0.1 Overview

SPIVET (Stereoscopic Particle Image VElocimetry and Thermometry) is a Python package providing a series of tools for analyzing stereoscopic PIV image sequences of experimental fluid or particle flows. The functionality provided by the package serves four primary purposes:

1. Extraction of a displacement (or velocity) field from raw PIV images.
2. Extraction of temperature values from images taken of thermochromic liquid crystals (TLC's).
3. Passive tracer advection framework for Lagrangian transport studies and computation of finite-time Lyapunov exponent fields.
4. Aggregation and storage of processed results in a portable file format that is compatible with a variety of visualization software suites (*e.g.*, ParaView [1], and VisIt [3]).

SPIVET services are provided by a collection of Python modules as described in the Architecture section (Section 0.3). At present, SPIVET does not have a graphical user interface, but is instead used via the command line and Python scripts. Example scripts are provided to get the user moving in the correct direction.

The present document is meant to provide the user with a introductory perspective of SPIVET, its structure, and its capabilities. Detailed module and function documentation is provided in the code and can be retrieved using Python's `help` function or by reading the code itself. A bibliography is also provided at the root of the source tree.

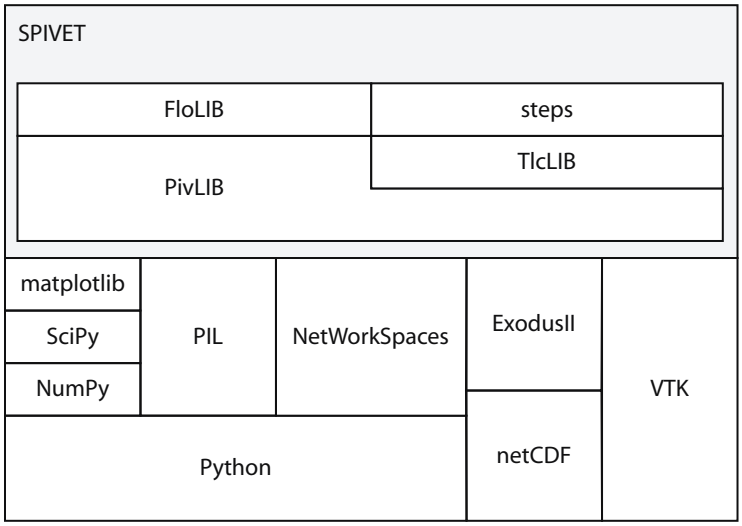

Figure 1: Architectural overview of SPIVET. Primary dependencies on third party software are also shown (see Section 0.7).

0.2 Licensing

SPIVET is released under the terms of Version 2 of the GNU Public License (GPL). A copy of the license is provided in the `LICENSE.SPIVET` file at the root of the source tree.

0.3 Architecture

SPIVET is an aggregate Python package composed of four principal lower-level components: PivLIB, TlcLIB, FloLIB, and SPIVET `steps`. A graphical depiction of SPIVET's architecture is shown in Figure 1.

The choice of Python as the primary language for SPIVET was motivated in large part by Python's:

- clear, compact, and easy to learn syntax,
- superb ability to glue disparate applications and libraries together into

one cohesive whole,

- and ease of quickly visualizing data via directly callable packages such matplotlib[4] (also referred to as pylab) and VTK [2].

However, Python does have some drawbacks, the largest of which is shared with most other interpreted languages: namely, poor execution speed as compared to a compiled language such as C. Python's performance penalty is particularly acute when executing loops over large data arrays. In such loops, Python code can easily run an order of magnitude or more slower than comparable compiled code. There are two dominant methods for avoiding such overhead: vectorized statements and external C modules.

Often the cleanest, most compact way of negating performance penalties associated with Python loops is to use 'vectorized' statements which have an implicit loop that is often implemented in lower-level compiled code. To leverage this approach, much of SPIVET's internal workings are built around NumPy [6] `ndarray`s (n-dimensional array class). Using NumPy `ndarray` objects, two similarly shaped arrays can be added and results stored in a third array with the single Python statement: `arrayc = arraya + arrayb`. Not only is the preceding Python code substantially faster than an explicit Python `for` loop, but it is cleaner and easier to read. Nevertheless, there are cases where vectorized statements are difficult to formulate or explicit loops are unavoidable. In these instances, SPIVET implements the necessary functionality in an external C module and subsequently wraps the module in Python.

SPIVET has been constructed to utilize parallel processing for extracting data from very large sequences of images[1]. Here again, Python has strengths and weaknesses. Threading of pure Python code is very problematic. A Python con-

[1] At present, SPIVET's use of parallel processing is limited to the reduction of raw images to flow field variables, as this is the most time consuming operation. Nonetheless, there are plenty of opportunities to utilize parallel programming in other parts of SPIVET (see Section 0.8).

struct known as the Global Interpreter Lock (GIL) effectively serializes Python code executing within a single process[2]. To work around these limitations and still provide parallel processing for large image sets, SPIVET relies on the NetWorkSpaces [8] library to provide process-level parallelism and inter-process communication. Overall, SPIVET in cooperation with NetWorkSpaces functions very similar to a parallel program utilizing MPI [5]. NetWorkSpaces manages the spawning of individual worker processes on each processor of a networked grid of computers, and coordinates the exchange of data between worker processes. By using process-level concurrency, the serializing effect of the GIL is avoided since each process has its own, private GIL.

A good deal of effort has been spent to minimize, where possible, the virtual memory footprint of SPIVET. SPIVET's internal data structures (Section 0.3.1) attempt to intelligently manage their utilization of virtual memory by temporarily storing dormant data to disk on a least recently used basis. This functionality frees virtual memory address space to be utilized for other purposes, and is of primary benefit to installations that still rely on a Python built for a 32-bit address space. The issue of virtual memory address space exhaustion will all but disappear once 64-bit operating systems and user-level applications (like Python) become standard.

The function of each of the four principal SPIVET sub-packages as shown in Figure 1 is discussed in the following sub-sections. SPIVET's third-party dependencies are covered more fully in Section 0.7.

[2]Python places few constraints on external libraries written in a compiled language, and these libraries are free to employ threads, MPI [5], or any other concurrent execution techniques as long as the code does not call back into the Python API to interact with a Python object.

exodusII	Path containing the ExodusII library
__init__.py	PivLIB initialization module
exodusII.py	Python wrapper module for ExodusII library
pivcolor.py	Colormaps for use with pylab
pivdata.py	SPIVET data structures
pivir.py	Image registration functions
pivlibc.c	Numerically intensive C functions
pivlinalg.py	Streamlined LAPACK wrappers
pivof.py	Optical flow driver functions
pivpg.py	Photogrammetry functions
pivpgcal.py	Photogrammetric calibration functions
pivpickle.py	Pickling functions that compress data
pivpost.py	Filtering and spurious vector removal
pivsim.py	Ray tracing optical simulator
pivsimc.c	C functions for ray tracing
pivtpsc.c	C functions for thin-plate splines and shape contexts
pivutil.py	Miscellaneous utilities, including the reading and writing of images, used by many modules

Table 1: PivLIB package contents.

0.3.1 PivLIB

The PivLIB package provides the core algorithms for Particle Image Velocimetry and other essential SPIVET functionality. These services include camera calibration, extraction of displacement vectors from images, reconstruction of 3D displacements from 2D displacements (the stereoscopic aspect of PIV), post processing of displacement fields via filtering, reading and writing of images, a ray tracing simulator, and SPIVET data structures. The functionality of the PivLIB package is spread across the files as shown in Table 1. A few notes follow.

SPIVET data structures

Internally, SPIVET stores non-image PIV data (*e.g.*, velocity) in a hierarchy of three, specialized, intelligent data structures that have been constructed to minimize the virtual memory footprint of SPIVET. The lowest level data structure

is the `PIVVar` (a class derived from a NumPy `ndarray` [6]) which holds a single flow field variable (*e.g.*, temperature, velocity, viscous stress tensor, etc) along with the variable's name and units. Because the `PIVVar` class is derived from a NumPy `ndarray`, the full compliment of NumPy and SciPy functionality or any other framework that utilizes `ndarray`s can operate directly on a `PIVVar`. And just like `ndarray` objects, `PIVVar`s can be operands of standard arithmetic operators as shown in the following Python code snippet[3]

```
vara = varb + varc
```

The data stored in a `PIVVar` is ordered according to the underlying uniform grid of PIV cells (see Section 0.4). `PIVVar` objects know how to temporarily store their contents to disk thereby freeing up virtual memory address space. The management of these 'active' and 'deactivated' `PIVVar` objects is provided by the SPIVET `PIVEpoch` container class.

In many experimental cases, multiple flow field variables are of interest. Indeed, SPIVET has been specifically crafted to extract velocity and temperature fields from PIV image sets, and the user may wish to compute further derived quantities. To this end, the `PIVEpoch` is a container class that stores the full set of flow variables which are valid at a given instant in time. The `PIVEpoch` class is derived from the standard Python dictionary and stores `PIVVar` objects by the variable's associated name. The `PIVEpoch` class autonomously maintains a cache of recently used `PIVVar`s, while transparently storing unused variables to disk.

The set of `PIVEpoch`s that describe the time evolution of a flow field are stored in the highest level container, the `PIVData` object. The `PIVData` class derives from the Python list and is a simple ordered container that can write

[3]The user should note, however, that although the newly created variable `vara` is a valid `PIVVar`, it has no name or units. The units and name of a `PIVVar` created by an arithmetic operation is by default set to the string `'NOTSET'`. In such cases, the `PIVVar` method `setattr()` can be used to assign a meaningful name and units.

its full contents to disk as an ExodusII [7] file. The `PIVData` object is also responsible for tracking cell size and mesh origin within SPIVET's internal world coordinate system (see Section 0.4).

Permanent storage and data exchange

For permanent, external storage of flow field data, SPIVET uses the ExodusII library [7] which writes all data stored in a `PIVData` object to a single, portable `ex2` file that can be directly loaded into visualization packages such as VisIt [3] or ParaView [1]. The ExodusII file format has been developed by Sandia National Laboratories for storage of large, time-varying datasets expressed on unstructured grids. SPIVET represents its data on a uniform, cell-centered, structured mesh, so the unstructured mesh handling facilities of the ExodusII file format are not of particular benefit. Nevertheless, the portability, clean API, and ExodusII reliance on the NetCDF framework [11] for the underlying file structure are all significant advantages. An ExodusII file written by a SPIVET `PIVData` object is, for all intents and purposes, a disk-based incarnation of the `PIVData` object itself. Hence, SPIVET can easily 'reanimate' any `PIVData` object stored in and ExodusII file for additional processing at a later time.

There is one particular peculiarity of the ExodusII file format that the user should be aware of: variable names (with units appended) are currently limited to 32 characters. When SPIVET writes an ExodusII file, each ExodusII variable is assigned a name (limited to a maximum length of 32 characters) that is the concatenation of the corresponding `PIVVar` variable name, a space, and the `PIVVar` units enclosed in brackets (*i.e.,* `[]`).

Optical simulations

The `pivsim` module provides an optically correct ray tracer for experimental simulation. The functionality of the module is limited, but it has been used to

__init__.py	TlcLIB initialization module
tlclibc.c	Numerically intensive C functions for TlcLIB
tlctc.py	Thermochromic functions that utilize an existing calibration
tlctccal.py	Thermochromic calibration
tlctf.py	Temperature extraction driver functions
tlcutil.py	Miscellaneous utility functions

Table 2: TlcLIB package contents.

do a full simulation of the entire PIV process, from photogrammetric calibration to a synthetic 'experiment.' In this manner, the module provides a simple means of investigating the relative importance of different factors on the aggregate PIV setup. Ray tracing simulations also permit accuracy analysis of PivLIB's PIV vector extraction algorithms in a well-controlled environment. However, the user should understand that such accuracy analyses often represent a floor since `pivsim` does not include errors from lens optics (e.g., spherical aberration, finite depth of field), camera noise, etc. These un-modeled errors are typically non-negligible and can actually be the dominant source.

0.3.2 TlcLIB

The TlcLIB package provides thermochromic liquid crystal (TLC) calibration and temperature field extraction facilities and is organized as shown in Table 2.

0.3.3 SPIVET steps

The `spivet.steps` module and its helpers provide an object-oriented wrapper around the procedural functions contained in the PivLIB and TlcLIB packages that automate the process of converting raw images into desired flow field data. The `steps` module also separates user configuration data (contained in a 'recipe') from the function implementation details. Hence data processing scripts using `steps` are more compact since the script is only charged with con-

__init__.py	FloLIB initialization module
floftle.py	Finite time Lyapunov exponent (FTLE) functions
floftlec.c	C library of numerically intensive FTLE functions
flotrace.py	Passive tracer functions
flotracec.py	C library of numerically intensive tracing functions
floutil.py	Generic utilities for derivatives and Runge-Kutta time-stepping
flovars.py	Functions for computing derived flow quantities
svcismat.h	Header file for tricubic interpolation

Table 3: FloLIB package contents.

figuring, initializing, and setting up the order in which the collection of steps are executed. Once a particular recipe is constructed, it can then be applied with limited or no modifications to a number of experimental datasets.

A recipe constructed from a collection of steps is the principal means for reducing raw image data into useable flow field variables such as velocity and temperature. Individual steps are provided to parse the full set of image files, partition the set of images for parallel processing, extract field variables, and post-process those field variables to remove spurious vectors or compute derived quantities (*e.g.*, vorticity). Users can also create their own derived steps which are stored in the `.spivet/usersteps` path under the user's home directory.

0.3.4 FloLIB

The set of FloLIB modules provide post-processing facilities for flow field data, with the primary functionality being: 1) computation of derivatives for field data stored in `PIVVar`s, 2) spatial interpolation of field variables, and 3) Lagrangian transport analyses by way of passive tracer advection. FloLIB consists of the files shown in Table 3.

0.4 Coordinate system and array indexing

SPIVET generally orders data with the x-axis (i-index) varying fastest, followed by the y-axis (j-index), and finally the z-axis (k-index). Hence the Python statement

```
val = velocity[2,0,10,20]
```

where `velocity` is a `PIVVar`, assigns the x-component[4] of velocity in cell $(i, j, k) = (20, 10, 0)$ to the variable `val`. The only time this rule is broken is if SPIVET needs to pass information to an external library (*e.g.*, ExodusII) that requires a different ordering. In these special cases, the change is transparent to the user. Note that as with C, array indexing begins with zero in Python.

SPIVET employs several coordinate systems to convert raw images into useable flow field variables, however the three most important are undoubtedly local array coordinates, SPIVET's internal world coordinates, and `PIVData` coordinates. Local array coordinates are cell-centered and dimensionless with the coordinate system origin located at cell $(i, j) = (0, 0)$ for a 2D array (*e.g.*, an image) or $(i, j, k) = (0, 0, 0)$ for a 3D array (*e.g.*, a `PIVVar`). Local array coordinate axes point in the direction of increasing index.

Once photogrammetric calibration has been performed, SPIVET will project camera images back onto the plane of illumination (created by laser or lightsheet). This projection results in a dewarped representation of the camera image where all of the new 'world pixels' can be scaled by one single constant to represent distances in mm. SPIVET adopts a convention regarding the orientation

[4]Note that the velocity components (first dimension of the `PIVVar`) are also ordered as z, y, x. This is another SPIVET convention that is *always* followed for *all* ordered data (`PIVVar`s or otherwise). A nine-component tensor stored in the `PIVVar atensor`, for example, will have its components ordered as $zz, zy, zx, yz, yy, yx, xz, xy, xx$. Hence `atensor[0,:,:,:]` will return an array of the zz-tensor component and `atensor[8,:,:,:]` will return the xx-component. Likewise a three-element vector, `tcrd`, containing the x, y, z-coordinates for a tracer would be stored in SPIVET as `tcrd = array([zcoord,ycoord,xcoord])`. When interacting with SPIVET, this convention must always be observed.

of world coordinates that is common, but certainly not universal, in computer graphics and image processing applications. SPIVET takes the internal world coordinate system to be right-handed and to have its origin located at the upper left corner of the plane (or collection of planes in 3D) being imaged or viewed[5]. The internal world z-axis is perpendicular to this viewed plane, oriented such that z-coordinates increase away from the viewer (or camera). The internal world x- and y-axes are parallel to the local i- and j-axes, with x increasing to the viewer's right and$y$ increasing down. SPIVET is designed to process images taken of many planes within the flow field, where each plane is imaged by moving the light sheet and cameras as needed. SPIVET assumes this motion is parallel to the world z-axis. The origin of the world coordinate system is therefore located in the center of world pixel $(i, j) = (0, 0)$ (the upper left pixel) of the first plane imaged. SPIVET world coordinates have units of mm.

For flexibility, the `PIVData` coordinate origin does not have to be coincident with the SPIVET world coordinate origin. Since SPIVET is principally built around correlation particle image velocimetry (CPIV) techniques, extracted PIV displacement vectors represent an average displacement for a group of world pixels (also known as an interrogation window in the literature). Hence the cell-centered local array origin of a `PIVVar` corresponds to a location in the center of the interrogation window. For obvious reasons, choosing the origin of the `PIVData` coordinate system to be coincident with the underlying local array origin is very beneficial. To this end, the `PIVData` `origin` member stores the offset between the origins of the SPIVET world system (centered in world pixel $(i, j) = (0, 0)$) and `PIVData` coordinate system (centered in `PIVData` cell $(i, j, k) = (0, 0, 0)$).

[5]So when looking at a computer monitor, for example, the origin is the upper left corner of the screen using this convention.

Although the origins of `PIVData` and SPIVET world coordinates can be offset, SPIVET does expect the particular coordinate axes to be parallel *and* point in the same direction. For the University of Michigan setup, the laboratory equipment sweeps the light sheet (and cameras) in a direction moving opposite the cameras' gaze. According to the definition of SPIVET world coordinates above, the light-sheet and camera are both moving in the negative z-direction. Given that SPIVET stores the extracted data from each plane in the order in which it was acquired, the `PIVData` local array k-axis points in the opposite direction of the world z-axis. The apparent mismatch is accounted for by another `PIVData` member, `cellsz`, which stores the cell size of the underlying `PIVData` mesh in units of mm. Values stored in `cellsz` can be negative, as is the case for the University of Michigan setup. `PIVData` cell centers can then be represented in valid SPIVET world coordinates by the pseudocode

```
cell_coords = cell_indices*pd.cellsz + pd.origin
```

where `pd` is taken to be a valid `PIVData` object.

0.5 Installation

SPIVET has been installed and tested on Linux and Mac OS X platforms. Some operations used within SPIVET are likely not portable to Windows machines. The installation procedure is detailed in the `README` file located in the root of the source tree.

0.6 Filesystem layout

The SPIVET source code is organized as shown in Figure 2.

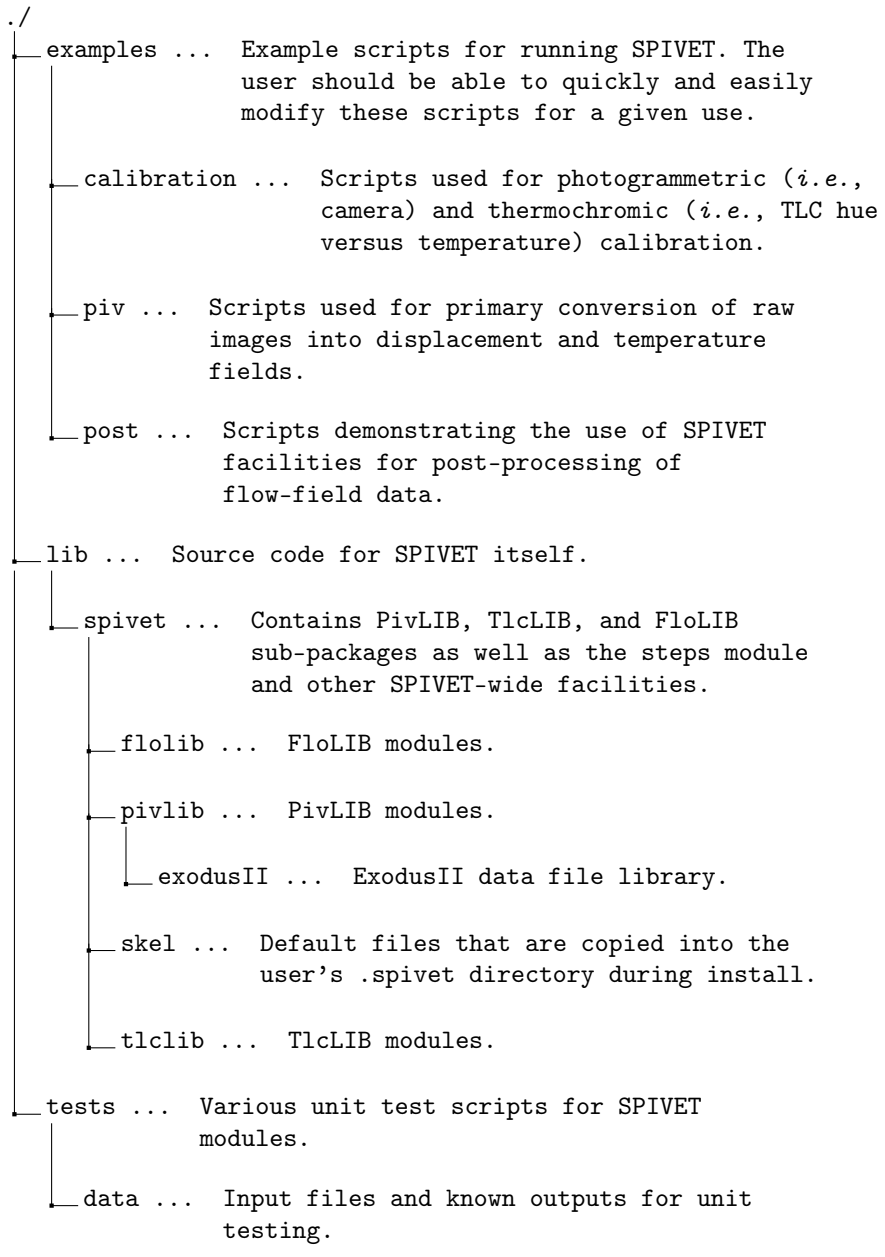

Figure 2: SPIVET source code directory structure.

0.7 Dependencies

SPIVET requires the following software to be installed. Unless otherwise noted, the latest version available should work.

- **Python** SPIVET has not been tested against Python version 3.0 or above. SPIVET was written for and works with Python 2.5 and 2.6. Python 3.0 and above is not backward compatible with the Python 2.x series. As a result, running SPIVET in Python 3.x will undoubtedly require some changes to the SPIVET code. See `http://docs.python.org/dev/3.0/whatsnew/3.0.html` for more info on the differences between Python 2.x and Python 3.x.
- **NumPy** Provides scientific computation facilities and efficient handling of arrays [6].
- **SciPy** SciPy provides higher level scientific computation facilities that are built on top of NumPy [9].
- **PIL** Python Imaging Library. PIL provides facilities for reading and writing images of various formats [10].
- **netCDF** Low-level file format [11] upon which the ExodusII specification [7] is built. Users only need to install netCDF as the ExodusII library is provided as part of SPIVET.
- **VTK** Visualization toolkit that provides storage, visualization, and geometric data manipulation capabilities [2].
- **matplotlib** Python package providing 2D plotting capabilities [4].
- **NetWorkSpaces** Provides the infrastructure for parallel processing [8]. NWS isn't required, but when processing large batches of images it's highly

recommended. An NWS installation must consist of the client (installed on all worker nodes) and the server (on one machine only). Note that the server does not have to be a worker node.

0.8 Avenues for improvement

While SPIVET generally does a reasonable job at tasks for which it was designed, there are a number of opportunities for improvement. The more important deficiencies are discussed in this section.

As discussed in the overview, SPIVET is currently without a graphical user interface (GUI). SPIVET was not designed with real-time processing or user interaction in mind. Instead, SPIVET is a library predominantly geared toward batch processing of large quantities of images, extracting flow field variables from those images, and then post processing the results as needed. For most of SPIVET's intended uses, a GUI would seem to provide little or no benefit. However a GUI that manages the creation, organization, and configuration of SPIVET `steps` recipes (see Section 0.3.3) would be quite valuable.

Significant improvement in execution speed and memory footprint could be realized by migrating some parts of the PivLIB image processing C code to single-precision arithmetic (or even integer arithmetic) and potentially vectorizing the corresponding code with SIMD instructions. External modules written in C or Fortran can use single- or double-precision arithmetic, and the NumPy `ndarray` class provides a mechanism for specifying whether a Python `ndarray` object holds single- or double-precision data. So facilities are available for working with single-precision data. Unfortunately, Python proper (with the exception of NumPy's `ndarray` class) is hardwired to use double-precision (*i.e.*, 64-bit) floating point arithmetic by default, so care must be taken to ensure that a Python coerced conversion from or to double-precision doesn't negate

any single-precision performance benefits.

Having an optically accurate ray tracing facility like that of `pivsim`, can be extremely valuable in analyzing experimental errors, developing extensions to existing PIV algorithms, and upstream planning for new laboratory experiments. The functionality provided by the `pivsim` module, however, is currently very limited. Commercial optical design packages provide a much larger feature set and have substantially more robust computational geometry kernels. However, these commercial packages tend to be expensive and require a substantial learning curve due to the very large feature set provided. It is this author's opinion that investing in a commercial package could be a wise investment since the software would provide the PIV practitioner with the opportunity to design and analyze experiments that optimally leverage equipment capabilities. Nevertheless, the simulation facilities provided by `pivsim` can still be useful particularly if run-time performance is improved. In the current configuration, `pivsim` is implemented utilizing Python objects constructed from a mixture of Python and C code. Code that is called repeatedly during ray tracing is fully implemented in C, but still uses the Python object model and corresponding API. Unfortunately, interacting with Python objects has substantial overhead regardless of whether the object is implemented in Python or C. The expense is a consequence of Python's interpreted nature and the resulting way in which class data and function members are accessed (using a string search of a dictionary). The performance of `pivsim` can be improved substantially by eliminating all vestiges of Python and migrating the entire existing implementation to C++. Python could then be used to call into this self-sufficient library to setup the simulation and trigger the actual ray tracing, however all other functionality should be implemented independently of the Python API.

The passive tracer framework of FloLIB is under enormous strain from the

perspectives of computational time and memory use. Advecting large numbers of passive tracers is by its very nature a computationally expensive and memory intensive undertaking. The algorithms as currently implemented in FloLIB provide a high degree of flexibility for advecting tracers that can later be used for a variety of purposes, but execution time and memory constraints effectively limit FloLIB's passive tracing facilities to 2D applications. High-resolution 3D tracing operations will require the underlying algorithms to be more tailored to the specific end-use. As a case in point, consider the advection of tracers for the purpose of computing the finite-time Lyapunov exponent (FTLE) field. FloLIB's generic passive tracer functions are currently used to compute and store the full set of tracers at each time-step. The tracer field is then later re-processed by separate FTLE-specific algorithms. As far as FTLE computation is concerned, this decoupling of tracer advection from FTLE computation consumes a tremendous amount of both computation time (much of which is wasted writing and then re-reading the tracer field) and storage space. If the user's principal interest is the FTLE field, however, a small cluster of tracers could be followed in time, the desired FTLE field computed, and the tracer coordinates themselves discarded. This application-specific tailoring of passive tracing algorithms could significantly improve performance[6].

At present, SPIVET enforces the use of mm units for internal world coordinates. This can lead to convoluted combinations of units when computing derived quantities such as density. SPIVET should permit the user to specify arbitrary spatial units.

[6]Tailoring of the code to end-use applications can be accomplished without necessitating the construction of a whole host of nearly identical, but slightly different tracing functions. The key is to re-architect the existing code into a more modular, object-oriented framework, with perhaps the producer-consumer design model used as guidance. Actually the current non-object-oriented configuration isn't far from this concept; the existing consumers simply don't intercept the produced data until it has been written to disk. Re-architecting the passive tracer code would also permit the code to be run in parallel (it is currently a serial implementation).

Although the majority of the most important functions have unit tests to help verify that SPIVET is operating correctly, not all parts of SPIVET have sufficient test cases. Implementing additional unit tests for under-tested or untested parts of the SPIVET framework would greatly improve the robustness of the code to downstream changes.

0.9 References

[1] Kitware, Inc. ParaView - Open Source Scientific Visualization. http://www.paraview.org.

[2] Kitware, Inc. VTK - The Visualization Toolkit. http://www.vtk.org.

[3] Lawrence Livermore National Laboratory. VisIt Visualization Tool. https://wci.llnl.gov/codes/visit.

[4] matplotlib. matplotlib: Python Plotting. http://matplotlib.sourceforge.net.

[5] MPI Forum. Message Passing Interface Forum. http://www.mpi-forum.org.

[6] NumPy. Scientific Computing Tools for Python - NumPy. http://numpy.scipy.org.

[7] Sandia National Laboratories. ExodusII. http://sourceforge.net/projects/exodusii.

[8] Scientific Computing Associates, Inc. NetWorkSpaces. http://www.lindaspaces.com/products/NWS_overview.html.

[9] SciPy. SciPy. http://numpy.scipy.org.

[10] Secret Labs AB. Python Imaging Library. http://www.pythonware.com/products/pil.

[11] Unidata. NetCDF (network Common Data Form). http://www.unidata.ucar.edu/software/netcdf.