JUGENE.

Taking a look at the first two columns in Table 1 we see that already for a single rack, it takes more than 5 minutes to run a simple helloworld program using the default Python installation location. A C++ program for comparison takes only 5 s. Plotting the run time of the helloworld program, we quickly see that the time increases linearly with the number of MPI tasks at a rate of 0.1 s per task (Blue squares in Figure 1). Extrapolating this to all 294912 cores of JUGENE, it would take more than 8 hours to start the Python interpreter resulting in 25 lost rack days (70 CPU years with 4 cores per CPU) and almost 10 metric tons of CO_2.

The linear behavior hints at serialization when the Python interpreter is loaded. As mentioned above, JUGENE's, compute nodes don't have their own disks. All IO is done via special IO nodes from a parallel file system and all nodes access the same Python image on the disk.

A similar behavior was discussed for the GPAW code in the mpi4py forum [PyOn10k]. GPAW [GPAW] uses its own Python MPI interface. Their work around was to use the ram disks of the IO nodes on Blue Gene/P.

# of Cores	Time [s]	Time [s]	Comments
1	5		
128	50	20	A single node card
512	55		Midplane in SMP mode
1024	100		Only rank 0 writes
2048	376		195 s if only rank 0 writes
4096	321	130	1 rack (smallest size for production runs)
8192	803	246	2 racks
16384	1817	371	4 racks. For comparison, a C++ program takes 25 s.
20480		389	5 racks
32768		667	8 racks
65536		927	16 racks
131071		1788	32 rack

TABLE 1: *Time measured for a simple MPI hello world program written using mpi4py on the Blue Gene/P JUGENE. The second column gives the times using the default location for Python on Blue Gene. The third column lists the times if Python is installed in the Work file system.*

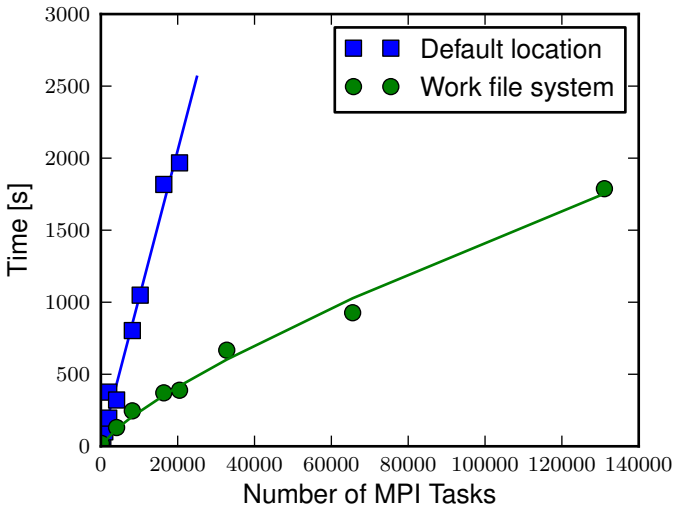


Fig. 1: *Scaling of the startup time of the Python interpreter on JUGENE before and after optimization. Using the default location of the Python installation, the startup time increases linearly with the number of MPI tasks. Moving the Python installation to the faster Work file system reduces the scaling exponent from 1 to 0.77.*

Based on this data, we filed a service request with IBM. After some experimentation, IBM finally suggested to install Python on the Work file system. The Work file system is usually used as a skratch space for simulation data that is written during a run. Its block size of 2 MB is optimized for large files and it reaches a bandwidth of 30 GB/s. Files written to the Work file system usually are deleted automatically after 90 days. In comparison the system and home file systems use a block size of 1 MB and reach a bandwidth of 8 GB/s.

With Python installed on the Work file system, the scaling of the runtime of the helloworld program becomes sublinear with an exponent of about 0.77 (see column three in Table 1 and green disks in Figure 1). This make production runs of up to 32 racks

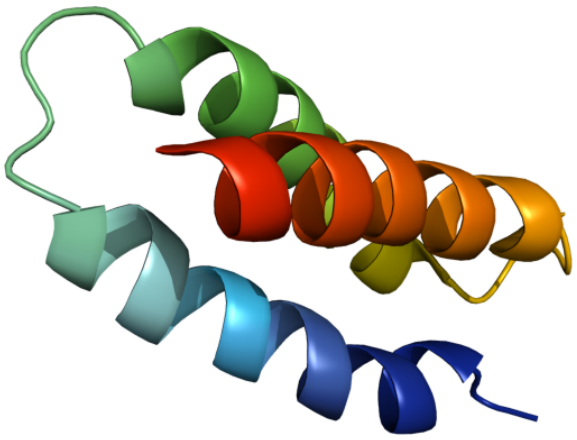


Fig. 2: *Cartoon rendering of the three-helix bundle GS-α_3W. The rendering was done with PyMOL [PyMOL].*

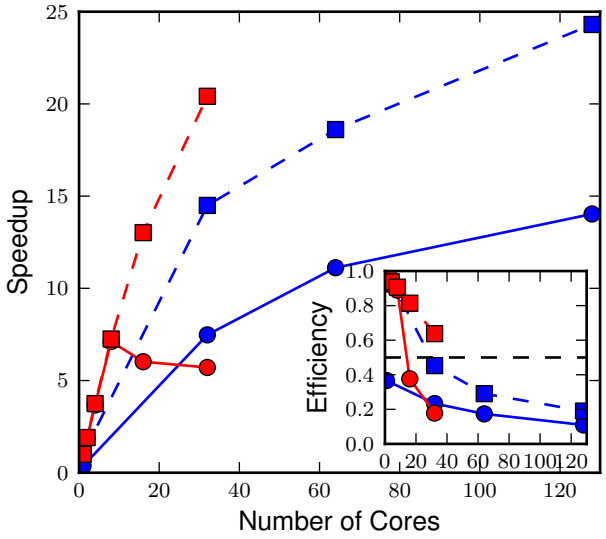


Fig. 3: *Parallel scaling of the duration of the energy calculation for the three-helix bundle GS-α_3W on JuRoPA (red) and JUGENE (blue). The speedup is relative to the time needed by the Fortran program for the calculation of the energy on a single core. The square symbols represent SMMP, the disks PySMMP.*

(131071 cores) feasible. Extrapolating the data to 72 racks, it would now take less than an hour to start a run on the entire machine.

I also ran the same test on our second supercomputer, JuRoPA. JuRoPA is an Intel Nehalem cluster. Each of its 3288 nodes has two quad-core processors with 24 GB of memory for a total of 26304 cores and 79 TB of main memory. It has a peak performance of 308 teraflops and is currently number 14 in the Top 500 list with 90% efficiency in the Linpack benchmark [Top500]. It uses Infiniband in a fat tree topology for communication and a Lustre file system for storage. In contrast to JUGENE, each node has its own local disk, where Python is installed. While the time to start Python and load mpi4py.MPI still increases linearly with the number of nodes, the prefactor is only 0.005 s per process.

Parallel energy calculation

As mentioned above, the energy calculation for the ECEPP/3 force field and the associated implicit solvent term are parallelized. Before they can be used, however, the appropriate communicator needs to be defined. For most simulations, except parallel tempering (see Section Parallel tempering), the communicator is a copy of the default communicator that includes all processes. To start, such a simulation, we need to assign this communicator to smmp.paral.my_mpi_comm. This must be the appropriate Fortran reference, which we can get using mpi4py.MPI.COMM_WORLD.py2f(). With this setup, we can now compare the speed and the scaling of the energy function when called from Python and Fortran.

Scaling in parallel programs refers to the speedup when the program runs on p processors compared to running it on one processor. If the run time with p processors is given by $t(p)$ then the speedup s is defined as $s(p) = t(1)/t(p)$ and the efficiency of the scaling is given by $e(p) = s(p)/p$. An efficiency of 50% is often considered acceptable.

As a benchmark system, I used the three-helix bundle GS-α_3W (PDB code: 1LQ7) with 67 amino acids and 1110 atoms (see Figure 2).

On JuRoPA, I used f2py's default optimization options for the Intel compiler to create the bindings. The Fortran program was compiled with the -fast option, which activates most optimizations and includes interprocedural optimizations. For a single core, the Fortran program is about 10% faster. The scaling on a single node is comparable, but it breaks down for PySMMP if more than one node is used (see Figure 3). This may be due to interactions between mpi4py and JuRoPA's MPI installation.

On JUGENE, the behavior is quite different. PySMMP was compiled with gfortran, SMMP with IBM's xlf compiler, which produces code that is almost three times faster on a single core. The shape of the scaling is comparable and saturates at about 128 cores.

Parallel tempering

Parallel tempering [PT], also known as replica exchange, is a method to sample a rough energy landscape more efficiently. Several copies of a system are simulated at different temperatures. In addition to regular Monte Carlo [MC] moves that change a configuration, we introduce a move that exchanges conformations of two different temperatures. The probability for such a move is $P_{\mathrm{PT}} = \exp(\Delta\beta\Delta E)$, where $\beta = 1/k_B T$, T is the temperature and k_B is the Boltzmann constant. With this exchange probability the statistics at each temperature remains correct, yet conformations can move to higher temperatures where it is easier to overcome large barriers. This allows for a more efficient sampling of the conformational space of a protein.

Parallel tempering is by its very nature a parallel algorithm. At each temperature, we perform a regular canonical MC simulation. After a number of updates n_{up}, we attempt an exchange between temperatures. If we create our own MPI communicators, we can use two levels of parallelism. For each temperature T_i, we use a number of processors p_i to calculate the energy in parallel. Usually, p_i is the same for all temperatures, but this is not a requirement. Assuming that $p_i = p$, and using n_T temperatures, we use a total of $p_{\mathrm{tot}} = n_T * p$ processors. For an average protein domain consisting of about 150 amino acids and 3000 atoms, $p = 128$, and $n_T = 64$ is a reasonable choice on a Blue Gene/P, for a total of $p_{\mathrm{tot}} = 8192$—a good size for a production run.

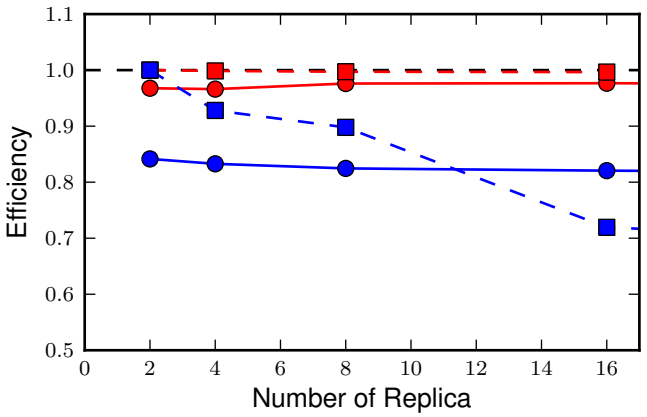


Fig. 4: *Efficiency of the scaling of parallel tempering. Parallel tempering is an example for weak scaling. The problem size, i.e., the number of temperatures, increases proportional to the number of processors. Ideally, the time stays constant and the efficiency is one. For JuRoPA (red), both lines are nearly constant. The Python implementation (disks) of parallel tempering takes only about 5% longer than the Fortran version (squares). On JUGENE (blue) each replica uses 128 cores for the energy calculation. The Python implementation takes about 20% longer for 2 replica than the Fortran implementation but for 16 replica the difference is down to about 10%.*

Parallel tempering is implemented in Fortran as part of SMMP. The speed of the Fortran implementation is the reference, for the following investigation of my implementation of parallel tempering in Python. Parallel tempering and canonical Monte Carlo are implemented as classes in the algorithms module. The canonical Monte Carlo class optionally uses the Fortran implementation of the Metropolis step. For the following comparison, only the calculation of the energy of a conformation is done in Fortran.

For parallel tempering, the number of processes increases proportionally with the number of replicas. This kind of scaling is called weak scaling. Ideally, the time stays constant. Figure 4 shows the scaling of parallel tempering on JuRoPA and JUGENE with respect to the pure Fortran program. On JuRoPA, one node was used per replica. On JUGENE 128 cores were used per replica. The overhead of implementing the algorithm in Python is about 5% on JuRoPA and the scaling is comparable to the Fortran code. On JUGENE, the overhead of the Python implementation is about 20% for 2 replicas. But the scaling of PySMMP is better and for 16 replicas, the Python version takes only about 10% longer.

Clustering

In addition to scalar properties such as energy, volume, secondary structure content, and distance to the native structure, we can save the conformation, i.e., the coordinates of the structures, we have seen. We can create histograms that show us for each temperature, how often, we found structures that had a distance to the native conformation that fell into a certain range. A commonly used measure is the root-mean-square deviation (rmsd) of the current conformation to the native one. Rmsd measures the average change in position of all atoms compared to a reference structure. Unfortunately, rmsd is not a very good measure. For small rmsd values, two structures that have a similar rmsd to the native structure, will also be similar to each other, but for larger rmsd values this is not the case. To determine, the recurrence and therefore the statistical weight of structures that are very different from a given reference structure, we can use clustering algorithms. A cluster can be defined in many different ways. Three intuitive definitions are

- Elements belong to the same cluster if their distance to each other is less than a given distance d_{cluster}.
- Elements belong to the same cluster if they have more connections to each other than to other elements.
- Two clusters are distinct if the density of elements within the cluster is much higher than between clusters.

The first definition works well with rmsd as distance measure if we choose d_{cluster} small enough and is an intuitive definition for clusters of structures, but it is computationally expensive. We usually have several tens of thousands of structures requiring billions of rmsd calculations to complete the distance matrix. We therefore started to look at alternatives. One alternative is to look for dense regions in high-dimensional spaces (the third definition). MAFIA [MAFIA] is a adaptive grid algorithm to determine such clusters. It looks for dense regions in increasingly higher dimension. A one-dimensional region is considered dense if the number of elements is larger than a threshold $n_t = \alpha \bar{n} w$, where α is a parameter, $\bar{n}$ is the average density of elements in that dimension, and w is the width of the region. An n-dimensional region is considered dense if the number of elements it contains is larger than the threshold of each of its one-dimensional sub spaces For each dimension, MAFIA divides space into n_{bins} uniform bins (see Figure 5). For each bin, it counts the number of elements in that bin creating a histogram. The next step is to reduce the number of bins by enveloping the histogram using n_{windows} windows. The value of each window is the maximum of the bins it contains. To build an adaptive grid, neighboring windows are combined into larger cells if their values differ by less than a factor β. For each adaptive-grid cell, the threshold n_t is calculated. The one-dimensional dense cells are used to find two dimensional candidate dense units. The algorithm combines the dense units found to find increasingly higher-dimensional dense units. It takes advantage of the fact that all $n-1$-dimensional projections of an n-dimensional dense unit are also dense to quickly reduce the number of higher-dimensional cells that need to be tested.

Since, we couldn't find an implementation of MAFIA, I implemented a Python version using NumPy and mpi4py. MAFIA combines task and data parallelism making it a good candidate for parallel compute clusters. The implementation consists of less than 380 lines of code, scales well, and can deal easily with tens of thousands of data points.

We are currently testing the usefulness of various ways to describe protein conformations as multi-dimensional vectors for clustering using PyMAFIA.

Conclusions

Today's supercomputers consist of tens to hundreds of thousands of cores and the number of cores is likely to grow. Using these large systems efficiently requires algorithms that provide a lot of parallelism. Python with mpi4py provides an avenue to implement and test these algorithms quickly and cleanly. The implementation of MAFIA shows that prototyping of a parallel program can be done efficiently in pure Python

On JuRoPA, the overhead of using Python instead of Fortran for the parallel tempering algorithm, is only about 3% if the energy

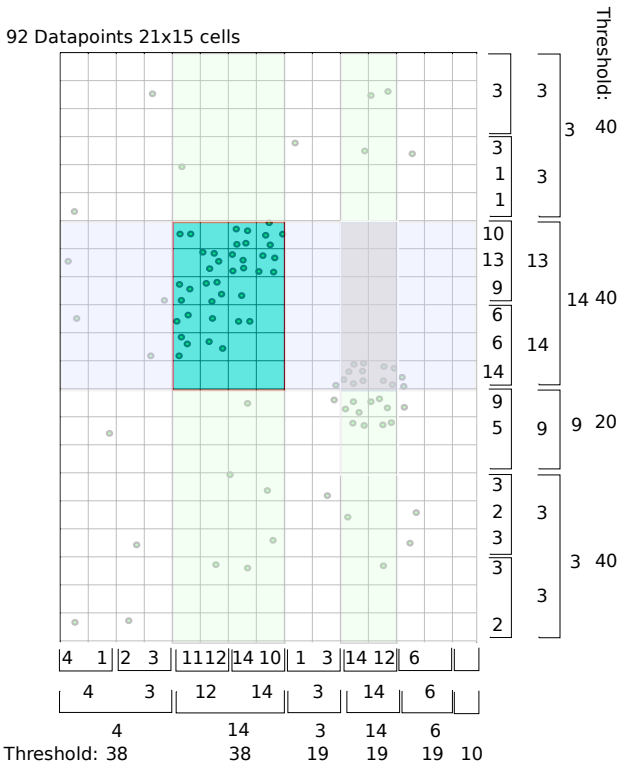


Fig. 5: *An illustration of MAFIA using a simple two-dimensional example with $\alpha = 1.5$. The light green columns and the light blue row are one-dimensional dense units. The areas where they cross are two-dimensional candidates for dense units, but only the darker cyan area is dense. It contains more particles than required by the thresholds of its one-dimensional components.*

calculation is done on a single node. But the scaling of the energy calculation when called from Fortran is better than the scaling of the same function called from Python. This may be due to the interplay between mpi4py and JuRoPA's MPI installation and needs further investigation.

Vendors are interested in making Python work on their machines. IBM helped us to improve the scaling of the startup time of Python on our Blue Gene/P. This now makes production runs with more than 100000 cores feasible and reduces the extrapolated time to start Python on the entire machine from more than eight hours to less than one hour.

Still, the goal remains to bring the startup time of the Python interpreter on JUGENE down near that of a regular binary program. We will continue to investigate.

REFERENCES

[BioPy] Cock PJ, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, Friedberg I, Hamelryck T, Kauff F, Wilczynski B, and de Hoon MJ. *Biopython: freely available Python tools for computational molecular biology and bioinformatics.* Bioinformatics **25** (11), 1422-3 (2009)

[MOD] Sali A. and Blundell T. L. *Comparative protein modelling by satisfaction of spatial restraints.* J. Mol. Biol. **234**, 779-815 (1993)

[PyLab] Hunter J.D. *Matplotlib: A 2D Graphics Environment* Computing in Science and Engineering, **9** (3), 90-95 (2007)

[PyQt] *Qt - A cross platform application and UI framework*, http://qt.nokia.com/

[SMMP] Eisenmenger, F., Hansmann, U.H.E., Hayryan, S. & Hu, C. *[SMMP] A modern package for simulation of proteins.* Comp. Phys. Comm. **138**, 192-212 (2001).

[SMMP3] Meinke, J.H., Mohanty, S., Eisenmenger, F. & Hansmann, U.H.E. *SMMP v. 3.0 - Simulating proteins and protein interactions in Python and Fortran.* Comp. Phys. Comm **178**, 459--470 (2007).

[MMTK] Hinsen, K *The Molecular Modeling Toolkit: A New Approach to Molecular Simulations* J. Comp. Chem. **21**, 79--85 (2000)

[Top500] *Top 500 List June 2010*, http://www.top500.org/list/2010/06/100

[PyOn10k] *Python on 10K of cores on BG/P*, http://groups.google.com/group/mpi4py/browse_thread/thread/3dc9b1d9eb153eb3

[GPAW] Mortensen, J.J., Hansen, L.B. & Jacobsen, K.W. *Real-space grid implementation of the projector augmented wave method.* Phys. Rev. B **71**, 035109 (2005).

[PyMOL] *The PyMOL Molecular Graphics System*, Version 1.2r3pre, Schroedinger, LLC., http://www.pymol.org/

[MC] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. & Teller, E. *Equation of state calculations by fast computing machines.* J. Chem. Phys. **21**, 1087 (1953).

[PT] Hukushima, K. & Nemoto, K. *Exchange Monte Carlo Method and Application to Spin Glass Simulations.* J. Phys. Soc. Jpn. **65**, 1604-1608 (1996).

[MAFIA] Nagesh, H., Goil, S. & Choudhary, A. *Parallel algorithms for clustering high-dimensional large-scale datasets.* Data mining for scientific and engineering applications (2001).

SpacePy - A Python-based Library of Tools for the Space Sciences

Steven K. Morley‡*, Daniel T. Welling‡, Josef Koller‡, Brian A. Larsen‡, Michael G. Henderson‡, Jonathan Niehof‡

Abstract—Space science deals with the bodies within the solar system and the interplanetary medium; the primary focus is on atmospheres and above—at Earth the short timescale variation in the the geomagnetic field, the Van Allen radiation belts and the deposition of energy into the upper atmosphere are key areas of investigation.

SpacePy is a package for Python, targeted at the space sciences, that aims to make basic data analysis, modeling and visualization easier. It builds on the capabilities of the well-known NumPy and matplotlib packages. Publication quality output direct from analyses is emphasized. The SpacePy project seeks to promote accurate and open research standards by providing an open environment for code development. In the space physics community there has long been a significant reliance on proprietary languages that restrict free transfer of data and reproducibility of results. By providing a comprehensive library of widely-used analysis and visualization tools in a free, modern and intuitive language, we hope that this reliance will be diminished for non-commercial users.

SpacePy includes implementations of widely used empirical models, statistical techniques used frequently in space science (e.g. superposed epoch analysis), and interfaces to advanced tools such as electron drift shell calculations for radiation belt studies. SpacePy also provides analysis and visualization tools for components of the Space Weather Modeling Framework including streamline tracing in vector fields. Further development is currently underway. External libraries, which include well-known magnetic field models, high-precision time conversions and coordinate transformations are accessed from Python using ctypes and f2py. The rest of the tools have been implemented directly in Python.

The provision of open-source tools to perform common tasks will provide openness in the analysis methods employed in scientific studies and will give access to advanced tools to all space scientists, currently distribution is limited to non-commercial use.

Index Terms—astronomy, atmospheric science, space weather, visualization

Introduction

For the purposes of this article we define space science as the study of the plasma environment of the solar system. That is, the Earth and other planets are all immersed in the Sun's tenuous outer atmosphere (the heliosphere), and all are affected in some way by natural variations in the Sun. This is of particular importance at Earth where the magnetized plasma flowing out from the Sun interacts with Earth's magnetic field and can affect technological systems and climate. The primary focus here is on planetary atmospheres and above - at Earth the short timescale variation in the the geomagnetic field, the Van Allen radiation belts [Mor10] and the deposition of energy into the upper atmosphere [Mly10] are key areas of investigation.

SpacePy was conceived to provide a convenient library for common tasks in the space sciences. A number of routine analyses used in space science are much less common in other fields (e.g. superposed epoch analysis) and modules to perform these analyses are provided. This article describes the initial release of SpacePy (0.1.0), available from Los Alamos National Laboratory. at http://spacepy.lanl.gov. Currently SpacePy is available on a non-commercial research license, but open-sourcing of the software is in process.

SpacePy organization

As packages such as NumPy, SciPy and matplotlib have become de facto standards in Python, we have adopted these as the prerequisites for SpacePy.

The SpacePy package provides a number of modules, for a variety of tasks, which will be briefly described in this article. HTML help for SpacePy is generated using epydoc and is bundled with the package. This can be most easily accessed on import of SpacePy (or any of its modules) by running the help() function in the appropriate namespace. A schematic of the organization of SpacePy is shown in figure 1. In this article we will describe the core modules of SpacePy and provide some short examples of usage and output.

The most general of the bundled modules is *Toolbox*. At the time of writing this contains (among others): a convenience function for graphically displaying the contents of dictionaries recursively; windowing mean calculations; optimal bin width estimation for histograms via the Freedman-Diaconis method; an update function to fetch the latest OMNI (solar wind/geophysical index) database and leap-second list; comparison of two time series for overlap or common elements.

The other modules have more specific aims and are primarily based on new classes. *Time* provides a container class for times in a range of time systems, conversion between those systems and extends the functionality of datetime for space science use. *Coordinates* provides a class, and associated functions, for the handling of coordinates and transformations between common coordinate systems. *IrbemPy* is a module that wraps the IRBEM magnetic field library. *Radbelt* implements a 1-D radial diffusion code along with diffusion coefficient calculations and plotting routines. *SeaPy* provides generic one- and two-dimensional superposed epoch analysis classes and some plotting and statistical testing

* *Corresponding author: smorley@lanl.gov*
‡ *Los Alamos National Laboratory*

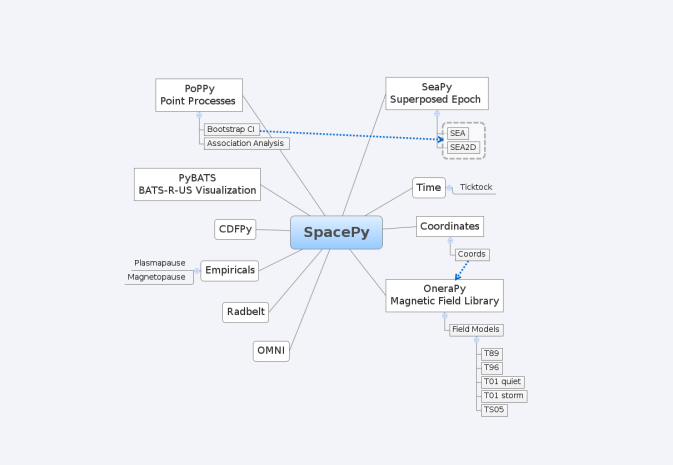


Fig. 1: *A schematic of the organization and contents of the SpacePy package at the time of writing.*

for superposed epoch analysis. *PoPPy* is a module for analysis of point processes, in particular it provides association analysis tools. *Empiricals* provides implementations of some common empirical models such as plasmapause and magnetopause locations. *PyBATS* is an extensive sub-package providing tools for the convenient reading, writing and display of output from the Space Weather Modeling Framework (a collection of coupled models of the Sun-Earth system). *PyCDF* is a fully object-oriented interface to the NASA Common Data Format library.

Time conversions

SpacePy provides a time module that enables convenient manipulation of times and conversion between time systems commonly used in space sciences:

1) NASA Common Data Format (CDF) epoch
2) International Atomic Time (TAI)
3) Coordinated Universal Time (UTC)
4) Gregorian ordinal time (RDT)
5) Global Positioning System (GPS) time
6) Julian day (JD)
7) modified Julian day (MJD)
8) day of year (DOY)
9) elapsed days of year (eDOY)
10) UNIX time (UNX)

This is implemented as a container class built on the functionality of the core Python datetime module. To illustrate its use, we present code which instantiates a `Ticktock` object, and fetches the time in different systems:

```
>>> import spacepy.time as spt
SpacePy: Space Science Tools for Python
SpacePy is released under license.
See __licence__ for details,
and help() for HTML help.
>>> ts = spt.Ticktock([`2009-01-12T14:30:00',
...                    `2009-01-13T14:30:00'],
...                    `ISO')
>>> ts
Ticktock([`2009-01-12T14:30:00',
          `2009-01-13T14:30:00']),
          dtype=ISO
>>> ts.UTC
[datetime.datetime(2009, 1, 12, 14, 30),
 datetime.datetime(2009, 1, 13, 14, 30)]
>>> ts.TAI
array([  1.61046183e+09,   1.61054823e+09])
>>> ts.isoformat(`microseconds')
>>> ts.ISO
[`2009-01-12T14:30:00.000000',
 `2009-01-13T14:30:00.000000']
```

Coordinate handling

Coordinate handling and conversion is performed by the *coordinates* module. This module provides the *Coords* class for coordinate data management. Transformations between cartesian and spherical coordinates are implemented directly in Python, but the coordinate conversions are currently handled as calls to the IRBEM library.

In the following example two locations are specified in a geographic cartesian coordinate system and converted to spherical coordinates in the geocentric solar magnetospheric (GSM) coordinate system. The coordinates are stored as object attributes. For coordinate conversions times must be supplied as many of the coordinate systems are defined with respect to, e.g., the position of the Sun, or the plane of the Earth's dipole axis, which are time-dependent.

```
>>> import spacepy.coordinates as spc
>>> import spacepy.time as spt
>>> cvals = spc.Coords([[1,2,4],[1,2,2]],
...                    `GEO', `car')
>>> cvals.ticktock = spt.Ticktock(
...     [`2002-02-02T12:00:00',
...      `2002-02-02T12:00:00'],
...     `ISO')
>>> newcoord = cvals.convert(`GSM', `sph')
```

A new, higher-precision C library to perform time conversions, coordinate conversions, satellite ephemeris calculations, magnetic field modeling and drift shell calculations—the LANLGeoMag (LGM) library—is currently being wrapped for Python and will eventually replace the IRBEM library as the default in SpacePy.

The IRBEM library

ONERA (Office National d'Etudes et Recherches Aerospatiales) provide a FORTRAN library, the IRBEM library [Bos07], that provides routines to compute magnetic coordinates for any location in the Earth's magnetic field, to perform coordinate conversions, to compute magnetic field vectors in geospace for a number of external field models, and to propagate satellite orbits in time.

A number of key routines in the IRBEM library have been wrapped uing f2py, and a 'thin layer' module *IrbemPy* has been written for easy access to these routines. Current functionality includes calls to calculate the local magnetic field vectors at any point in geospace, calculation of the magnetic mirror point for a particle of a given pitch angle (the angle between a particle's velocity vector and the magnetic field line that it immediately orbits such that a pitch angle of 90 degrees signifies gyration perpendicular to the local field) anywhere in geospace, and calculation of electron drift shells in the inner magnetosphere.

As mentioned in the description of the *Coordinates* module, access is also provided to the coordinate transformation capabilities of the IRBEM library. These can be called directly, but *IrbemPy* is easier to work with using `Coords` objects. This is by design as we aim to incorporate the LGM library and replace the calls to IRBEM with calls to LGM without any change to the *Coordinates* syntax.

OMNI

The OMNI database [Kin05] is an hourly resolution, multi-source data set with coverage from November 1963; higher temporal resolution versions of the OMNI database exist, but with coverage from 1995. The primary data are near-Earth solar wind, magnetic field and plasma parameters. However, a number of modern magnetic field models require derived input parameters, and [Qin07] have used the publicly-available OMNI database to provide a modified version of this database containing all parameters necessary for these magnetic field models. These data are currently updated and maintained by Dr. Bob Weigel and are available through ViRBO (Virtual Radiation Belt Observatory)[1].

In SpacePy this data is made available on request on install; if not downloaded when SpacePy is installed and attempt to import the *omni* module will ask the user whether they wish to download the data. Should the user require the latest data, the `update` function within `spacepy.toolbox` can be used to fetch the latest files from ViRBO.

As an example, we fetch the OMNI data for the powerful "Hallowe'en" storms of October and November, 2003. These geomagnetic storms were driven by two solar coronal mass ejections that reached the Earth on October 29th and November 20th.

```
>>> import spacepy.time as spt
>>> import spacepy.omni as om
>>> import datetime as dt
>>> st = dt.datetime(2003,10,20)
>>> en = dt.datetime(2003,12,5)
>>> delta = dt.timedelta(days=1)
>>> ticks = spt.tickrange(st, en, delta, `UTC')
>>> data = om.get_omni(ticks)
```

data is a dictionary containing all the OMNI data, by variable, for the timestamps contained within the `Ticktock` object *ticks*

Superposed Epoch Analysis

Superposed epoch analysis is a technique used to reveal consistent responses, relative to some repeatable phenomenon, in noisy data [Chr08]. Time series of the variables under investigation are extracted from a window around the epoch and all data at a given time relative to epoch forms the sample of events at that lag. The data at each time lag are then averaged so that fluctuations not consistent about the epoch cancel. In many superposed epoch analyses the mean of the data at each time u relative to epoch, is used to represent the central tendency. In SeaPy we calculate both the mean and the median, since the median is a more robust measure of central tendency and is less affected by departures from normality. SeaPy also calculates a measure of spread at each time relative to epoch when performing the superposed epoch analysis; the interquartile range is the default, but the median absolute deviation and bootstrapped confidence intervals of the median (or mean) are also available. The output of the example below is shown in figure 2.

```
>>> import spacepy.seapy as se
>>> import spacepy.omni as om
>>> import spacepy.toolbox as tb
>>> epochs = se.readepochs(`SI_GPS_epochs_OMNI.txt')
>>> st, en = datetime.datetime(2005,1,1),
...          datetime.datetime(2009,1,1)
>>> einds, oinds = tb.tOverlap([st, en],
...                            om.omnidata[`UTC'])
>>> omni1hr = array(om.omnidata[`UTC'])[oinds]
>>> delta = datetime.timedelta(hours=1)
```

1. http://virbo.org/QinDenton

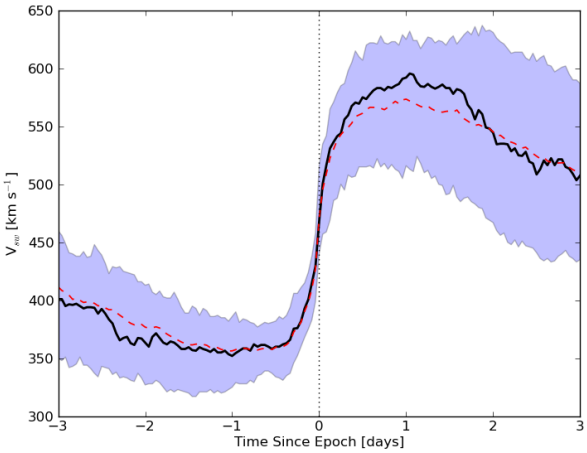


Fig. 2: *A typical output from the SpacePy Sea class using OMNI solar wind velocity data. The black line marks the superposed epoch median, the red dashed line marks the superposed epoch mean, and the blue fill marks the interquartile range. This figure was generated using the code in the text and a list of 67 events published by [Mor10].*

```
>>> window= datetime.timedelta(days=3)
>>> sevx = se.Sea(om.omnidata[`velo'][oinds],
...               omni1hr, epochs, window, delta)
>>> sevx.sea()
>>> sevx.plot(epochline=True, yquan=`V$_{sw}$',
              xunits=`days', yunits=`km s$^{-1}$')
```

More advanced features of this module have been used in analyses of the Van Allen radiation belts and can be found in the peer-reviewed literature [Mor10].

Association analysis

This module provides a point process class `PPro` and methods for association analysis (see, e.g., [Mor07]). This module is intended for application to discrete time series of events to assess statistical association between the series and to calculate confidence limits. Since association analysis is rather computationally expensive, this example shows timing. To illustrate its use, we here reproduce the analysis of [Wil09] using SpacePy. After importing the necessary modules, and assuming the data has already been loaded, `PPro` objects are instantiated. The association analysis is performed by calling the `assoc` method and bootstrapped confidence intervals are calculated using the `aa_ci` method. It should be noted that this type of analysis is computationally expensive and, though currently implemented in pure Python may be rewritten using Cython or C to gain speed.

```
>>> import datetime as dt
>>> import spacepy.time as spt
>>> onsets = spt.Ticktock(onset_epochs, `CDF')
>>> ticksR1 = spt.Ticktock(tr_list, `CDF')
>>> lags = [dt.timedelta(minutes=n)
...         for n in xrange(-400,401,2)]
>>> halfwindow = dt.timedelta(minutes=10)
>>> pp1 = poppy.PPro(onsets.UTC, ticksR1.UTC,
...                  lags, halfwindow)
>>> pp1.assoc()
>>> pp1.aa_ci(95, n_boots=4000)
>>> pp1.plot()
```

The output is shown in figure 3 and can be compared to figure 6a of [Wil09].

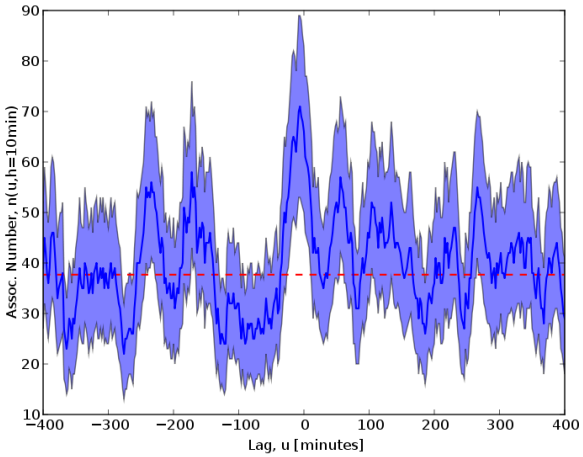


Fig. 3: *Reproduction of the association analysis done by [Wil09], using the* PoPPy *module of SpacePy. The figure shows a significant association around zero time lag between the two point processes under study (northward turnings of the interplanetary magnetic field and auroral substorm onsets).*

NASA Common Data Format

At the time of writing, limited support for NASA CDF[2] has been written in to SpacePy. NASA themselves have worked with the developers of both IDL™ and MatLab™. In addition to the standard C library for CDF, they provide a FORTRAN interface and an interface for Perl—the latest addition is support for C#. As Python is not supported by the NASA team, but is growing in popularity in the space science community we have written a module to handle CDF files.

The C library is made available in Python using *ctypes* and an object-oriented "thin layer" has been written to provide a Pythonic interface. For example, to open and query a CDF file, the following code is used:

```
>>> import spacepy.pycdf as cdf
>>> myfile = cdf.CDF()
>>> myfile.keys()
```

The CDF object inherits from the `collections.MutableMapping` object and provides the user a familiar 'dictionary-like' interface to the file contents. Write and edit capabilities are also fully supported, further development is being targeted towards the generation of ISTP-compliant CDF files[3] for the upcoming Radiation Belt Storm Probes (RBSP) mission.

As an example of this use, creating a new CDF from a master (skeleton) CDF has similar syntax to opening one:

```
>>> cdffile = cdf.CDF('cdf_file.cdf',
...                   'master_cdf_file.cdf')
```

This creates and opens `cdf_filename.cdf` as a copy of `master_cdf_filename.cdf`. The variables can then be populated by direct assignment, as one would populate any new object. Full documentation can be found both in the docstrings and on the SpacePy website.

2. http://cdf.gsfc.nasa.gov/
3. http://spdf.gsfc.nasa.gov/sp_use_of_cdf.html

Radiation belt modeling

Geosynchronous communications satellites are especially vulnerable to outer radiation belt electrons that can penetrate deep into the system and cause electrostatic charge buildup on delicate electronics. The complicated physics combined with outstanding operational challenges make the radiation belts an area of intense research. A simple yet powerful numerical model of the belts is included in SpacePy in the *RadBelt* module. This module allows users to easily set up a scenario to simulate, obtain required input data, perform the computation, then visualize the results. The interface is simple enough to allow users to easily include an analysis of radiation belt conditions in larger magnetospheric studies, but flexible enough to allow focused, in-depth radiation belt research.

The model is a radial diffusion model of trapped electrons of a single energy and a single pitch angle. The heart of the problem of radiation belt modeling through the diffusion equation is the specification of diffusion coefficients, source and loss terms. Determining these values is a complicated problem that is tackled in a variety of different ways, from first principles approaches to simpler empirical relationships. The *RadBelt* module approaches this with a paradigm of flexibility: while default functions that specify these values are given, many are available and additional functions are easy to specify. Often, the formulae require input data, such as the Kp or Dst indices. This is true for the *RadBelt* defaults. These data are obtained automatically from the OMNI database, freeing the user from the tedious task of fetching data and building input files. This allows simple comparative studies between many different combinations of source, loss, and diffusion models.

Use of the *RadBelt* module begins with instantiation of an `RBmodel` object. This object represents a version of the radial diffusion code whose settings are controlled by its various object attributes. Once the code has been properly configured, the time grid is created by specifying a start and stop date and time along with a step size. This is done through the `setup_ticks` instance method that accepts *datetime* or *Ticktock* arguments. Finally, the `evolve` method is called to perform the simulation, filling the `PSD` attribute with phase space densities for all L and times specified during configuration. The instance method `plot` yields a quick way to visualize the results using matplotlib functionality. The example given models the phase space density during the "Hallowe'en" storms of 2003. The results are displayed in figure 4. In the top frame, the phase space density is shown. The white line plotted over the spectrogram is the location of the last closed drift shell, beyond which the electrons escape the magnetosphere. Directly below this frame is a plot of the two geomagnetic indices, Dst and Kp, used to drive the model. With just a handful of lines of code, the model was setup, executed, and the results were visualized.

```
>>> from spacepy import radbelt as rb
>>> import datetime as dt
>>> r = rb.RBmodel()
>>> starttime = dt.datetime(2003,10,20)
>>> endtime   = dt.datetime(2003,12,5)
>>> delta = dt.timedelta(minutes=60)
>>> r.setup_ticks(starttime, endtime,
...               delta, dtype=`UTC')
>>> r.evolve()
>>> r.plot(clims=[4,11])
```

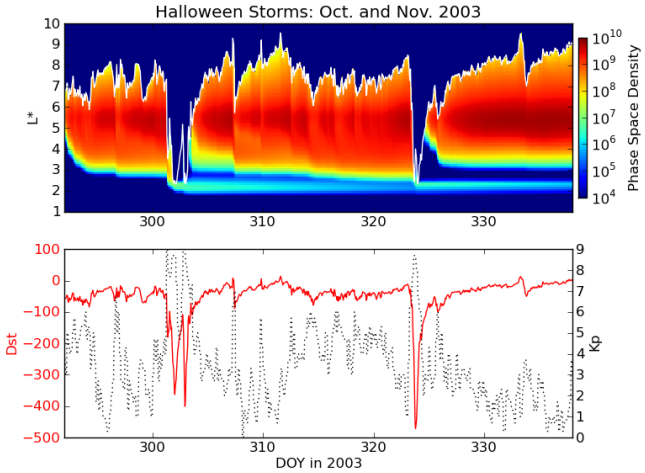


Fig. 4: *RadBelt simulation results for the 2003 Hallowe'en storms. The top frame shows phase space density as a function of drift shell and time. The bottom frame shows the geomagnetic Kp and Dst indices during the storm.*

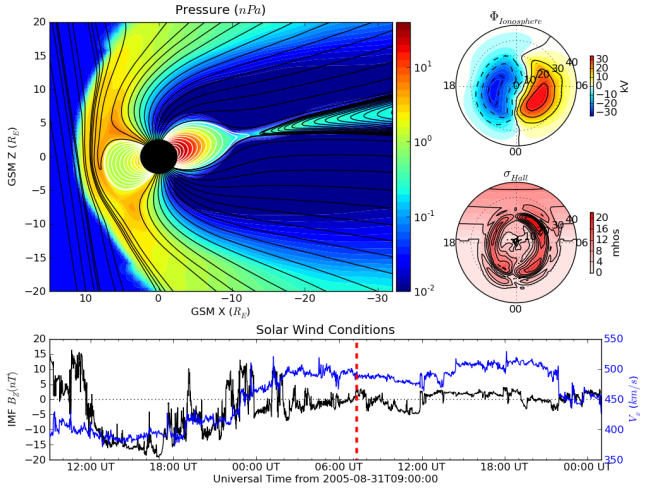


Fig. 5: *Typical output desired by users of BATS-R-US and the SWMF. The upper left frame is a cut through the noon-midnight meridian of the magnetosphere as simulated by BATS-R-US at 7:15 UT on September 1, 2005. The dial plots to the left are the ionospheric electric potential and Hall conductivity at the same time as calculated by RIM. Below are the solar wind conditions driving both models.*

Visualizing space weather models

The Block Adaptive Tree Solar wind Roe-type Upwind Scheme code, or BATS-R-US, is a widely used numerical model in the space science community. It is a magnetohydrodynamic (MHD) code [Pow99], which means it combines Maxwell's equations for electromagnetism with standard fluid dynamics to produce a set of equations suited to solving spatially large systems while using only modest computational resources. It is unique among other MHD codes in the space physics community because of its automatic grid refinement, compile-time selection of many different implementations (including multi fluid, Hall resistive, and non-isotropic MHD), and its library of run-time options (such as solver and scheme configuration, output specification, and much more). It has been used in a plethora of space applications, from planetary simulations (including Earth [Wel10b] and Mars [Ma07]) to solar and interplanetary investigations [Coh09]. As a key component of the Space Weather Modeling Framework (SWMF) [Tot07], it has been coupled to many other space science numerical models in order to yield a true 'sun to mud' simulation suite that handles each region with the appropriate set of governing equations.

Visualizing output from the BATS-R-US code comes with its own challenges. Good analysis requires a combination of two and three dimensional plots, the ability to trace field lines and stream lines through the domain, and the slicing of larger datasets in order to focus on regions of interest. Given that BATS-R-US is rarely used by itself, it is also important to be able to visualize output from the coupled codes used in conjunction. Professional computational fluid dynamic visualization software solutions excel at the first points, but are prohibitively expensive and often leave the user searching for other solutions when trying to combine the output from all SWMF modules into a single plot. Scientific computer languages, such as IDL™ and MatLab™, are flexible enough to tackle the latter issue, but do not contain the proper tools required by fluid dynamic applications. Because all of these solutions rely on proprietary software, there are always license fees involved before plots can be made.

The *PyBats* package of SpacePy attempts to overcome these difficulties by providing a free, platform independent way to read and visualize BATS-R-US output as well as output from models that are coupled to it. It builds on the functionality of NumPy and matplotlib to provide specialized visualization tools that allow the user to begin evaluating and exploring output as quickly as possible.

The core functionality of *PyBats* is a set of classes that read and write SWMF file formats. This includes simple ASCII log files, ASCII input files, and a complex but versatile self-descriptive binary format. Because many of the codes that are integrated into the SWMF use these formats, including BATS-R-US, it is possible to begin work right away with these classes. Expanded functionality is found in code-specific modules. These contain classes to read and write output files, inheriting from the *PyBats* base classes when possible. Read/write functionality is expanded in these classes through object methods for plotting, data manipulation, and common calculations.

Figure 5 explores the capabilities of *PyBats*. The figure is a typical medley of desired output from a basic simulation that used only two models: BATS-R-US and the Ridley Ionosphere Model. Key input data that drove the simulation is shown as well. Creating the upper left frame of figure 5, a two dimensional slice of the simulated magnetosphere saved in the SWMF binary format, would require far more work if the base classes were chosen. The *bats* submodule expands the base capability and makes short work of it. Relevant syntax is shown below. The file is read by instantiating a *Bats2d* object. Inherited from the base class is the ability to automatically detect bit ordering and the ability to carefully walk through the variable-sized records stored in the file. The data is again stored in a dictionary as is grid information; there is no time information for the static output file. Extra information, such as simulation parameters and units, are also placed into object attributes. The unstructured grid is not suited for matplotlib, so the object method *regrid* is called. The object remembers that it now has a regular grid; all data and grid vectors are now two dimensional arrays. Because this is a computationally expensive step, the regridding is performed to a resolution of 0.25 Earth

radii and only for a subset of the total domain. The object method `contourf`, a wrapper to the matplotlib method of the same name, is used to add the pressure contour to an existing axis, `ax`. The wrapped function accepts keys to the *grid* and *data* dictionaries of the *Bats2d* object to prevent the command from becoming overly verbose. Extra keyword arguments are passed to matplotlib's `contourf` method. If the original file contains the size of the inner boundary of the code, this is reflected in the object and the method `add_body` is used to place it in the plot.

```
>>> import pybats.bats as bats
>>> obj = bats.Bats2d(`filename')
>>> obj.regrid(0.25, [-40, 15], [-30,30])
>>> obj.contourf(ax, `x', `y', `p')
>>> obj.add_body(ax)
>>> obj.add_planet_field(ax)
```

The placement of the magnetic field lines is a strength of the *bats* module. Magnetic field lines are simply streamlines of the magnetic field vectors and are traced through the domain numerically using the Runge-Kutta 4 method. This step is implemented in C to expedite the calculation and wrapped using f2py. The `Bats2d` method `add_planet_field` is used to add multiple field lines; this method finds closed (beginning and ending at the inner boundary), open (beginning or ending at the inner boundary, but not both), or pure solar wind field lines (neither beginning or ending at the inner boundary) and attempts to plot them evenly throughout the domain. Closed field lines are colored white to emphasize the open-closed boundary. The user is naive to all of this, however, as one call to the method works through all of the steps.

The last two plots, in the upper right hand corner of figure 5, are created through the code-specific *rim* module, designed to handle output from the Ridley Ionosphere Model (RIM) [Rid02].

PyBats capabilities are not limited to what is shown here. The `Stream` class can extract values along the streamline as it integrates, enabling powerful flow-aligned analysis. Modules for other codes coupled to BATS-R-US, including the Ring current Atmosphere interactions Model with Self-Consistent Magnetic field (RAM-SCB, *ram* module) and the Polar Wind Outflow Model (PWOM, *pwom* module) are already in place. Tools for handling virtual satellites (output types that simulate measurements that would be made if a suite of instruments could be flown through the model domain) have already been used in several studies. Combining the various modules yields a way to richly visualize the output from all of the coupled models in a single language. *PyBats* is also in the early stages of development, meaning that most of the capabilities are yet to be developed. Streamline capabilities are currently being upgraded by adding adaptive step integration methods and advanced placement algorithms. `Bats3d` objects are being developed to complement the more frequently used two dimensional counterpart. A GUI interface is also under development to provide users with a point-and-click way to add field lines, browse a time series of data, and quickly customize plots. Though these future features are important, *PyBats* has already become a viable free alternative to current, proprietary solutions.

SpacePy in action

A number of key science tasks undertaken by the SpacePy team already heavily use SpacePy. Some articles in peer-reviewed literature have been primarily produced using the package (e.g. [Mor10], [Wel10a]). The Science Operations Center for the RBSP mission is also incorporating SpacePy into its processing stream.

The tools described here cover a wide range of routine analysis and visualization tasks utilized in space science. This software is currently available on a non-commercial research license, but the process to release it as free and open-source software is underway. Providing this package in Python makes these tools accessible to all, provides openness in the analysis methods employed in scientific studies and will give access to advanced tools to all space scientists regardless of affiliation or circumstance. The SpacePy team can be contacted at spacepy-info@lanl.gov.

References

[Bos07] D. Boscher, S. Bourdarie, P. O'Brien and T. Guild *ONERA-DESP library v4.1*, http://irbem.sourceforge.net/, 2007.

[Chr08] C. Chree *Magnetic declination at Kew Observatory, 1890 to 1900*, Phil. Trans. Roy. Soc. A, 208, 205–246, 1908.

[Coh09] O. Cohen, I.V. Sokolov, I.I. Roussev, and T.I. Gombosi *Validation of a synoptic solar wind model*, J. Geophys. Res., 113, 3104, doi:10.1029/2007JA012797, 2009.

[Kin05] J.H. King and N.E. Papitashvili *Solar wind spatial scales in and comparisons of hourly Wind and ACE plasma and magnetic field data*, J. Geophys. Res., 110, A02209, 2005.

[Ma07] Y.J. Ma, A.F. Nagy, G. Toth, T.E. Cravens, C.T. Russell, T.I. Gombosi, J.-E. Wahlund, F.J. Crary, A.J. Coates, C.L. Bertucci, F.M. Neubauer *3D global multi-species Hall-MHD simulation of the Cassini T9 flyby*, Geophys. Res. Lett., 34, 2007.

[Mly10] M.G. Mlynczak, L.A. Hunt, J.U. Kozyra, and J.M. Russell III *Short-term periodic features observed in the infrared cooling of the thermosphere and in solar and geomagnetic indexes from 2002 to 2009* Proc. Roy. Soc. A, doi:10.1098/rspa.2010.0077, 2010.

[Mor07] S.K. Morley and M.P. Freeman *On the association between northward turnings of the interplanetary magnetic field and substorm onset*, Geophys. Res. Lett., 34, L08104, 2007.

[Mor10] S.K. Morley, R.H.W. Friedel, E.L. Spanswick, G.D. Reeves, J.T. Steinberg, J. Koller, T. Cayton, and E. Noveroske *Dropouts of the outer electron radiation belt in response to solar wind stream interfaces: Global Positioning System observations*, Proc. Roy. Soc. A, doi:10.1098/rspa.2010.0078, 2010.

[Pow99] K. Powell, P. Roe, T. Linde, T. Gombosi, and D.L. De Zeeuw *A solution-adaptive upwind scheme for ideal magnetohydrodynamics*, J. Comp. Phys., 154, 284-309, 1999.

[Qin07] Z. Qin, R.E. Denton, N. A. Tsyganenko, and S. Wolf *Solar wind parameters for magnetospheric magnetic field modeling*, Space Weather, 5, S11003, 2007.

[Rid02] A.J. Ridley and M.W. Liemohn *A model-derived storm time asymmetric ring current driven electric field description* J. Geophys. Res., 107, 2002.

[Tot07] Toth, G., D.L.D. Zeeuw, T.I. Gombosi, W.B. Manchester, A.J. Ridley, I.V. Sokolov, and I.I. Roussev *Sun to thermosphere simulation of the October 28-30, 2003 storm with the Space Weather Modeling Framework*, Space Weather, 5, S06003, 2007.

[Vai09] R. Vainio, L. Desorgher, D. Heynderickx, M. Storini, E. Fluckiger, R.B. Horne, G.A. Kovaltsov, K. Kudela, M. Laurenza, S. McKenna-Lawlor, H. Rothkaehl, and I.G. Usoskin *Dynamics of the Earth's Particle Radiation Environment*, Space Sci. Rev., 147, 187--231, 2007.

[Wel10a] D.T. Welling, and A.J. Ridley *Exploring sources of magnetospheric plasma using multispecies MHD*, J. Geophys. Res., 115, 4201, 2010.

[Wel10b] D.T. Welling, V. Jordanova, S. Zaharia, A. Glocer, and G. Toth *The effects of dynamic ionospheric outflow on the ring current*, Los Alamos National Laboratory Technical Report, LA-UR 10-03065, 2010.

[Wil09] J.A. Wild, E.E. Woodfield, and S.K. Morley, *On the triggering of auroral substorms by northward turnings in the interplanetary magnetic field*, Ann. Geophys., 27, 3559-3570, 2009.

Numerical Pyromaniacs: The Use of Python in Fire Research

Kristopher Overholt‡*

Abstract—Python along with various numerical and scientific libraries was used to create tools that enable fire protection engineers to perform various calculations and tasks including educational instruction, experimental work, and data visualization. Examples of web calculators and tools for creating 3D geometry for fire models using Python are presented. The use of Python in the fire research field will foster many new ideas, tools, and innovation in the field of fire protection research and engineering.

Index Terms—fire protection engineering, fluid dynamics, visualization

Introduction

The use of Python in fire protection engineering and fire research has many useful applications and allows for an abundance of possibilities for the fire science community. Recently, Python has been used to create data analysis tools for fire experiments and simulation data, to assist users with fire modeling and fire dynamics calculations, to perform flame height tracking for warehouse fires and bench-scale tests using GUI programs, and to present flame spread results using a visual method of superimposed video plotting. These tools have emerged in the form of web applications, GUIs, and data reduction scripts which interact with existing computational fluid dynamics (CFD) tools such as the freely available and open-source fire modeling program, Fire Dynamics Simulator [FDS], which is maintained by the National Institute of Standards and Technology (NIST).

Python (and its associated scientific and numerical libraries) is the perfect tool for the development of these tools, which advance the capabilities of the fire research community, due to its free and open source nature and widespread, active community. Thus, it is important to identify efforts and projects that allow engineers and scientists to easily dive into learning Python programming and to utilize it in an environment that is familiar and efficient for practical engineering use.

Along this line, tools and packages such as the Enthought Python Distribution [EPD] and Python(x,y) [pythonxy] help with the installation of Python and its associated scientific and numerical packages. Other tools such as Spyder [SPY] and Sage [SAGE] allow engineers and students to work in a familiar environment that is similar to commercial engineering programs with the advantage of a full-featured programming language that is based on an open-source foundation.

Another application of Python in fire research is for the verification and validation of fire models. As with other CFD tools, the developers of fire models have recently become interested in the use of verification and validation to assess the performance and accuracy of the models regarding their correct physical representation. The process of validation compares the results from the computer model to analytical results or experimental data and involves a large amount of data and plots. For this purpose, we can use Python to automate our workflow and generate plots from updated data, and we are considering using Python to generate regression statistics between newer versions of the fire modeling software so that we can better quantify the numerical variations that are related to changes in the CFD code.

From data fitting to creating publication-quality plots using matplotlib [MPL], Python is quickly emerging as a useful tool for engineers and scientists to interact with data in a meaningful and programmatic manner that is familiar to them. In a time when a budget cut can result in numerous dropped licenses for proprietary data analysis software (which can cause the proprietary scripts and data sets to be unusable), Python is in the position to become more ubiquitous in fire protection engineering and the engineering field in general.

Superimposed Video Plotting Method

A few years ago, while I was attending a fire conference, a presenter showed a video with real-time seismic data plots superimposed over a video of the burning building where the measurements were being taken from. This was certainly a useful way to convey multiple streams of information visually and allow the data tell a story. Because videos can convey much more information than static plots, this method allows for both qualitative and quantitative information to be communicated simultaneously.

Shortly thereafter, I created a Python script and used matplotlib to import videos, adjust frame rates, plot on the imported figures, and export the video frames with plots. I used the script to generate video plots of warehouse commodity fire tests with actual and predicted flame heights vs. time, as shown in Figure 1.

I also used the script to show video plots of the predicted flame heights for small-scale tests, as shown in Figure 2.

Real-time video plots are a great visual method for teaching, communication, and telling a story with your data. At the time of this writing, no existing programs or tools are available for this process. Python was a great tool for this purpose, and it would not require much more effort to create a GUI for this tool.

* *Corresponding author: koverholt@gmail.com*
‡ *University of Texas at Austin*

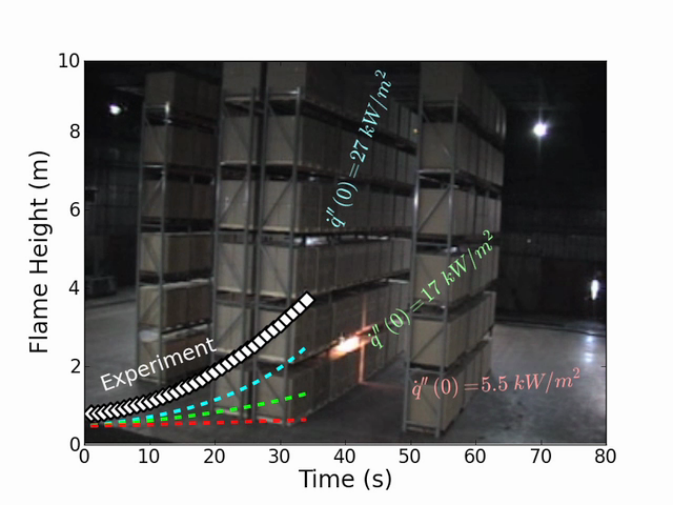


Fig. 1: *Plot of flame height vs. time on a warehouse fire video created with Python and matplotlib.*

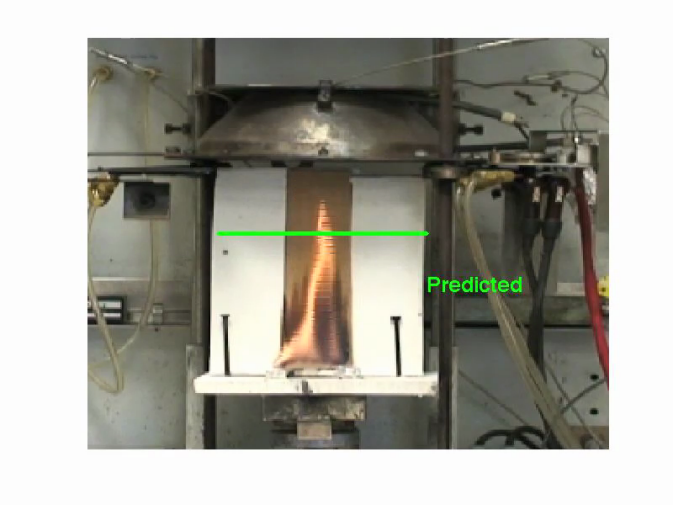


Fig. 2: *Plot of flame height vs. time on a bench-scale experiment video created with Python and matplotlib.*

Web Tools and Engineering Calculators

As I was learning Python a few years ago and using it in my research, I created a spreadsheet tool that would generate text input lines for a 3D mesh for a fire modeling tool (FDS) that involved tedious calculations and Poisson-friendly integer requirements. After repeatedly running these calculations by hand one, I created a Python script to generate the appropriate numbers and parameters to use in a text input file, and I wanted to share this tool with others. Based on some simple Python CGI examples, I created a mesh size calculator web tool. Previously, I had never created a web application, but with Python it was quite easy and fun. The interface for the web calculator is shown in Figure 3.

Today, on my website [FDSmesh], the mesh calculator web tool gets used about 1,000 times a month by engineers and scientists around the world. The source code of the web tool is freely available on Google Code under the MIT License and is linked from the webpage that contains the web calculator. Because the source code is available, this will hopefully be helpful to others who want to create a simple web calculator tool using Python. The output of the web calculator is shown in Figure 4.

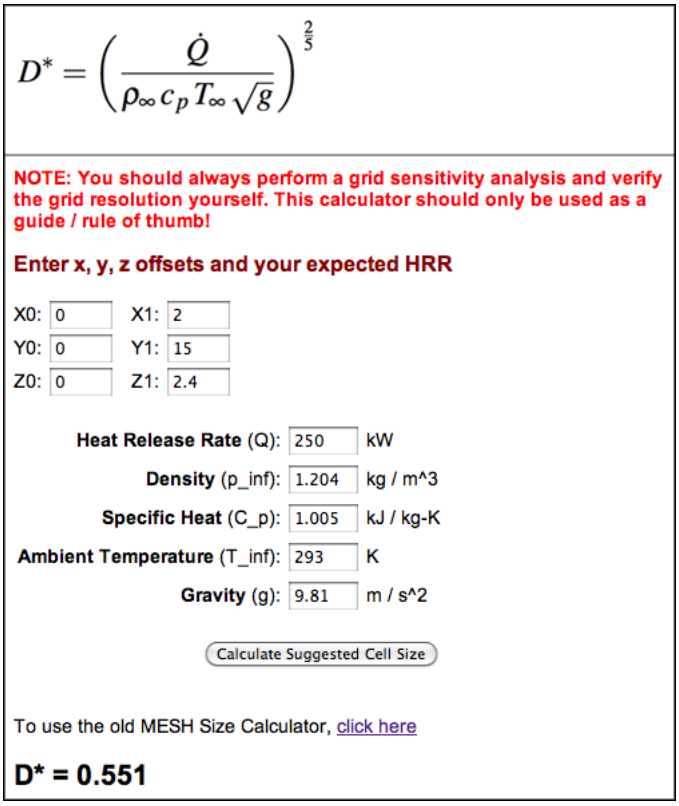


Fig. 3: *Interface for FDS mesh size calculator web tool.*

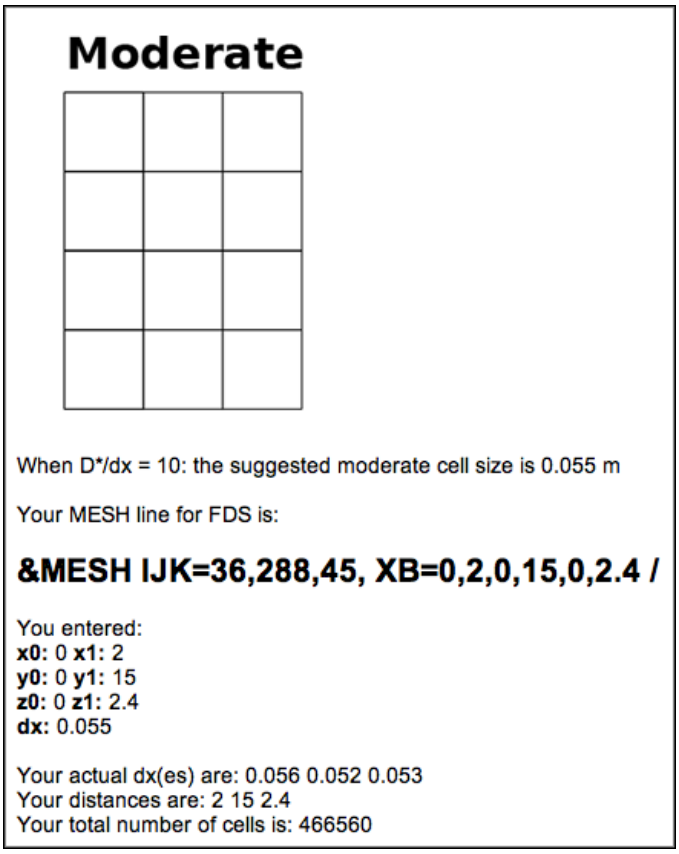


Fig. 4: *Results from FDS mesh calculator web tool.*

Since then, I have also developed a few other web calculators. Some future tools that I wish to develop include a suite of fire engineering and fire dynamics tools that can be used online. A legacy computer tool for fire protection engineers is a program called FPETool (fire protection engineering tool) [FPETool], which contains a set of fire dynamics calculations, and this program was heavily used in the 1980s and 1990s. FPETool is still available as a free download from NIST, but only as a DOS executable. Because of this, the suite of tools and fire dynamics calculators in FPEtool are no longer used in the field. The equations and methods in FPEtool could be recreated as a web-based, open-source, and community-supported project using Python. Python offers our field the ability to easily and quickly create web tools, from simple calculators to complex web applications, and this results in a more efficient workflow for engineers, a method for third-party developers to contribute to the fire modeling community, and promotion of the effective use of fire dynamics and tools for life safety design work.

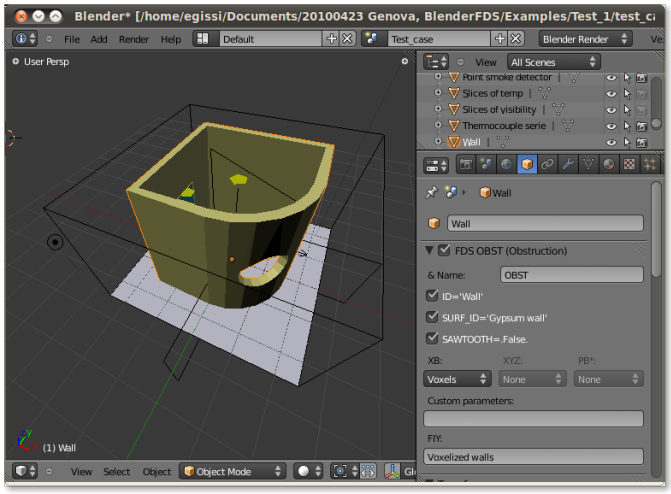


Fig. 5: *Interface for creating and exporting 3D fire model geometry in Blender.*

Creating 3D Geometry for Fire Models

Regarding the increasing amount of interaction between Python and fire models, third-party developers in the fire modeling community (including myself) have recently released a tool to model 3D geometry and generate a text-based input file for the FDS fire modeling software. The tool is called BlenderFDS and is an extension for [Blender] that was written in Python. Before the release of BlenderFDS, users of FDS had to create geometry for a case either manually using a text editor or by using a commercial user interface. Now, using BlenderFDS, FDS users can create complex buildings and irregular geometry (e.g., cylinders, angled roofs) and automatically have the geometry broken up into the rectilinear format that FDS requires.

Blender handles the interactive creation and manipulation of 3D objects, and BlenderFDS then voxelizes the 3D geometry into rectilinear shapes and outputs a text-based FDS input file. BlenderFDS works by scanning the geometry in Blender on the x, y, and z axis and then generating optimized obstruction lines with 3D coordinates in ASCII format. Using this method, complex objects can be represented as multiple lines of simplified geometry in the FDS input file. This approach could be used in other fields that utilize 3D modeling to help with the creation of input files. The interface for the BlenderFDS extension in Blender is shown in Figure 5.

BlenderFDS allows for the quick creation of complex geometry in a visual manner, and it can even be used to model an entire building, as shown in Figure 6.

We hope to continue adding functionality to BlenderFDS and create a comprehensive GUI for creating input files for fire models, and we (the developers) have appreciated the ease of use and the implementation process of using Python with Blender for this project. The source code for the BlenderFDS project is freely available on the [BlenderFDS] website on Google Code and is licensed under the GNU GPL. We are also exploring additional 3D fire modeling solutions in Blender and other popular CFD postprocessing tools, which will be discussed in the next section.

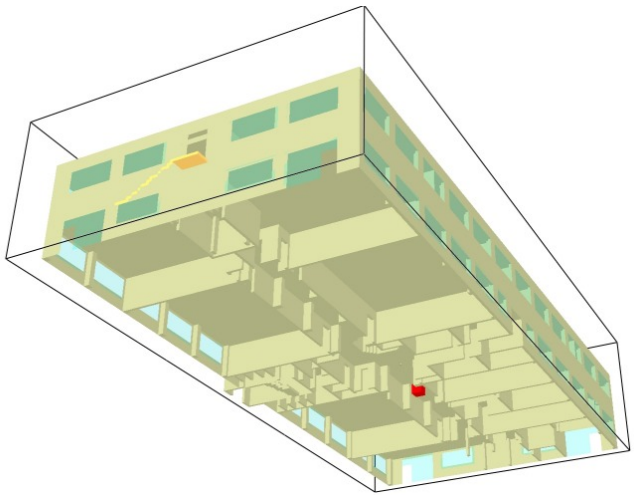


Fig. 6: *3D geometry output shown in FDS that was generated using the BlenderFDS plugin.*

Visualizing Smoke and Fire for CFD simulations

With the availability of numerous CFD-related tools such as [Paraview], [Mayavi], and Blender, we have been exploring the use of these tools for the visualization of realistic and scientifically-based fire and smoke for 3D fire simulations. An example of the improved rendering of fire and smoke in the upcoming release of Blender 2.5 is shown in Figure 7.

Such a visualization tool would allow for graphical improvements in the output and a standardized data format for visualization and analysis for CFD tools. Finally, such a tool would also allow for more community involvement and support for the visualization software.

Future Plans for Python in Fire Research

The use of Python in fire protection engineering is still in its early stages; future applications in the fire research field include additional fire dynamics and engineering web calculation tools, tools to analyze and visualize output from CFD programs such as FDS, and the design and implementation of a standardized, open format for experimental fire test data.

Interactive data analysis tools that are based on Python, such as Spyder and Sage, will allow Python to be used more in the engineering field as a flexible, free, and powerful tool with a supportive and active community. For Python to be used more in

Fig. 7: *Realistic flames and smoke rendered in Blender [Price].*

the engineering field as a replacement for commercial tools, more emphasis should be placed on the development of interactive data analysis and GUI tools.

Python can also be utilized more in tools such as Blender (for geometry creation), Spyder (for interactive data analysis and scripting), or Mayavi (for visualization), which allows for the possibility of many new innovations in fire research. Additionally, Python can be incorporated into the field of CFD and high performance computing.

In conclusion, the use of Python in fire protection engineering and fire research is of utmost importance because these fields involve public safety and strive to produce safer buildings and materials to protect people and property around the world from the dangers of fire. Python and the scientific Python community are a good fit for this endeavor, and I hope to interact and learn more from the Python community to create additional solutions that can advance our field.

REFERENCES

[FDS] http://fire.nist.gov/fds
[EPD] http://www.enthought.com/products/epd.php
[pythonxy] http://code.google.com/p/pythonxy/
[SPY] http://code.google.com/p/spyderlib/
[SAGE] http://www.sagemath.org/
[MPL] http://matplotlib.sourceforge.net/
[FDSmesh] http://www.koverholt.com/fds-mesh-size-calc
[FPETool] http://www.bfrl.nist.gov/866/fmabbs.html#FPETOOL
[Blender] http://www.blender.org/
[BlenderFDS] http://www.blenderfds.org
[Paraview] http://www.paraview.org/
[Mayavi] http://mayavi.sourceforge.net/
[Price] Used with permission from Andrew Price, http://www.blenderguru.com

A Programmatic Interface for Particle Plasma Simulation in Python

Min Ragan-Kelley‡*, John Verboncoeur‡

Abstract—Particle-in-Cell (PIC) simulations are a popular approach to plasma physics problems in a variety of applications. These simulations range from interactive to very large, and are well suited to parallel architectures, such as GPUs. PIC simulations frequently serve as input to other simulations, as a part of a larger system. Our project has two goals: facilitate exploitation of increasing availability of parallel compute resources in PIC simulation, and provide an intuitive and efficient programmatic interface to these simulations. We plan to build a modular backend with multiple levels of parallelism using tools such as PyCUDA/PyOpenCL and IPython. The modular design, following the goals of our Object-Oriented Particle-in-Cell (OOPIC) code this is to replace, enables comparison of multiple algorithms and approaches. On the frontend, we will use a runtime compilation model to generate an optimized simulation based on available resources and input specification. Maintaining NumPy arrays as the fundamental data structure of diagnostics will allow users great flexibility for data analysis, allowing the use of many existing powerful tools for Python, as well as the definition of arbitrary derivative diagnostics in flight. The general design and preliminary performance results with the PyCUDA backend will be presented. This project is early in development, and input is welcome.

Index Terms—simulation, CUDA, OpenCL, plasma, parallel

Introduction

Plasma physics simulation is a field with widely varied problem scales. Some very large 3D problems are long term runs on the largest supercomputers, and there are also many simple prototyping and demonstration simulations that can be run interactively on a single laptop. Particle-in-Cell (PIC) simulation is one common approach to these problems. Unlike full particle simulations where all particle-particle Coulomb interactions are computed, or fully continuous simulations where no particle interactions are considered, the PIC model has arrays of particles in continuous space and their interactions are mediated by fields defined on a grid. Thus, a basic PIC simulation consists of two base data structures and three major computation kernels. The data structures are one (or more) list(s) of particles and the grid problem, with the source term and fields defined at discrete locations in space. The first kernel (Weight) is weighing the particle contributions to the source term on the grid. The second kernel (Solve) updates the fields on the grid from the source term by solving Poisson's Equation or Maxwell's equations. The third kernel (Push) updates the position and velocity of the particles based on the field values on the grid, which involves interpolating field values from the grid locations to the particle positions.

Our background in PIC is developing the Object Oriented Particle in Cell (OOPIC) project [OOPIC]. The motivation for OOPIC is developing an extensible interactive plasma simulation with live plotting of diagnostics. As with OO programming in general, the goal of OOPIC was to be able to develop new components (new Field Solvers, Boundary Conditions, etc.) with minimal change to existing code. OOPIC has been quite successful for small to moderate simulations, but has many shortcomings due to the date of the design (1995). The only way to interact with OOPIC is a mouse-based Tk interface. This makes interfacing OOPIC simulations with other simulation codes (a popular desire) very difficult. By having a Python frontend replace the existing Tk, we get a full interactive programming environment as our interface. With such an environment, and NumPy arrays as our native data structure, our users are instantly flexible to use the many data analysis and scripting tools available in Python and from the SciPy community [NumPy], [SciPy].

The performance component of the design is to use code generation to build the actual simulation program as late as possible. The later the program is built, the fewer assumptions made, which allows our code as well as compilers to maximize optimization. With respect to OOPIC, it also has the advantage of putting flexible control code in Python, and simpler performance code in C/CUDA, rather than building a single large C++ program with simultaneous goals of performance and flexibility.

Modular Design

1. Input files are Python Scripts.

With OOPIC, simulations are specified by an input file, using special syntax and our own interpreter. An input file remains, but it is now a Python script. OOPIC simulations are written in pure Python and interpreted in a private namespace, which allows the user to build arbitrary programming logic into the input file itself, which is very powerful.

2. Interfaces determine the simulation construction.

The mechanism for building a `Device` object from an input file follows an interface-based design, via `zope.interface` [Zope]. The constructor scans the namespace in which the input file was executed for object that provide our interfaces and performs the appropriate construction. This allows users to extend our functionality without altering our package, thus supporting new or proprietary components.

* *Corresponding author: minrk@berkeley.edu*
‡ *University of California, Berkeley*

3. Python Objects generate C/CUDA kernels or code snippets.

Once the `Device` has been fully specified, the user invokes a compile method, which prompts the device to walk through its various components to build the actual simulation code. In the crudest cases, this amounts to simply inserting variables and code blocks in code templates and compiling them.

4. The simulation is run interactively

The primary interface is to be an IPython session [IPython]. Simple methods, such as `run()` will advance the simulation in time, `save()` dumps the simulation state to a file. But a method exposing the full power of a Python interface is `runUntil()`. `runUntil()` takes a function and argument list, and executes the function at a given interval. The simulation continues to run until the function returns true. Many PIC simulations are run until a steady state is achieved before any actual analysis is done. If the user can quantify the destination state in terms of the diagnostics, then runUntil can be used to evolve the system to a precisely defined point that may occur at an unknown time.

5. Diagnostics can be fetched, plotted, and declared on the fly

All diagnostics are, at their most basic level, exposed to the user as NumPy arrays. This allows the use of all the powerful data analysis and plotting tools available to Python users. Since the simulation is dynamic, diagnostics that are not being viewed are not computed. This saves on the communication and possible computation cost of moving/analyzing data off of the GPU. The Device has an attribute Device.diagnostics, which is a dict of all available diagnostics, and a second list, Device.activeDiagnostics, which is only the diagnostics currently being computed. Users can define new diagnostics, either through the use of provided methods, such as `cross(A,B)`, which will efficiently add a diagnostic that computes the cross product two diagnostics, or even the fully arbitrary method of passing a function to the constructor of `DerivativeDiagnostic`, which will be evaluated each time the diagnostic is to be updated. This can take a method, and any other diagnostics to be passed to the function as arguments. This way, perfectly arbitrary diagnostics can be registered, even if it does allow users to write very slow functions to be evaluated in the simulation loop.

Interfaces

Interface based design, as learned developing the parallel computing kernel of IPython, provides a good model for developing pluggable components. Each major component presents an interface. For instance, a whole simulation presents the interface IDevice. New field solvers present ISolver, all diagnostics present a simple set of methods in IDiagnostic, and more specific diagnostic groups provide extended sets, such as ITimeHistory and IFieldDiagnostic. Some common interface elements are provided below.

IDiagnostic

IDiagnostic provides the basic interface common to all Diagnostics:

- `save()`: save the data to a file, either ascii or `numpy.tofile()`
- `data`: a NumPy array, containing the data
- `interval`: an integer, the interval at which the Diagnostic's data is to be updated

IDevice

IDevice is the full simulation interface:

- `save(fname)`: dumps the full simulation state to a file
- `restore(fname)`: reciprocal of save()
- `run(steps=None)`: run either continuously, or a specified number of steps
- `step()`: equivalent to run(1)
- `runUntil(interval, f, args)`: run in batches of interval steps until f(*args) returns True.
- `diagnostics`: a list of diagnostics available
- `activeDiagnostics`: a list of diagnostics currently being evaluated
- `addDiagnostic(d)`: registers a new diagnostic to be computed, such as derivative diagnostics

Diagnostics

Diagnostics will have two classes. First class diagnostics are fast, native diagnostics, computed as a part of the compute kernel in C/CUDA. The second class of diagnostics, Derivative Diagnostics, are more flexible, but potential performance sinks because users can define arbitrary new diagnostics interactively, which can be based on any Python function.

PyCUDA tests

We built a simple test problem with PyCUDA [PyCUDA]. It is a short-range n-body particle simulation where particles interact with each other within a cutoff radius. The density is controlled, such that each particle has several (~10) interactions. The simulation was run on two NVIDIA GPUs (C1060 and GTX 260-216) with various numbers of threads per block (tpb) [C1060], [GTX260]. This was mainly a test of simple data structures, and we found promising performance approaching 40% of the theoretical peak performance on the GPUs in single precision [Figure 1].

The sawtooth pattern in Figure 1 is clarified by plotting a normalized runtime of the same data [Figure 2]. The runtime plot reveals that adding particles does not increase the runtime until a threshold is passed, because many particles are computed in parallel. The threshold is that number of particles. Since there is one particle per thread, the steps are located at intervals of the number of threads-per-block (tpb) times the number of blocks that can be run at a time (30 for C1060, and 27 for GTX-260).

Challenges

There are a few points where we anticipate challenges in this project.

First, and most basic, is simply mapping PIC to the GPU. Ultimately we intend to have backends for multi-machine simulations leveraging both multicore CPUs and highly parallel GPUs, likely with a combination of OpenCL and MPI. However, the first backend is for 1 to few NVidia GPUs with CUDA/PyCUDA. This is a useful starting point because the level of parallelism for modestly sized problems is maximized on this architecture. We should encounter many of the data structure and API issues involved. PIC is primarily composed of two problems: grid-based field solve, and many particle operations. Both of these models

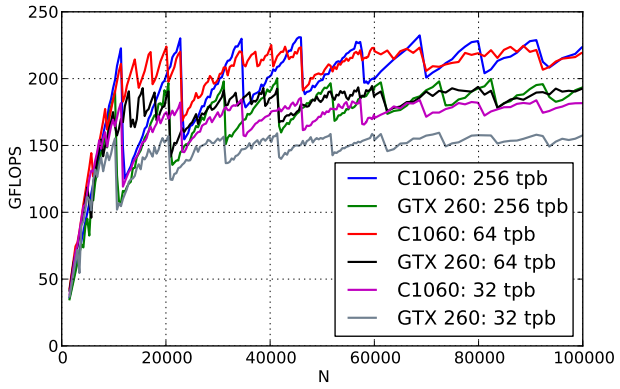


Fig. 1: *FP performance vs number of particles in the simulation (N). 230 GFLOPS is 37% of the 622 GFLOPS theoretical peak of a C1060, when not using dual-issue MAD+MUL. 'tpb' indicates threads-per-block - the number of threads allowed in each threadblock.*

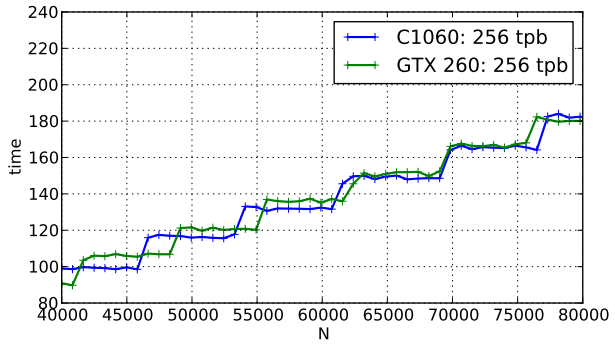


Fig. 2: *Normalized runtime increases at discrete steps of tbp* # of blocks: 256*30=7680 for C1060, and 256*27=6912 for GTX-260.*

are popular to investigate on GPUs, but there is still much to be learned about the coupling of the two.

Diagnostics also pose a challenge because it is important that computing and displaying diagnostics not contribute significantly to execution time. Some target simulations run at interactive speeds, and an important issue to track when writing Python code in general, and particularly multi-device code, is data copying.

Code generation is another challenge we face. Our intention is to build a system where the user specifies as little of the backend as possible. They enter the physics, and likely the spatial and time resolution, and our Python code generates C+CUDA code that will run efficiently. This is not easily done, but once complete will be quite valuable.

Future Plans

Ultimately we intend to have a GUI, likely built with Chaco/ETS, to replicate and extend functionality in OOPIC, as well as extending backends to fully general hardware [ETS]. But for now, there is plenty of work to do exploring simpler GPU simulations and code generation strategies behind the interactive Python interface.

The code will be licensed under the GNU Public License (GPL) once it is deemed ready for public use [GPL].

References

[OOPIC] J.P. Verboncoeur, A.B. Langdon and N.T. Gladd, *An Object-Oriented Electromagnetic PIC Code*, Comp. Phys. Comm., 87, May11, 1995, pp. 199-211.
[NumPy] http://numpy.scipy.org
[SciPy] http://www.scipy.org
[Zope] http://www.zope.org/Products/ZopeInterface
[IPython] http://ipython.scipy.org
[PyCUDA] http://mathema.tician.de/software/PyCUDA
[GTX260] http://www.nvidia.com/object/product_geforce_gtx_260_us.html
[C1060] http://www.nvidia.com/object/product_tesla_c1060_us.html
[ETS] http://code.enthought.com/projects
[GPL] http://www.gnu.org/licenses/gpl.html

PySPH: A Python Framework for Smoothed Particle Hydrodynamics

Prabhu Ramachandran[‡*], Chandrashekhar Kaushik[‡]

Abstract—[PySPH] is a Python-based open source parallel framework for Smoothed Particle Hydrodynamics (SPH) simulations. It is distributed under a BSD license. The performance critical parts are implemented in [Cython]. The framework provides a load balanced, parallel execution of solvers. It is designed to be easy to extend. In this paper we describe the architecture of PySPH and how it can be used.

At it's core PySPH provides a particle kernel, an SPH kernel and a solver framework. Serial and parallel versions of solvers for some standard problems are also provided. The parallel solver uses [mpi4py]. We employ a simple but elegant automatic load balancing strategy for the parallelization. Currently, we are able to perform free surface simulations and some gas dynamics simulations. PySPH is still a work in progress and we will discuss our future plans for the project.

Index Terms—parallel, Cython, fluid dynamics, simulation

Introduction

SPH Primer

Smoothed Particle Hydrodynamics (SPH) is a computational simulation technique. It was developed to simulate astral phenomena by [Gingold77] and [Lucy77] in 1977. Since then, it has been used in numerous other fields including fluid-dynamics, gas-dynamics and solid mechanics.

The central idea behind SPH is the use of integral interpolants. Consider a function $f(r)$. It can be represented by the equation

$$f(r) = \int f(r')\delta(r-r')dr' \tag{1}$$

Replacing the delta distribution with an approximate delta function, W, gives us:

$$f(r) = \int f(r')W(r-r',h)dr'. \tag{2}$$

The above equation estimates the value of function f at a point r in space using the weighted values of f at points near it. The weight decreases as the distance between r and r' increase. h in the above equation represents the particle interaction radius. The *support* of the kernel W is some small multiple of h. Outside the support, the value of W is set to zero. Compact support is computationally advantageous since it allows us to avoid an N^2 interaction among particles.

* Corresponding author: prabhu@aero.iitb.ac.in
‡ IIT Bombay

The above equation can be written in summation form as

$$f(r_i) = \sum_j f(r_j)\frac{m_j}{\rho_j}W(r_i-r_j,h) \tag{3}$$

The above equation forms the core of all SPH calculations. The index j loops over all neighboring particles. m_j is the mass of a particle and ρ_j is the density of the particle. The term

$$\frac{m_j}{\rho_j},$$

can be thought of as representing a volume element [Morris96]. Gradients and divergence encountered in the equations representing fluid motion are represented using similar summations. SPH finds widespread use in many domains. [Monaghan05] and [Morris97] give extensive details about the SPH method.

Related Work

Despite the age of SPH and its applicability to many domains, there does not seem to be much effort in developing a unified framework for SPH. [SPHysics] is a FORTRAN-based open source package for performing SPH. It's primary objective is to model free-surface flows. From the provided documentation we feel that it is not easy to set up simulations in this package. [SPH2000] is another parallel framework for SPH written in C++. This code however does not seem to be in active development currently. Moreover, they show exactly one example simulation with their code. Neither package has a publicly accessible source code repository. Therefore, an open source package that is easy to experiment with and extend will be a useful contribution to the community, especially when combined with the flexibility of Python [Oliphant07].

PySPH [PySPH] was created to address this need. It is an open source, parallel, framework for Smoothed Particle Hydrodynamics (SPH) implemented in Python.

Choice of implementation language

We use a combination of Python and [Cython] to implement the framework. Python is a high-level, object-oriented, interpreted programming language which is easy to learn. Python code is also very readable. There are numerous packages (both scientific and otherwise) that can be used to enhance the productivity of applications. A Python-based SPH implementation can take advantage of these packages, which could enhance it in various aspects, from providing plotting facilities (2D and 3D), to generating GUI's, to running SPH simulations from the web, to parallelization. All these features can also be accessed through an interactive

interpreter. [Oliphant07] discusses how Python can be used for scientific computing.

Python, however, is an interpreted language. Thus, compute-intensive tasks implemented in pure Python will be prohibitively slow. To overcome this, we delegate all performance-critical tasks to a language called Cython [Cython]. Cython makes writing C extensions for Python nearly as simple as writing Python code itself. A Cython module is compiled by a compiler into a C extension module. When the C code is compiled, it becomes a module that may be imported from Python. Most of Python's features are available in Cython. Thus, by delegating all performance-critical components to Cython, we are able to overcome the performance hit due to the interpreted nature of Python and still use all of Python's features.

An overview of features

PySPH currently allows a user to set up simulations involving incompressible fluids and free surfaces in two and three dimensions. The framework supports complex geometries. However, only a few simple shapes have been currently implemented. The framework has been designed from the ground up to be parallel. We use mpi4py [mpi4py] for the parallel solver. The parallel solver is automatically load balanced.

In the following, we outline the framework, discuss the current status and future improvements that are planned.

The Framework

The whole framework was designed to enable simple simulations to be set up very easily, and yet be flexible enough to add complex features. We present a high level view of a particle-based simulation in the following.

Guiding Principle - High level view of a simulation

A simulation always involves a few key objects:

- *Solver*: The solver is an object that manages the entire simulation. It typically delegates its activities to other objects like integrators, component managers and arrays of particles.
- *Entities*: The simulation involves distinct collections of particles each representing a particular physical entity. Each entity is a derived class from the base class *EntityBase*. For example, *Fluid* and *Solid* are two different classes and a user may create a collection of fluids and solids using this. This allows a user to set up a simulation with a collection of physical entities.

The high level view outlined in Figure 1 served as the guiding principle while designing various components of the framework.

The various tasks shown in Figure 1 are explained below:

- **Create and set up the solver**: Initially, we create an appropriate solver object for the simulation. Different solvers are used for different kinds of simulations. We also set up various parameters of the solver.
- **Create physical entities**: In this step, we add the physical entities (made of up particles), that will take part in the simulation. Multiple sets of particles could be added, one for each physical entity involved.

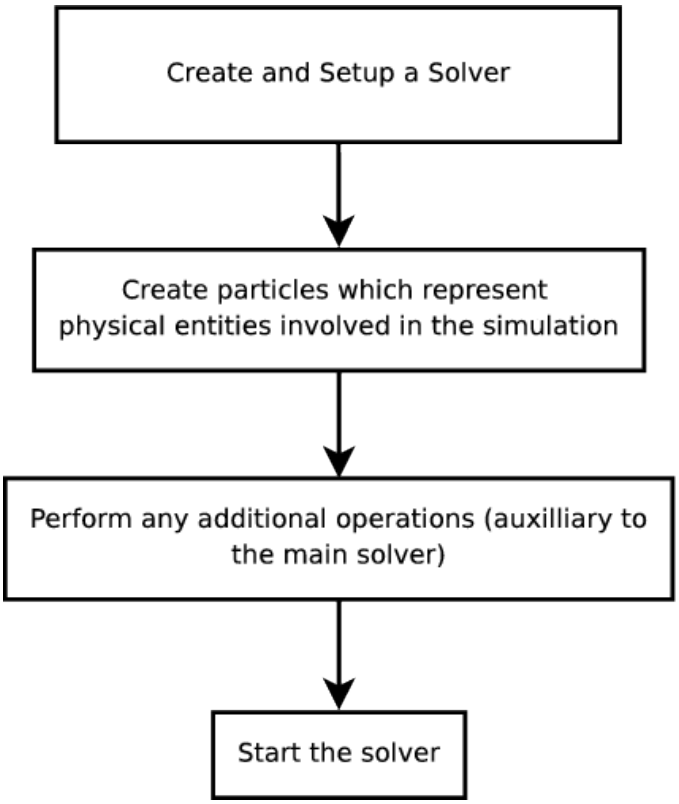


Fig. 1: *Outline of tasks to set up a simulation.*

- **Additional operations to the solver**: We may require the solver to perform additional operations (apart from the main simulation), like writing data to file, plotting the data etc. This is configured during this step.
- **Start the solver**: The solver iterations are started.

The outline given above is very generic. This set of steps is useful in setting up almost any particle-based simulation. Parallel simulations too should adhere to the basic outline given above. Given below is pseudo-Python code to run a simple serial simulation:

```
# Imports...
solver = FSFSolver(time_step=0.0001,
             total_simulation_time=10.,
             kernel=CubicSpline2D())

# create the two entities.
dam_wall  = Solid(name='dam_wall')
dam_fluid = Fluid(name='dam_fluid')

# The particles for the wall.
rg = RectangleGenerator(...)
dam_wall.add_particles(
                  rg.get_particles())
solver.add_entity(dam_wall)
# Particles for the left column of fluid.
rg = RectangleGenerator(...)
dam_fluid.add_particles(
                  rg.get_particles())
solver.add_entity(dam_fluid)

# start the solver.
solver.solve()
```

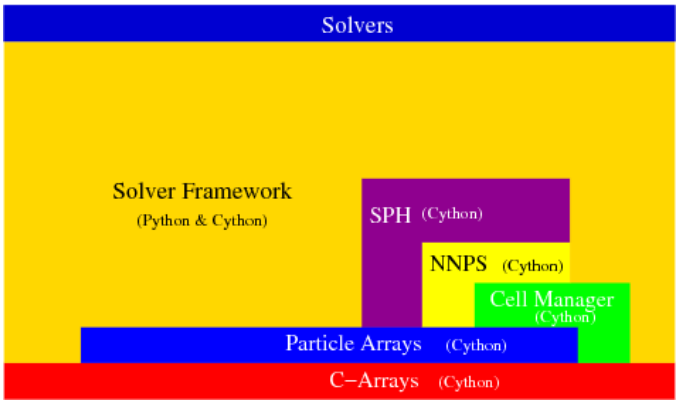


Fig. 2: *Architecture of the framework*

Architecture Overview

The architecture may be broadly split into the following:

- the particle kernel,
- the SPH kernel,
- the solver framework,
- serial and parallel solvers.

The overall architecture of the framework is shown in Figure 2. We discuss this in detail in the following sections.

Particle kernel

A fast implementation of arrays in Cython forms the foundation of the framework. Arrays are ubiquitous in the implementation, hence the implementation is made as fast as possible (close to C performance) using Cython. The base class for these arrays is called **BaseArray** and subclasses of these in the form of **IntArray**, **FloatArray** etc. are made available. These expose a **get_npy_array** method which returns a numpy array which internally uses the same C data buffer. Our arrays may be resized and are up to 4 times faster than numpy arrays when used from Cython.

The **ParticleArray** module uses these arrays extensively and allows us to represent collections of particles in the framework. It is also implemented in Cython to achieve maximum performance. Each **ParticleArray** maintains a collection of particle properties and uses the arrays to store this data. Since the arrays allow the developer to manipulate them as numpy arrays, it becomes easy to perform calculations on the particle properties, if required.

One of the central requirements of the SPH is to find the nearest neighbors of a given particle. This is necessary in order to calculate the influence of each particle on the others. We do this using a nearest neighbor algorithm (Nearest Neighbor Particle Search - NNPS) which bins the domain into a collection of fixed size cells. Particles are organized into a dictionary keyed on a tuple indicative of the location of the particle. The nearest neighbor search is collectively performed by the **CellManager** class and the **nnps** modules. Both are implemented in Cython.

SPH kernel

The SPH kernel consits of the **sph** module which contains classes to perform the SPH summation (as given in the equations in the introductory section) and also to represent particle interactions. This includes a variety of kernels. These are implemented so as to use the **nnps** and other modules discussed earlier. These are all implemented in Cython for performance.

Solver framework

Finally, bringing all the underlying modules together is the **Solver framework**. The framework is component based, and allows users to write components, which are subclasses of **SolverComponent**, with a standard interface set. The **SolverComponent** is the base class for all classes that perform any operation on any of the entities. Many abstractions required for a solver have been implemented, and a user can inherit from various classes to implement new formulations. The **ComponentManager** manages all the **SolverComponents** used by the solver. It is also responsible for the property requirements of each of the components involved in a calculation. Thus, if an entity is operated by a component that requires a particular property to be available, the manager ensures that the entity is suitably set up. An **Integrator** class handles the actual time integration. The **Integrator** is also a **SolverComponent**. These are implemented in a combination of Python and Cython.

Solvers

New solvers are written using the various abstractions developed in the solver framework and all of them derive from the **SolverBase** class. Serial and parallel solvers are written using the functionality made available in the solver framework.

Parallelization

In SPH simulations, particles simply influence other particles in a small neighborhood around them. Thus, in order to perform a parallel simulation one needs to:

- partition the particles among different processors, and
- share neighboring particle information between some of the processors.

For an SPH simulation, this does require a reasonable amount of communication overhead since the particles are moving and the neighbor information keeps changing. In addition to this, we would like the load on the processors to be reasonably balanced. This is quite challenging.

Our objective was to maintain an outline similar to the serial code for setting up simulations that run in parallel. For parallelization of the framework, ideally only the **CellManager** needs to be aware of the parallelism. The components in the solver framework simply operate on particle data that they are presented with. This is achievable to a good extent, except when a component requires global data, in which case the serial component may need to subclassed and a parallel version written, which collects the global data before executing the serial version code. A good example for this is when a component needs to know the maximum speed of sound in the entire domain in order to limit the time-step say.

The pseudo-code of a typical parallel simulation is the same as the serial example given earlier with just one change to the solver as below:

```
solver = ParallelFSFSolver(
              time_step=0.0001,
              total_simulation_time=10.,
              kernel=CubicSpline2D())

# Code to load particles in proc with
# rank 0.
```

In the above pseudo-code, the only thing that changes is the fact that we instantiate a parallel solver rather than a serial one.

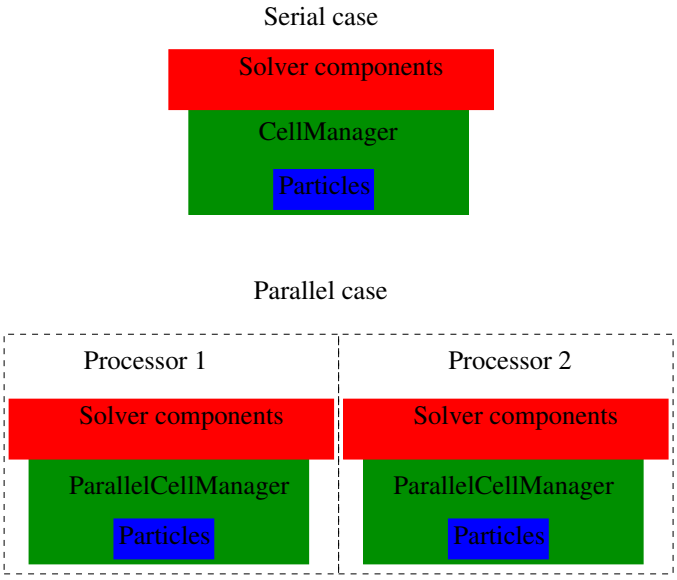


Fig. 3: *The parallel solvers simply use a ParallelCellManager instead of a CellManager.*

We also ensure that the particles are all loaded only on the first processor. The **ParallelCellManager** manages the parallel neighbor information. It also performs automatic load-balancing by distributing the particles to different processors on demand based on the number of particles in each processor.

The full details of the parallelization are beyond the scope of this article but we provide a brief outline of the general approach. More details can be obtained from [Kaushik09].

The basic idea of the parallelization involves the following key steps:

- Particles are organized into small cubical **Cells**. Each cell manages a set of particles. Cells are created and destroyed on demand depending on where the particles are present.
- A region consists of a set of usually (but not always) connected cells. Each region is managed by one processor.
- The domain of particles is decomposed into cells and regions and allocated to different processors.
- Cells are moved between processors in order to balance the load.

In addition, the **ParallelCellManager** ensures that each processor has all the necessary information such that an SPH computation may be performed on the the particles it manages.

Figure 3 outlines how the parallel and serial solvers are set up internally. In both cases, solver components operate on cell managers to obtain the nearest neighbors and get the particles, the only difference being the **ParallelCellManager**, which manages the load distribution and communication in the parallel case.

It is important to note that the basic ideas for the parallel algorithm were implemented and tested in pure Python using mpi4py. This was done in highly fragmented time and was possible only because of the convenience of both Python and mpi4py. Mpi4py allows us to send Python objects to processors and this allowed us to focus on the algorithm without worrying about the details of MPI. The use of Python enabled rapid prototyping and its libraries made it easy to visualize the results. In roughly

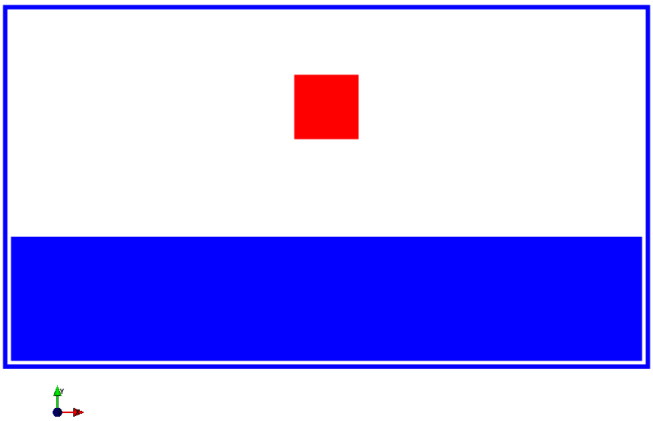


Fig. 4: *Initial condition of a square block of water falling towards a vessel with water.*

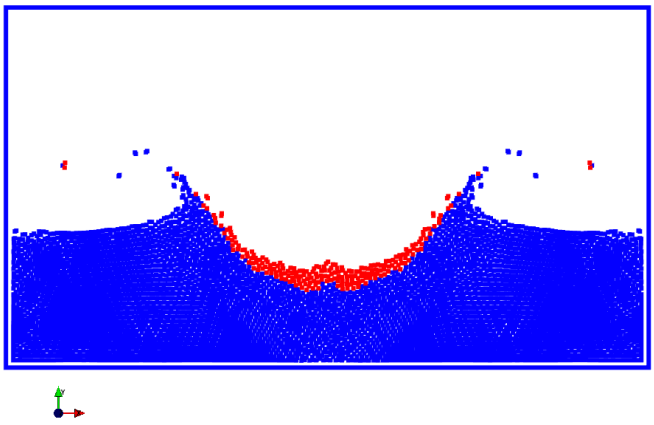


Fig. 5: *Square block of water after it strikes a vessel containing water simulated with the SPH.*

1500 lines we had implemented the core ideas, added support for visualization, logging and command line options. The initial design was subsequently refined and parts of it implemented in Cython. Thus, the use of Python clearly allowed us to prototype rapidly and yet obtain good performance with Cython.

Current status

Figures 4, 5 show the fluid at a particular instant when a square block of water strikes a vessel filled with water. This is a two-dimensional simulation.

Figure 6 shows a typical 3D dam-break problem being simulated with 8 processors. The fluid involved is water. The colors indicate the processor on which the particles are located.

The current capabilities of PySPH include the following:

- Fully automatic, load balanced, parallel framework.
- Fairly easy to script.
- Good performance.
- Relatively easy to extend.
- Solver for incompressible free surface flows.

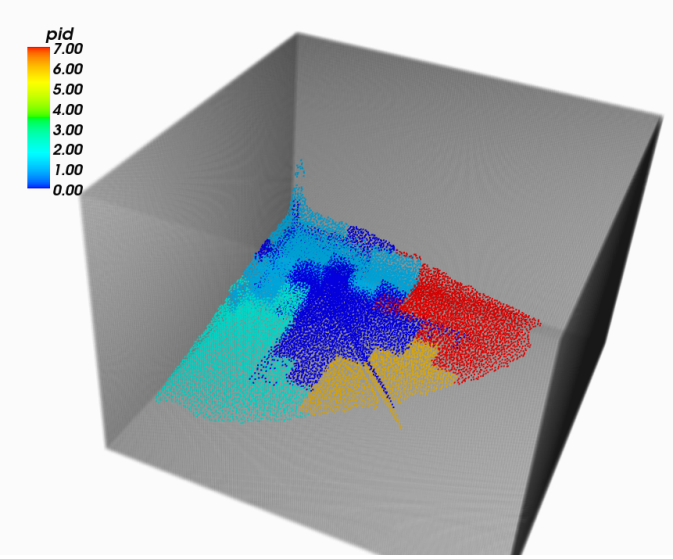


Fig. 6: *3D dam-break problem simulated on 8 processors with particles colored as per processor ID indicating a load balanced simulation.*

Most importantly, we have a working framework and a reasonable design which provides good performance. However, there are several things we need to improve.

Future work

Our code is available in the form of a Mercurial repository on Google's project hosting site [PySPH]. However, the code is not ready for a proper release yet because we would like to perform a redesign of some parts of the solver framework. At the moment, they are a little too complex. Once this is done we would like to do the following:

- Improve the documentation.
- Reduce any compulsory dependence on VTK or TVTK.
- Improve testing on various platforms.
- A full-fledged release.
- Support for gas-dynamics problems.
- Support for solid mechanics problems.

This would take a few more months and at which point we will make a formal release.

Conclusions

We have provided a high-level description of the current capabilities and architecture of PySPH. We have also mentioned what we believe are the future directions we would like to take. We think we have made an important beginning and believe that PySPH will help enable open research and computing using particle-based computing in the future. It is important to note that Python has been centrally important in the development of PySPH by way of its rapid prototyping capability and access to a plethora of libraries.

REFERENCES

[Cython] http://www.cython.org

[Gingold77] R. A. Gingold and J. J. Monaghan. *Smoothed particle hydrodynamics: theory and application to non-spherical stars*, Mon. Not. R. astr. Soc., 181:375-389, 1977.

[Kaushik09] Chandrashekhar P. Kaushik. *A Python based parallel framework for Smoothed Particle Hydrodynamics*, M.Tech. dissertation, Department of Computer Science and Engineering, IIT Bombay, 2009.

[Lucy77] L. B. Lucy. *A numerical approach to testing the fission hypothesis*, The Astronomical Journal, 82(12):1013-1024, December 1977.

[Monaghan05] J. J. Monaghan. *Smoothed particle hydrodynamics*, Reports on Progress in Physics, 68(8):1703-1759, 2005.

[Morris96] J. P. Morris. *Analysis of smoothed particle hydrodynamics with applications*, PhD Thesis, Monash University, Australia, 1996.

[Morris97] J. P. Morris, P. J. Fox and Yi Zhu. *Modeling low Reynolds number incompressible flows using SPH*, Journal of Computational Physics, 136(1):214-226, 1997.

[mpi4py] http://mpi4py.scipy.org

[Oliphant07] Travis E. Oliphant. *Python for scientific computing*, Computing in science and engineering, 9:10-20, 2007.

[PySPH] http://pysph.googlecode.com

[SPH2000] S. Ganzenmuller, S. Pinkenburg and W. Rosenstiel. *SPH2000: A Parallel Object-Oriented Framework for Particle Simulations with SPH*, Lecture notes in computer science, 3648:1275-1284, 2005.

[SPHysics] Gòmez-Gesteira M., Rogers, B.D., Dalrymple, R.A., Crespo, A.J.C. and Narayanaswamy, M. *User guide for the SPHysics code 1.4*, http://wiki.manchester.ac.uk/sphysics.

Audio-Visual Speech Recognition using SciPy

Helge Reikeras*, Ben Herbst‡, Johan du Preez‡, Herman Engelbrecht‡

Abstract—In audio-visual automatic speech recognition (AVASR) both acoustic and visual modalities of speech are used to identify what a person is saying. In this paper we propose a basic AVASR system implemented using SciPy, an open source Python library for scientific computing. AVASR research draws from the fields of signal processing, computer vision and machine learning, all of which are active fields of development in the SciPy community. As such, AVASR researchers using SciPy are able to benefit from a wide range of tools available in SciPy.

The performance of the system is tested using the Clemson University audio-visual experiments (CUAVE) database. We find that visual speech information is in itself not sufficient for automatic speech recognition. However, by integrating visual and acoustic speech information we are able to obtain better performance than what is possible with audio-only ASR.

Index Terms—speech recognition, machine learning, computer vision, signal processing

Introduction

Motivated by the multi-modal manner humans perceive their environment, research in Audio-Visual Automatic Speech Recognition (AVASR) focuses on the integration of acoustic and visual speech information with the purpose of improving accuracy and robustness of speech recognition systems. AVASR is in general expected to perform better than audio-only automatic speech recognition (ASR), especially so in noisy environments as the visual channel is not affected by acoustic noise.

Functional requirements for an AVASR system include acoustic and visual feature extraction, probabilistic model learning and classification.

In this paper we propose a basic AVASR system implemented using SciPy. In the proposed system mel-frequency cepstrum coefficients (MFCCs) and active appearance model (AAM) parameters are used as acoustic and visual features, respectively. Gaussian mixture models are used to learn the distributions of the feature vectors given a particular class such as a word or a phoneme. We present two alternatives for learning the GMMs, expectation maximization (EM) and variational Bayesian (VB) inference.

The performance of the system is tested using the CUAVE database. The performance is evaluated by calculating the misclassification rate of the system on a separate test data data. We find that visual speech information is in itself not sufficient for automatic speech recognition. However, by integrating visual and acoustic speech we are able to obtain better performance than what is possible with audio-only ASR.

* *Corresponding author: helge@ml.sun.ac.za*
‡ *Stellenbosch University*

Feature extraction

Acoustic speech

MFCCs are the standard acoustic features used in most modern speech recognition systems. In [Dav80] MFCCs are shown experimentally to give better recognition accuracy than alternative parametric representations.

MFCCs are calculated as the cosine transform of the logarithm of the short-term energy spectrum of the signal expressed on the mel-frequency scale. The result is a set of coefficients that approximates the way the human auditory system perceives sound.

MFCCs may be used directly as acoustic features in an AVASR system. In this case the dimensionality of the feature vectors equals the number of MFCCs computed. Alternatively, velocity and acceleration information may be included by appending first and second order temporal differences to the feature vectors.

The total number of feature vectors obtained from an audio sample depends on the duration and sample rate of the original sample and the size of the window that is used in calculating the cepstrum (a windowed Fourier transform).

MFCCs are available in the `scikits.talkbox.features.mfcc`. The default number of MFCCs computed is thirteen.

Example usage:

```
from scikits.audiolab import wavread
from scikits.talkbox.features import mfcc

# data: raw audio data
# fs: sample rate
data, fs = wavread('sample.wav')[:2]

# ceps: cepstral cofficients
ceps = mfcc(input, fs=fs)[0]
```

Figure 1 shows the original audio sample and mel-frequency cepstrum for the word “zero”.

Visual speech

While acoustic speech features can be extracted through a sequence of transformations applied to the input audio signal, extracting visual speech features is in general more complicated. The visual information relevant to speech is mostly contained in the motion of visible articulators such as lips, tongue and jaw. In order to extract this information from a sequence of video frames it is advantageous to track the complete motion of the face and facial features.

AAM [Coo98] fitting is an efficient and robust method for tracking the motion of deformable objects in a video sequence.

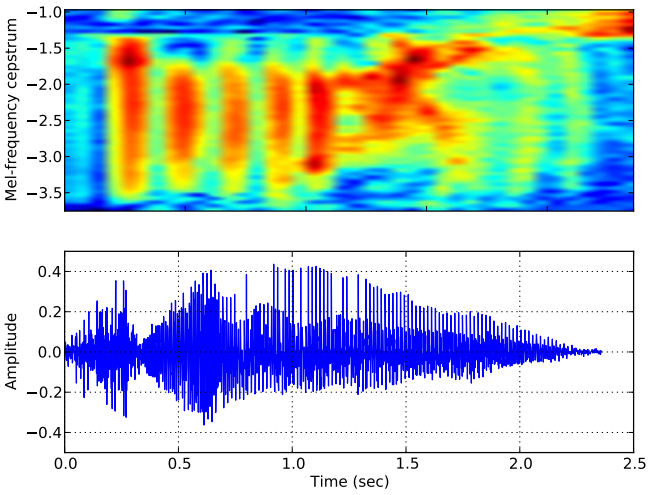


Fig. 1: *Acoustic feature extraction from an audio sample of the word "zero". Mel-frequency cepstrum (top) and original audio sample (bottom).*

AAMs model variations in shape and texture of the object of interest. To build an AAM it is necessary to provide sample images with the shape of the object annotated. Hence, in contrast to MFCCs, AAMs require prior training before being used for tracking and feature extraction.

The shape of an appearance model is given by a set of (x,y) coordinates represented in the form of a column vector

$$\mathbf{s} = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n)^{\mathrm{T}}. \tag{1}$$

The coordinates are relative to the coordinate frame of the image.

Shape variations are restricted to a base shape $\mathbf{s}_0$ plus a linear combination of a set of N shape vectors

$$\mathbf{s} = \mathbf{s}_0 + \sum_{i=1}^{N} p_i \mathbf{s}_i \tag{2}$$

where p_i are called the shape parameters of the AAM.

The base shape and shape vectors are normally generated by applying principal component analysis (PCA) to a set of manually annotated training images. The base shape $\mathbf{s}_0$ is the mean of the object annotations in the training set, and the shape vectors are N singular vectors corresponding to the N largest singular values of the data matrix (constructed from the training shapes). Figure 2 shows an example of a base mesh and the first three shape vectors corresponding to the three largest singular values of the data matrix.

The appearance of an AAM is defined with respect to the base shape $\mathbf{s}_0$. As with shape, appearance variation is restricted to a base appearance plus a linear combination of M appearance vectors

$$A(\mathbf{x}) = A_0 + \sum_{i=1}^{M} \lambda_i A_i(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathbf{s}_0. \tag{3}$$

To generate an appearance model, the training images are first shape-normalized by warping each image onto the base mesh using a piecewise affine transformation. Recall that two sets of three corresponding points are sufficient for determining an affine transformation. The shape mesh vertices are first triangulated. The collection of corresponding triangles in two shapes meshes then defines a piecewise affine transformation between the two shapes.

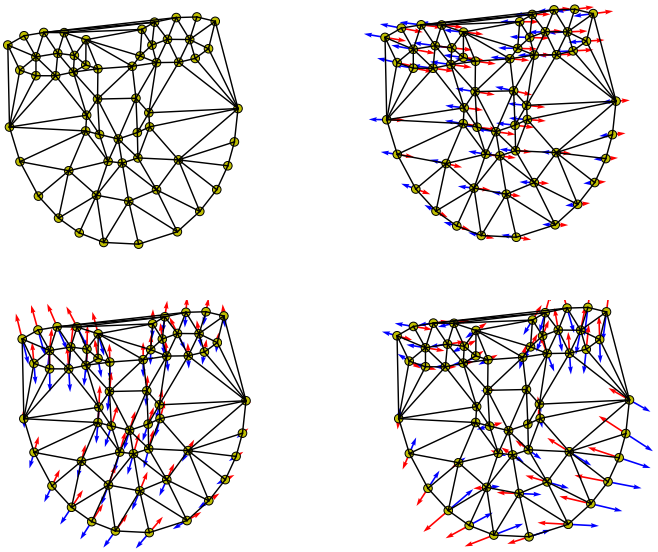


Fig. 2: *Triangulated base shape $\mathbf{s}_0$ (top left), and first three shape vectors $\mathbf{p}_1$ (top right), $\mathbf{p}_2$ (bottom left) and $\mathbf{p}_3$ (bottom right) represented by arrows superimposed onto the triangulated base shape.*

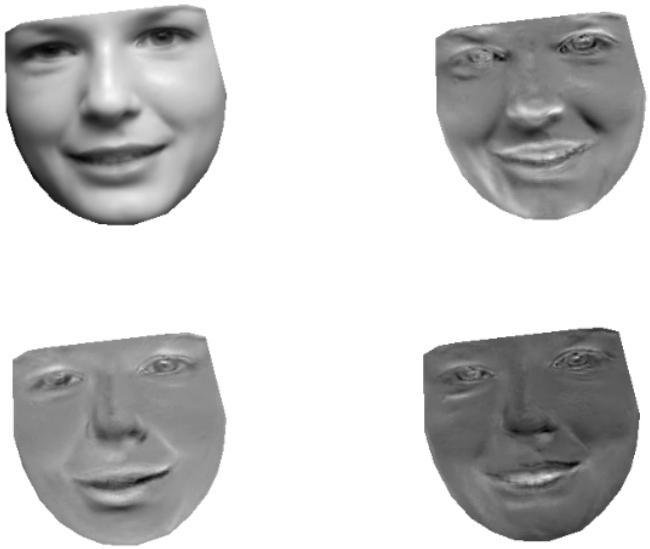


Fig. 3: *Mean appearance A_0 (top left) and first three appearance images A_1 (top right), A_2 (bottom left) and A_3 (bottom right).*

The pixel values within each triangle in the training shape $\mathbf{s}$ are warped onto the corresponding triangle in the base shape $\mathbf{s}_0$ using the affine transformation defined by the two triangles.

The appearance model is generated from the shape-normalized images using PCA. Figure 3 shows the base appearance and the first three appearance images.

Tracking of an appearance in a sequence of images is performed by minimizing the difference between the base model appearance, and the input image warped onto the coordinate frame of the AAM. For a given image I we minimize

$$\underset{\boldsymbol{\lambda},\mathbf{p}}{\operatorname{argmin}} \sum_{\mathbf{x}\in\mathbf{s}_0} \left[A_0(\mathbf{x}) + \sum_{i=1}^{M} \lambda_i A_i(\mathbf{X}) - I(\mathbf{W}(\mathbf{x};\mathbf{p})) \right]^2 \tag{4}$$

where $\mathbf{p} = \{p_1, \ldots, p_N\}$ and $\boldsymbol{\lambda} = \{\lambda_1, \ldots, \lambda_N\}$. For the rest of the

discussion of AAMs we assume that the variable $\mathbf{x}$ takes on the image coordinates contained within the base mesh $\mathbf{s}_0$ as in (4).

In (4) we are looking for the optimal alignment of the input image, warped backwards onto the frame of the base appearance $A_0(\mathbf{x})$.

For simplicity we will limit the discussion to shape variation and ignore any variation in texture. The derivation for the case including texture variation is available in [Mat03]. Consequently (4) now reduces to

$$\underset{\mathbf{p}}{\operatorname{argmin}} \sum_{\mathbf{x}} [A_0(\mathbf{x}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}))]^2. \tag{5}$$

Solving (5) for $\mathbf{p}$ is a non-linear optimization problem. This is the case even if $\mathbf{W}(\mathbf{x};\mathbf{p})$ is linear in $\mathbf{p}$ since the pixel values $I(\mathbf{x})$ are in general nonlinear in $\mathbf{x}$.

The quantity that is minimized in (5) is the same as in the classic Lucas-Kanade image alignment algorithm [Luc81]. In the Lukas-Kanade algorithm the problem is first reformulated as

$$\underset{\Delta\mathbf{p}}{\operatorname{argmin}} \sum_{\mathbf{x}} [A_0(\mathbf{X}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}))]^2. \tag{6}$$

This equation differs from (5) in that we are now optimizing with respect to $\Delta\mathbf{p}$ while assuming $\mathbf{p}$ is known. Given an initial estimate of $\mathbf{p}$ we update with the value of $\Delta\mathbf{p}$ that minimizes (6) to give

$$\mathbf{p}^{\text{new}} = \mathbf{p} + \Delta\mathbf{p}.$$

This will necessarily decrease the value of (5) for the new value of $\mathbf{p}$. Replacing $\mathbf{p}$ with the updated value for $\mathbf{p}^{\text{new}}$, this procedure is iterated until convergence at which point $\mathbf{p}$ yields the (local) optimal shape parameters for the input image I.

To solve (6) Taylor expansion is used [Bak01] which gives

$$\underset{\Delta\mathbf{p}}{\operatorname{argmin}} \sum_{\mathbf{x}} \left[A_0(\mathbf{W}(\mathbf{x};\mathbf{p})) - I(\mathbf{W}(\mathbf{x};\mathbf{p})) - \nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \Delta\mathbf{p} \right]^2 \tag{7}$$

where ∇I is the gradient of the input image and $\partial\mathbf{W}/\partial\mathbf{p}$ is the Jacobian of the warp evaluated at $\mathbf{p}$.

The optimal solution to (7) is found by setting the partial derivative with respect to $\Delta\mathbf{p}$ equal to zero which gives

$$2\sum_{\mathbf{x}} \left[\nabla\mathbf{I} \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \right]^{\mathrm{T}} \left[A_0(\mathbf{x}) - I(\mathbf{W}(\mathbf{x})) - \nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \Delta\mathbf{p} \right] = 0. \tag{8}$$

Solving for $\Delta\mathbf{p}$ we get

$$\Delta\mathbf{p} = \mathbf{H}^{-1} \sum_{\mathbf{x}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \right]^{\mathrm{T}} [A_0(\mathbf{x}) - I(\mathbf{W}(\mathbf{x};\mathbf{p}))] \tag{9}$$

where $\mathbf{H}$ is the Gauss-Newton approximation to the Hessian matrix given by

$$\mathbf{H} = \sum_{\mathbf{x}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \right]^{\mathrm{T}} \left[\nabla I \frac{\partial \mathbf{W}}{\partial \mathbf{p}} \right]. \tag{10}$$

For a motivation for the backwards warp and further details on how to compute the piecewise linear affine warp and the Jacobian see [Mat03].

A proper initialization of the shape parameters $\mathbf{p}$ is essential for the first frame. For subsequent frames $\mathbf{p}$ may be initialized as the optimal parameters from the previous frame.

The Lucas-Kanade algorithm is a Gauss-Newton gradient descent algorithm. Gauss-Newton gradient descent is available in `scipy.optimize.fmin_ncg`.

Example usage:

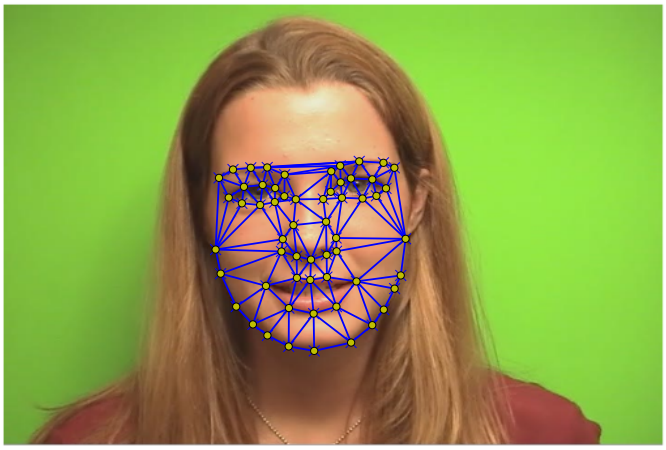


Fig. 4: *AAM fitted to an image.*

```
from scipy import empty
from scipy.optimize import fmin_ncg
from scikits.image.io import imread

# NOTE: The AAM module is currently under development
import aam

# Initialize AAM from visual speech training data
vs_aam = aam.AAM('./training_data/')

I = imread('face.jpg')

def error_image(p):
    """ Compute error image given p """

    # Piecewise linear warp the image onto
    # the base AAM mesh
    IW = vs_aam.pw_affine(I,p)

    # Return error image
    return aam.A0-IW

def gradient_descent_images(p):
    """ Compute gradient descent images given p """
    ...
    return gradIW_dWdP

def hessian(p):
    """ Compute hessian matrix """"
    ...
    return H

# Update p
p = fmin_ncg(f=error_image,
                 x0=p0,
                 fprime=gradient_descent_images,
                 fhess=hessian)
```

Figure 4 shows an AAM fitted to an input image. When tracking motion in a video sequence an AAM is fitted to each frame using the previous optimal fit as a starting point.

In [Bak01] the AAM fitting method described above is referred to as "forwards-additive".

As can be seen in Figure 2 the first two shape vectors mainly correspond to the movement in the up-down and left-right directions, respectively. As these components do not contain any speech related information we can ignore the corresponding shape parameters p_1 and p_2 when extracting visual speech features. The remaining shape parameters, $p_3, \ldots, p_N$, are used as visual features in the AVASR system.

Models for audio-visual speech recognition

Once acoustic and visual speech features have been extracted from respective modalities, we learn probabilistic models for each of the classes we need to discriminate between (e.g. words or phonemes). The models are learned from manually labeled training data. We require these models to *generalize* well; i.e. the models must be able to correctly classify novel samples that was not present in the training data.

Gaussian Mixture Models

Gaussian Mixture Models (GMMs) provide a powerful method for modeling data distributions under the assumption that the data is independent and identically distributed (i.i.d.). GMMs are defined as a weighted sum of Gaussian probability distributions

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{11}$$

where π_k is the weight, $\boldsymbol{\mu}_k$ the mean, and $\boldsymbol{\Sigma}_k$ the covariance matrix of the kth mixture component.

Maximum likelihood

The log likelihood function of the GMM parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ given a set of D-dimensional observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{12}$$

Note that the log likelihood is a function of the GMM parameters $\boldsymbol{\pi}, \boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. In order to fit a GMM to the observed data we maximize this likelihood with respect to the model parameters.

Expectation maximization

The Expectation Maximization (EM) algorithm [Bis07] is an efficient iterative technique for optimizing the log likelihood function. As its name suggests, EM is a two stage algorithm. The first (*E* or *expectation*) step calculates the expectations for each data point to belong to each of the mixture components. It is also often expressed as the *responsibility* that the kth mixture component takes for "explaining" the nth data point, and is given by

$$r_{nk} = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}. \tag{13}$$

Note that this is a "soft" assignment where each data point is assigned to a given mixture component with a certain probability. Once the responsibilities are available the model parameters are updated ("M" or "maximization"' step). The quantities

$$N_k = \sum_{n=1}^{N} r_{nk} \tag{14}$$

$$\bar{\mathbf{x}}_k = \sum_{n=1}^{N} r_{nk} \mathbf{x}_n \tag{15}$$

$$\mathrm{S}_k = \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \bar{\mathbf{x}}_k)(\mathbf{x}_n - \bar{\mathbf{x}}_k)^{\mathrm{T}} \tag{16}$$

are first calculated. Finally the model parameters are updated as

$$\pi_k^{\text{new}} = \frac{N_k}{N} \tag{17}$$

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{\bar{\mathbf{x}}_k}{N_k} \tag{18}$$

$$\boldsymbol{\Sigma}_k^{\text{new}} = \frac{S_k}{N_k}. \tag{19}$$

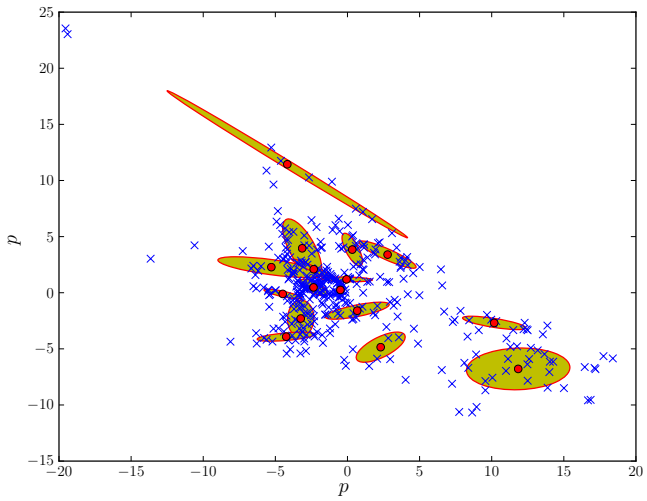


Fig. 5: *Visual speech GMM of the word "zero" learned using EM algorithm on two-dimensional feature vectors.*

The EM algorithm in general only converges to a local optimum of the log likelihood function. Thus, the choice of initial parameters is crucial. See [Bis07] for the derivation of the equations.

GMM-EM is available in `scikits.learn.em`.

Example usage:

```
from numpy import loadtxt
from scikits.learn.em import GM, GMM, EM

# Data dimensionality
D = 8

# Number of Gaussian Mixture Components
K = 16

# Initialize Gaussian Mixture Model
gmm = GMM(GM(D,K))

# X is the feature data matrix

# Learn GMM
EM().train(X,gmm)
```

Figure 5 shows a visual speech GMM learned using EM. For illustrative purposes only the first two speech-related shape parameters p_3 and p_4 are used. The shape parameters are obtained by fitting an AAM to each frame of a video of a speaker saying the word "zero". The crosses represent the training data, the circles are the means of the Gaussians and the ellipses are the standard deviation contours (scaled by the inverse of the weight of the corresponding mixture component for visualization purposes). The video frame rate is 30 frames per second (FPS) and the number of mixture components used is 16.

Note that in practice more than two shape parameters are used, which usually also requires an increase in the number of mixture components necessary to sufficiently capture the distribution of the data.

Variational Bayes

An important question that we have not yet answered is how to choose the number of mixture components. Too many components lead to redundancy in the number of computations, while too few may not be sufficient to represent the structure of the data.

Additionally, too many components easily lead to overfitting. Overfitting occurs when the complexity of the model is not in proportion to the amount of available training data. In this case the data is not sufficient for accurately estimating the GMM parameters.

The maximum likelihood criteria is unsuitable to estimate the number of mixture components since it increases monotonically with the number of mixture components. Variational Bayesian (VB) inference is an alternative learning method that is less sensitive than ML-EM to over-fitting and singular solutions while at the same time leads to automatic model complexity selection [Bis07].

As it simplifies calculation we work with the precision matrix $\mathbf{\Lambda} = \mathbf{\Sigma}^{-1}$ instead of the covariance matrix.

VB differs from EM in that the parameters are modeled as random variables. Suitable conjugate distributions are the Dirichlet distribution

$$p(\boldsymbol{\pi}) = C(\boldsymbol{\alpha}_0) \prod_{k=1}^{K} \pi_k^{\alpha_0 - 1} \tag{20}$$

for the mixture component weights, and the Gaussian-Wishart distribution

$$p(\boldsymbol{\mu}, \mathbf{\Lambda}) = \prod_{k=1}^{K} \mathcal{N}(\boldsymbol{\mu}_k | \boldsymbol{m}_0, \beta_0 \Lambda_k) \mathcal{W}(\Lambda_k | \mathbf{W}_0, \boldsymbol{\nu}_0) \tag{21}$$

for the means and precisions of the mixture components.

In the VB framework, learning the GMM is performed by finding the posterior distribution over the model parameters given the observed data. This posterior distribution can be found using VB inference as described in [Bis07].

VB is an iterative algorithm with steps analogous to the EM algorithm. Responsibilities are calculated as

$$r_{nk} = \frac{\rho_{nk}}{\sum_{j=1}^{K} \rho_{nj}}. \tag{22}$$

The quantities ρ_{nk} are given in the log domain by

$$\begin{aligned} \ln \rho_{nk} &= \mathbb{E}[\ln \pi_k] + \frac{1}{2}\mathbb{E}[\ln |\mathbf{\Lambda}|] - \frac{D}{2}\ln 2\pi \\ &\quad - \frac{1}{2}\mathbb{E}_{\boldsymbol{\mu}_k, \mathbf{\Lambda}_k}[(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} \mathbf{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k)] \end{aligned} \tag{23}$$

where

$$\begin{aligned} \mathbb{E}_{\boldsymbol{\mu}, \mathbf{\Lambda}}[(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} \mathbf{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k)] &= D\beta_k^{-1} \\ + \nu_k (\mathbf{x}_n - \mathbf{m}_k)^{\mathrm{T}} \mathbf{W}_k (\mathbf{x}_n - \mathbf{m}_k) & \end{aligned} \tag{24}$$

and

$$\ln \widetilde{\pi}_k = \mathbb{E}[\ln \pi_k] = \psi(\alpha_k) - \psi(\widehat{\alpha}_k) \tag{25}$$

$$\begin{aligned} \ln \widetilde{\Lambda}_k &= \mathbb{E}[\ln |\mathbf{\Lambda}_k|] = \sum_{i=1}^{D} \psi\left(\frac{\nu_k + 1 - i}{2}\right) \\ &\quad + D\ln 2 + \ln |\mathbf{W}_k|. \end{aligned} \tag{26}$$

Here $\widehat{\alpha} = \sum_k \alpha_k$ and ψ is the derivative of the logarithm of the gamma function, also called the digamma function. The digamma function is available in SciPy as `scipy.special.psi`.

The analogous M-step is performed using a set of equations similar to those found in EM. First the quantities

$$N_k = \sum_n r_{nk} \tag{27}$$

$$\bar{\mathbf{x}}_k = \frac{1}{N_k} \sum_n r_{nk} \mathbf{x}_n \tag{28}$$

$$\mathbf{S}_k = \frac{1}{N_k} \sum_n r_{nk} (\mathbf{x}_n - \bar{\mathbf{x}}_k)(\mathbf{x}_n - \bar{\mathbf{x}}_k)^{\mathrm{T}} \tag{29}$$

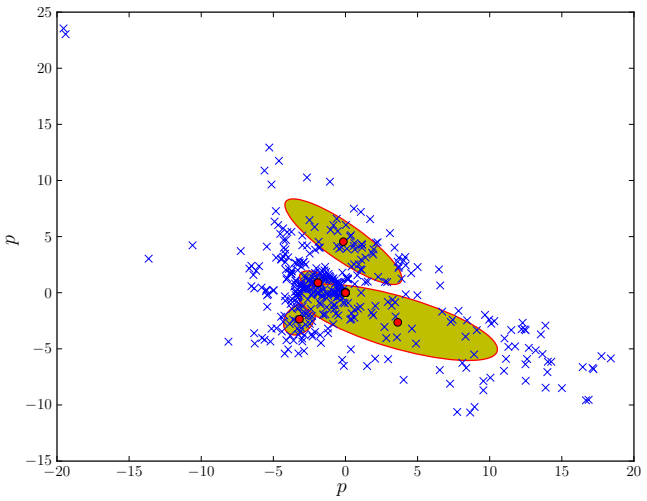


Fig. 6: Visual speech GMM of the word "zero" learned using the VB algorithm on two-dimensional feature vectors.

are calculated. The posterior model parameters are then updated as

$$\alpha_k^{\text{new}} = \alpha_0 + N_k \tag{30}$$

$$\beta_k^{\text{new}} = \beta_0 + N_k \tag{31}$$

$$\mathbf{m}_k^{\text{new}} = \frac{1}{\beta_k}(\beta_0 \mathbf{m}_0 + N_k \bar{\mathbf{x}}_k) \tag{32}$$

$$\begin{aligned} \mathbf{W}_k^{\text{new}} &= \mathbf{W}_0 + N_k \mathbf{S}_k + \\ &\quad \frac{\beta_0 N_k}{\beta_0 + N_k} (\bar{\mathbf{x}} - \mathbf{m}_0)(\bar{\mathbf{x}} - \mathbf{m}_0)^{\mathrm{T}} \end{aligned} \tag{33}$$

$$\nu_k^{\text{new}} = \nu_0 + N_k. \tag{34}$$

Figure 6 shows a GMM learned using VB on the same data as in Figure 5. The initial number of components is again 16. Compared to Figure 5 we observe that VB results in a much sparser model while still capturing the structure of the data. In fact, the redundant components have all converged to their prior distributions and have been assigned the weight of 0 indicating that these components do not contribute towards "explaining" the data and can be pruned from the model. We also observe that outliers in the data (which is likely to be noise) is to a large extent ignored.

We have recently developed a Python VB class for `scikits.learn`. The class conforms to a similar interface as the EM class and will soon be available in the development version of `scikits.learn`.

Experimental results

A basic AVASR system was implemented using SciPy as outlined in the previous sections.

In order to test the system we use the CUAVE database [Pat02]. The CUAVE database consists of 36 speakers, 19 male and 17 female, uttering isolated and continuous digits. Video of the speakers is recorded in frontal, profile and while moving. We only use the portion of the database where the speakers are stationary and facing the camera while uttering isolated digits. We use data from 24 speakers for training and the remaining 12 for testing. Hence, data from the speakers in the test data are not used for

Fig. 7: *Frames from the CUAVE audio-visual data corpus.*

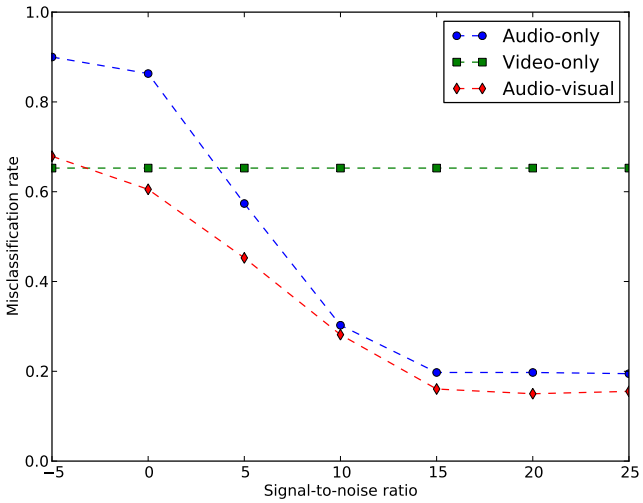


Fig. 8: *Misclassification rate.*

training. This allows us to evaluate how well the models generalize to speakers other than than those used for training. A sample frame from each speaker in the dataset is shown in Figure 7.

In the experiment we build an individual AAM for each speaker by manually annotating every 50th frame. The visual features are then extracted by fitting the AAM to each frame in the video of the speaker.

Training the speech recognition system consists of learning acoustic and visual GMMs for each digit using samples from the training data. Learning is performed using VB inference. Testing is performed by classifying the test data. To evaluate the performance of the system we use the misclassification rate, i.e. the number of wrongly classified samples divided by the total number of samples.

We train acoustic and visual GMMs separately for each digit. The probability distributions (see (11)) are denoted by $p(\mathbf{x}_A)$ and $p(\mathbf{x}_V)$ for the acoustic and visual components, respectively. The probability of a sample $(\mathbf{x}_A, \mathbf{x}_V)$ belonging to digit class c is then given by $p_A(\mathbf{x}_A|c)$ and $p_V(\mathbf{x}_V|c)$.

As we wish to test the effect of noise in the audio channel, acoustic noise ranging from -5dB to 25dB signal-to-noise ratio (SNR) in steps of 5 dB is added to the test data. We use additive white Gaussian noise with zero mean and variance

$$\sigma_\eta^2 = 10^{\frac{-\text{SNR}}{10}}. \tag{35}$$

The acoustic and visual GMMs are combined into a single classifier by exponentially weighting each GMM in proportion to an estimate of the information content in each stream. As the result no longer represent probabilities we use the term *score*. For a given digit we get the combined audio-visual model

$$\text{Score}(\mathbf{x}_{AV}|c) = p(\mathbf{x}_A|c)^{\lambda_A} p(\mathbf{x}_V|c)^{\lambda_V} \tag{36}$$

where

$$0 \leq \lambda_A \leq 1 \tag{37}$$

$$0 \leq \lambda_V \leq 1 \tag{38}$$

and

$$\lambda_A + \lambda_V = 1. \tag{39}$$

Note that (36) is equivalent to a linear combination of log likelihoods.

The stream exponents cannot be determined through a maximum likelihood estimation, as this will always result in a solution with the modality having the largest probability being assigned a weight of 1 and the other 0. Instead, we discriminatively estimate the stream exponents. As the number of classes in our experiment is relatively small we perform this optimization using a brute-force grid search, directly minimizing the misclassification rate. Due to the constraint (39) it is only necessary to vary λ_A from 0 to 1. The corresponding λ_V will then be given by $1 - \lambda_A$. We vary λ_A from 0 to 1 in steps of 0.1. The set of parameters λ_A and λ_V that results in the lowest misclassification rate are chosen as optimum parameters.

In the experiment we perform classification for each of the SNR levels using (36) and calculate the average misclassification rate. We compare audio-only, visual-only, and audio-visual classifiers. For the audio-only classifier the stream weights are $\lambda_A = 1$ and $\lambda_V = 0$ and for visual-only $\lambda_A = 0$ and $\lambda_V = 1$. For the audio-visual classifier the discriminatively trained stream weights are used. Figure 8 shows average misclassification rate for the different models and noise levels.

From the results we observe that the visual channel does contain information relevant to speech, but that visual speech is not in itself sufficient for speech recognition. However, by combining acoustic and visual speech we are able to increase recognition performance above that of audio-only speech recognition, especially the presence of acoustic noise.

Conclusion

In this paper we propose a basic AVASR system that uses MFCCs as acoustic features, AAM parameters as visual features, and GMMs for modeling the distribution of audio-visual speech feature data. We present the EM and VB algorithms as two alternatives for learning the audio-visual speech GMMs and demonstrate how VB is less affected than EM by overfitting while leading to automatic model complexity selection.

The AVASR system is implemented in Python using SciPy and tested using the CUAVE database. Based on the results we conclude that the visual channel does contain relevant speech information, but is not in itself sufficient for speech recognition. However, by combining features of visual speech with audio

features, we find that AVASR gives better performance than audio-only speech recognition, especially in noisy environments.

Acknowledgments

The authors wish to thank MIH Holdings for funding the research presented in this paper and for granting permission to contribute the research source code to SciPy.

REFERENCES

[Dav80] S. Davis, I. Matthews. *Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences*, IEEE Transactions on Acoustics, Speech, and Signal Processing, 28(8),357-366, 1980

[Luc81] B.D. Lucas, T. Kanade. *An iterative image registration technique with an application to stereo vision*, Proceedings of Imaging understanding workshop, 121-130, 1981

[Coo98] T.F. Cootes, G.J. Edwards, C. J .Taylor, *Active appearance models*, Proceedings of the European Conference on Computer Vision, 1998

[Bak01] S. Baker and I. Matthews, *Lucas Kanade 20 Years On: A Unifying Framework*, International Journal of Computer Vision, 2000

[Pat02] E.K. Patterson, S. Gurbuz, Z. Tufekci, J.N. Gowdy, *CUAVE: A new audio-visual database for multimodeal human-compuer inferface research*, 2002

[Mat03] I. Matthews, S. Baker, *Active Appearance Models Revisited*, International Journal of Computer Vision, 2003

[Bis07] C.M. Bishop. *Pattern recognition and machine learning*, Springer, 2007

Statsmodels: Econometric and Statistical Modeling with Python

Skipper Seabold[§*], Josef Perktold[‡]

Abstract—*Statsmodels* is a library for statistical and econometric analysis in Python. This paper discusses the current relationship between statistics and Python and open source more generally, outlining how the *statsmodels* package fills a gap in this relationship. An overview of *statsmodels* is provided, including a discussion of the overarching design and philosophy, what can be found in the package, and some usage examples. The paper concludes with a look at what the future holds.

Index Terms—statistics, econometrics, R

Introduction

Statsmodels (http://statsmodels.sourceforge.net/) is a library for statistical and econometric analysis in Python[1]. Its intended audience is both theoretical and applied statisticians and econometricians as well as Python users and developers across disciplines who use statistical models. Users of R, Stata, SAS, SPSS, NLOGIT, GAUSS or MATLAB for statistics, financial econometrics, or econometrics who would rather work in Python for all its benefits may find *statsmodels* a useful addition to their toolbox. This paper introduces *statsmodels* and is aimed at the researcher who has some prior experience with Python, NumPy/SciPy [SciPy][2].

On a historical note, *statsmodels* was started by Jonathan Taylor, a statistician now at Stanford, as part of SciPy under the name *models*. Eventually, *models* was removed from SciPy and became part of the NIPY neuroimaging project [NIPY] in order to mature. Improving the *models* code was later accepted as a SciPy-focused project for the Google Summer of Code 2009 and again in 2010. It is currently distributed as a SciKit, or add-on package for SciPy.

The current main developers of *statsmodels* are trained as economists with a background in econometrics. As such, much of the development over the last year has focused on econometric applications. However, the design of *statsmodels* follows a consistent pattern to make it user-friendly and easily extensible by developers from any discipline. New contributions and ongoing work are making the code more useful for common statistical modeling needs. We hope that continued efforts will result in a package useful for all types of statistical modeling.

* Corresponding author: jsseabold@gmail.com
§ American University
‡ CIRANO, University of North Carolina Chapel Hill

The State of the Union: Open Source and Statistics

Currently R is the open source language of choice for applied statistics. In applied econometrics, proprietary software pacakges such as Gauss, MATLAB, Stata, SAS, and NLOGIT remain the most popular and are taught in most graduate programs. However, there is a growing call for the use of open source software in economic research due in large part to its reliability, transparency, and the paradigm it offers for workflow and innovation [YaltaYalta], [YaltaLucchetti]. In particular R is increasing in popularity as evidenced by the recent textbooks by Cryer and Chan (2008), Kleiber and Zeileis (2008), and Vinod (2008). Gretl is another notable open source alternative for econometrics [Gretl].

However, there are those who would like to see Python become the language of choice for economic research and applied econometrics. Choirat and Seri's "Econometrics with Python" is the first publication of which we are aware that openly advocates the use of Python as the language of choice for econometricians [ChoiratSeri]. Bilina and Lawford express similar views [BilinaLawford]. Further, John Stachurski has written a Python-based textbook, *Economic Dynamics: Theory and Computation* [Stachurski], and Alan Isaac's "Simulating Evolutionary Games: a Python-Based Introduction" showcases Python's abilities for implementing agent-based economic models [Isaac].

In depth arguments for the choice of Python are beyond the scope of this paper; however, Python is well known for its simple syntax, gentle learning curve, and large standard library. Beyond this, it is a language for much more than statistics and can be the one toolbox for researchers across discplines. A few examples of statistics-related packages that are outside of the main numpy/scipy code are packages for Markov Chain Monte Carlo and Bayesian statistics [PyMC], machine learning and multivariate pattern analysis [scikits-learn], [PyMVPA], neuroimaging [NIPY] and neuroscience time series [NITIME], visualization [Matplotlib], [Enthought], and efficient handling of large datasets [PyTables].

We hope that *statsmodels* too can become an integral a part of the Scientific Python community and serve as a step in the direction of Python becoming a serious open source language for statistics. Towards this end, others are working on an R-like formula framework to help users specify and manipulate models [charlton], and packages like pandas [pandas] (discussed in these proceedings) and larry [larry] are providing flexible data structures and routines for data analysis currently lacking in NumPy.

Statsmodels: Development and Design

It should not be the case that different conclusions can be had from the same data depending on the choice of statistical software or its version; however, this is precisely what Altman and MacDonald (2003) find [AltmanMcDonald]. Given the importance of numerical accuracy and replicablity of research and the multitude of software choices for statistical analysis, the development of *statsmodels* follows a process to help ensure accurate and transparent results. This process is known as Test-Driven Development (TDD). In its strictest form, TDD means that tests are written before the functionality which it is supposed to test. While we do not often take the strict approach, there are several layers in our development process that ensure that our results are correct versus existing software (often R, SAS, or Stata). Any deviations from results in other software are noted in the test suite.

First, we employ a distributed version control system in which each developer has his own copy of the code, a branch, to make changes outside of the main codebase. Once a model is specified, early changes, such as working out the best API or bug hunting, take place in the main branch's, or trunk's, sandbox directory so that they can be tried out by users and developers who follow the trunk code. Tests are then written with results taken from another statistical package or Monte Carlo simulation when it is not possible to obtain results from elsewhere. After the tests are written, the developer asks for a code review on our mailing list (http://groups.google.ca/group/pystatsmodels). When all is settled, the code becomes part of the main codebase. Periodically, we release the software in the trunk for those who just want to download a tarball or install from PyPI, using *setuptools*' easy_install. This workflow, while not foolproof, helps make sure our results are and remain correct. If they are not, we are able to know why and document discrepancies resulting in the utmost transparency for end users. And if all else fails, looking at the source code is trivial to do (and encouraged!).

The design of the package itself is straightforward. The main idea behind the design is that a model is itself an object to be used for data reduction. Data can be both endogenous and exogenous to a model, and these constituent parts are related to each other through statistical theory. This statistical relationship is usually justified by an appeal to discipline-specific theory. Note that in place of endogenous and exogenous, one could substitute the terms dependent and independent variables, regressand and regressors, response and explanatory variables, etc., respectively, as you prefer. We maintain the endogenous-exogenous terminology throughout the package, however.

With this in mind, we have a base class, *Model*, that is intended to be a template for parametric models. It has two main attributes *endog* and *exog*[3] and placeholders for *fit* and *predict* methods. *LikelihoodModel* is a subclass of *Model* that is the workhorse for the regression models. All *fit* methods are expected to return some results class. Towards this end, we also have a base class *Results* and *LikelihoodModelResults* which inherits from *Results*. The result objects have attributes and methods that contain common post-estimation results and statistical tests. Further, these are computed lazily, meaning that they are not computed until the user asks for them so that those who are only interested in, say, the fitted parameters are not slowed by computation of extraneous results. Every effort is made to ensure that the constructors of each sublcass of *Model*, the call signatures of its methods, and the post-estimation results are consistent throughout the package.

Package Overview

Currently, we have five modules in the main codebase that contain statistical models. These are *regression* (least squares regression models), *glm* (generalized linear models), *rlm* (robust linear models), *discretemod* (discrete choice models), and *contrast* (contrast analysis). *Regression* contains generalized least squares (*GLS*), weighted least squares (*WLS*), and ordinary least squares (*OLS*). *Glm* contains generalized linear models with support for six common exponential family distributions and at least ten standard link functions. *Rlm* supports M-estimator type robust linear models with support for eight norms. *Discretemod* includes several discrete choice models such as the *Logit*, *Probit*, Multinomial Logit (*MNLogit*), and *Poisson* within a maximum likelihood framework. *Contrast* contains helper functions for working with linear contrasts. There are also tests for heteroskedasticity, autocorrelation, and a framework for testing hypotheses about linear combinations of the coefficients.

In addition to the models and the related post-estimation results and tests, *statsmodels* includes a number of convenience classes and functions to help with tasks related to statistical analysis. These include functions for conveniently viewing descriptive statistics, a class for creating publication quality tables, and functions for translating foreign datasets, currently only Stata's binary *.dta* format, to numpy arrays.

The last main part of the package is the datasets. There are currently fourteen datasets that are either part of the public domain or used with express consent of the original authors. These datasets follow a common pattern for ease of use, and it is trivial to add additional ones. The datasets are used in our test suite and in examples as illustrated below.

Examples

All of the following examples use the datasets included in *statsmodels*. The first example is a basic use case of the OLS model class to get a feel for the rest of the package, using Longley's 1967 dataset [Longley] on the US macro economy. Note that the Longley data is known to be highly collinear (it has a condition number of 456,037), and as such it is used to test accuracy of least squares routines than to examine any economic theory. First we need to import the package. The suggested convention for importing *statsmodels* is

```
>>> import scikits.statsmodels as sm
```

Numpy is assumed to be imported as:

```
>>> import numpy as np
```

Then we load the example dataset.

```
>>> longley = sm.datasets.longley
```

The datasets have several attributes, such as descriptives and copyright notices, that may be of interest; however, we will just load the data.

```
>>> data = longley.load()
```

Many of the *Dataset* objects have two attributes that are helpful for tests and examples -*endog* and *exog*- though the whole dataset is available. We will use them to construct an *OLS* model instance. The constructor for *OLS* is

```
def __init__(self, endog, exog)
```

It is currently assumed that the user has cleaned the dataset and that a constant is included, so we first add a constant and then instantiate the model.

```
>>> data.exog = sm.add_constant(data.exog)
>>> longley_model = sm.OLS(data.endog, data.exog)
```

We are now ready to fit the model, which returns a *RegressionResults* class.

```
>>> longley_res = longley_model.fit()
>>> type(longley_res)
<class 'sm.regression.RegressionResults'>
```

By default, the least squares models use the pseudoinverse to compute the parameters that solve the objective function.

```
>>> params = np.dot(np.linalg.pinv(data.exog),
                    data.endog)
```

The instance *longley_res* has several attributes and methods of interest. The first is the fitted values, commonly β in the general linear model, $Y = X\beta$, which is called *params* in *statsmodels*.

```
>>> longley_res.params
array([  1.50618723e+01, -3.58191793e-02,
    -2.02022980e+00,  -1.03322687e+00,
    -5.11041057e-02,   1.82915146e+03,
    -3.48225863e+06])
```

Also available are

```
>>> [_ for _ in dir(longley_res) if not
     _.startswith('_')]
['HC0_se', 'HC1_se', 'HC2_se', 'HC3_se', 'aic',
'bic', 'bse', 'centered_tss', 'conf_int',
'cov_params', 'df_model', 'df_resid', 'ess',
'f_pvalue', 'f_test', 'fittedvalues', 'fvalue',
'initialize', 'llf', 'model', 'mse_model',
'mse_resid', 'mse_total', 'nobs', 'norm_resid',
'normalized_cov_params', 'params', 'pvalues',
'resid', 'rsquared', 'rsquared_adj', 'scale', 'ssr',
'summary', 't', 't_test', 'uncentered_tss', 'wresid']
```

All of the attributes and methods are well-documented in the docstring and in our online documentation. See, for instance, help(longley_res). Note as well that all results objects carry an attribute *model* that is a reference to the original model instance that was fit whether or not it is instantiated before fitting.

Our second example borrows from Jeff Gill's *Generalized Linear Models: A Unified Approach* [Gill]. We fit a Generalized Linear Model where the endogenous variable has a binomial distribution, since the syntax differs somewhat from the other models. Gill's data comes from the 1998 STAR program in California, assessing education policy and outcomes. The endogenous variable here has two columns. The first specifies the number of students above the national median score for the math section of the test per school district. The second column specifies the number of students below the median. That is, *endog* is (number of sucesses, number of failures). The explanatory variables for each district are measures such as the percentage of low income families, the percentage of minority students and teachers, the median teacher salary, the mean years of teacher experience, per-pupil expenditures, the pupil -teacher ratio, the percentage of student taking college credit courses, the percentage of charter schools, the percent of schools open year round, and various interaction terms. The model can be fit as follows

```
>>> data = sm.datasets.star98.load()
>>> data.exog = sm.add_constant(data.exog)
>>> glm_bin = sm.GLM(data.endog, data.exog,
                     family=sm.families.Binomial())
```

Note that you must specify the distribution family of the endogenous variable. The available families in *scikits.statsmodels.families* are *Binomial*, *Gamma*, *Gaussian*, *InverseGaussian*, *NegativeBinomial*, and *Poisson*.

The above examples also uses the default canonical logit link for the Binomial family, though to be explicit we could do the following

```
>>> links = sm.families.links
>>> glm_bin = sm.GLM(data.endog, data.exog,
        family=sm.families.Binomial(link=
        links.logit))
```

We fit the model using iteratively reweighted least squares, but we must first specify the number of trials for the endogenous variable for the Binomial model with the endogenous variable given as (success, failure).

```
>>> trials = data.endog.sum(axis=1)
>>> bin_results = glm_bin.fit(data_weights=trials)
>>> bin_results.params
array([ -1.68150366e-02,   9.92547661e-03,
    -1.87242148e-02,  -1.42385609e-02,
     2.54487173e-01,   2.40693664e-01,
     8.04086739e-02,  -1.95216050e+00,
    -3.34086475e-01,  -1.69022168e-01,
     4.91670212e-03,  -3.57996435e-03,
    -1.40765648e-02,  -4.00499176e-03,
    -3.90639579e-03,   9.17143006e-02,
     4.89898381e-02,   8.04073890e-03,
     2.22009503e-04,  -2.24924861e-03,
     2.95887793e+00])
```

Since we have fit a GLM with interactions, we might be interested in comparing interquartile differences of the response between groups. For instance, the interquartile difference between the percentage of low income households per school district while holding the other variables constant at their mean is

```
>>> means = data.exog.mean(axis=0) # overall means
>>> means25 = means.copy() # copy means
>>> means75 = means.copy()
```

We can now replace the first column, the percentage of low income households, with the value at the first quartile using scipy.stats and likewise for the 75th percentile.

```
>>> from scipy.stats import scoreatpercentile as sap
>>> means25[0] = sap(data.exog[:,0], 25)
>>> means75[0] = sap(data.exog[:,0], 75)
```

And compute the fitted values, which are the inverse of the link function at the linear predicted values.

```
>>> lin_resp25 = glm_bin.predict(means25)
>>> lin_resp75 = glm_bin.predict(means75)
```

Therefore the percentage difference in scores on the standardized math tests for school districts in the 75th percentile of low income households versus the 25th percentile is

```
>>> print "%4.2f percent" % ((lin_resp75-
                             lin_resp25)*100)
-11.88 percent
```

The next example concerns the testing of joint hypotheses on coefficients and is inspired by a similar example in Bill Greene's *Econometric Analysis* [Greene]. Consider a simple static investment function for a macro economy

$$\ln I_t = \beta_1 + \beta_2 \ln Y_t + \beta_3 i_t + \beta_4 \Delta p_t + \beta_5 t + \varepsilon_t \qquad (1)$$

In this example, (log) investment, I_t is a function of the interest rate, i_t, inflation, Δp_t, (log) real GDP, Y_t, and possibly follows a

linear time trend, t. Economic theory suggests that the following model may instead be correct

$$\ln I_t = \beta_1 + \ln Y_t + \beta_3 (i_t - \Delta p_t) + \varepsilon_t \quad (2)$$

In terms of the (1) this implies that $\beta_3 + \beta_4 = 0$, $\beta_2 = 1$, and $\beta_5 = 0$. This can be implemented in *statsmodels* using the *macrodata* dataset. Assume that *endog* and *exog* are given as in (1)

```
>>> inv_model = sm.OLS(endog, exog).fit()
```

Now we need to make linear restrictions in the form of $R\beta = q$

```
>>> R = [[0,1,0,0,0],[0,0,1,1,0],[0,0,0,0,1]]
>>> q = [1,0,0]
```

$R\beta = q$ implies the hypotheses outlined above. We can test the joint hypothesis using an F test, which returns a *ContrastResults* class

```
>>> Ftest = inv_model.f_test(R,q)
>>> print Ftest
<F test: F=array([[ 194.4428894]]),
p=[[  1.27044954e-58]], df_denom=197, df_num=3>
```

Assuming that we have a correctly specified model, given the high value of the F statistic, the probability that our joint null hypothesis is true is essentially zero.

As a final example we will demonstrate how the *SimpleTable* class can be used to generate tables. SimpleTable is also currently used to generate our regression results summary. Continuing the example above, one could do

```
>>> print inv_model.summary(yname="lninv",
            xname=["const","lnY","i","dP","t"])
```

To build a table, we could do:

```
>>> gdpmean = data.data['realgdp'].mean()
>>> invmean = data.data['realinv'].mean()
>>> gdpstd = data.data['realgdp'].std()
>>> invstd = data.data['realinv'].std()
>>> mydata = [[gdpmean, gdpstd],[invmean,
        invstd]]
>>> myheaders = ["Mean", "Std Dev."]
>>> mystubs = ["Real GDP", "Real Investment"]
>>> tbl = sm.iolib.SimpleTable(mydata,
        myheaders, mystubs, title =
        "US Macro Data", data_fmts=['%4.2f'])
>>> print tbl
         US Macro Data
================================
                 Mean  Std Dev.
--------------------------------
Real GDP        7221.17 3207.03
Real Investment 1012.86  583.66
--------------------------------
```

LaTeX output can be generated with something like

```
>>> fh = open('./tmp.tex', 'w')
>>> fh.write(tbl.as_latex_tabular())
>>> fh.close()
```

While not all of the functionality of *statsmodels* is covered in the above, we hope it offers a good overview of the basic usage from model to model. Anything not touched on is available in our documentation and in the examples directory of the package.

Conclusion and Outlook

Statsmodels is very much still a work in progress, and perhaps the most exciting part of the project is what is to come. We currently have a good deal of code in our sandbox that is being cleaned up, tested, and pushed into the main codebase as part of the Google Summer of Code 2010. This includes models for time-series analysis, kernel density estimators and nonparametric regression, panel or longitudinal data models, systems of equation models, and information theory and maximum entropy models.

We hope that the above discussion gives some idea of the appoach taken by the project and provides a good overview of what is currently offered. We invite feedback, discussion, or contributions at any level. If you would like to get involved, please join us on our mailing list available at http://groups.google.com/group/pystatsmodels or on the scipy-user list. If you would like to follow along with the latest development, the project blog is http://scipystats.blogspots.com and look for release announcements on the scipy-user list.

All in all, we believe that the future for Python and statistics looks bright.

Acknowledgements

In addition to the authors of this paper, many others have contributed to the codebase. Thanks are due to Jonathan Taylor and contributors to the models code while it was in SciPy and NIPY. Thanks also go to those who have provided code, constructive comments, and discussion on the mailing lists.

References

[AltmanMcDonald] M. Altman and M.P. McDonald. 2003. "Replication with Attention to Numerical Accuracy." *Political Analysis*, 11.3, 302-7.

[BilinaLawford] R. Bilina and S. Lawford. July 4, 2009. *Python for Unified Research in Econometrics and Statistics*, July 4, 2009. Available at SSRN: http://ssrn.com/abstract=1429822

[charlton] Charlton. Available at https://github.com/charlton

[ChoiratSeri] C. Choirat and R. Seri. 2009. "Econometrics with Python." *Journal of Applied Econometrics*, 24.4, 698-704.

[CryerChan] J.D. Cryer and K.S. Chan. 2008. *Time Series Analysis: with Applications in R*, Springer.

[Enthought] Enthought Tool Suite. Available at http://code.enthought.com/.

[Gill] J. Gill. 2001. *Generalized Linear Models: A Unified Approach*. Sage Publications.

[Greene] W. H. Greene. 2003. *Econometric Analysis* 5th ed. Prentice Hall.

[Gretl] Gnu Regression, Econometrics, and Time-series Library: gretl. Available at http://gretl.sourceforge.net/.

[Isaac] A.G. Isaac. 2008. "Simulating Evolutionary Games: a Python- Based Introduction." *Journal of Artificial Societies and Social Simulation*. 11.3.8. Available at http://jasss.soc.surrey.ac.uk/11/3/8.html

[KleiberZeileis] C. Kleiber and A. Zeileis. 2008. *Applied Econometrics with R*, Springer.

[Longley] J.W. Longley. 1967. "An Appraisal of Least Squares Programs for the Electronic Computer from the Point of View of the User." *Journal of the American Statistical Association*, 62.319, 819-41.

[Matplotlib] J. Hunter, *et al.* Matplotlib. Available at http://matplotlib.sourceforge.net/index.html.

1. The examples reflect the state of the code at the time of writing. The main model API is relatively stable; however, recent refactoring has changed the organization of the code. See online documentation for the current usage.

2. Users who wish to learn more about NumPy can do so at http://www.scipy.org/Tentative_NumPy_Tutorial, http://www.scipy.org/Numpy_Example_List, or http://mentat.za.net/numpy/intro/intro.html. For those coming from R or MATLAB, you might find the following helpful: http://mathesaurus.sourceforge.net/ and http://www.scipy.org/NumPy_for_Matlab_Users

3. The *exog* attribute is actually optional, given that we are developing support for (vector) autoregressive processes in which all variables could at times be thought of as "endogenous".

[larry] K.W. Goodman. Larry: Labeled Arrays in Python. Available at http://larry.sourceforge.net/.

[NIPY] NIPY: Neuroimaging in Python. Available at http://nipy.org.

[NITIME] Nitime: time-series analysis for neuroscience. Available at http://nipy.org/nitime

[pandas] W. McKinney. Pandas: A Python Data Analysis Library. Available at http://pandas.sourceforge.net/.

[PyMC] C. Fonnesbeck, D. Huard, and A. Patil, PyMC: Pythonic Markov chain Monte Carlo. Available at http://code.google.com/p/pymc/

[PyMVPA] M. Hanke, *et. al.* PyMVPA: Multivariate Pattern Analysis in Python. Available at http://www.pymvpa.org/.

[PyTables] F. Alted, I. Vilata, *et al.* PyTables: Hierarchical Datasets in Python. Available at http://www.pytables.org.

[scikits-learn] Pedregosa, F. *et al.* `scikits.learn`: machine learning in Python. Available at http://scikits-learn.sourceforge.net.

[SciPy] T. Oliphant, *et al.* SciPy. Available at http://www.scipy.org.

[Stachurski] J. Stachurski. 2009. *Economic Dynamics: Theory and Computation*. MIT Press.

[Vinod] H.D. Vinod. 2008. *Hands-on Intermediate Econometrics Using R*, World Scientific Publishing.

[YaltaLucchetti] A.T. Yalta and R. Lucchetti. 2008. "The GNU/Linux Platform and Freedom Respecting Software for Economists." *Journal of Applied Econometrics*, 23.3, 279-86.

[YaltaYalta] A.T. Yalta and A.Y. Yalta. 2010. *Should Economists Use Open Source Software for Doing Research*. Computational Economics, 35, 371-94.