Python for CFD: A case study

Prabhu Ramachandran *

Computers and Fluids Laboratory,
Department of Aerospace Engineering,
IIT-Madras, Chennai, INDIA 600 036

Abstract

In this paper we discuss the benefits obtained by the use of Python in our CFD computations. Our research involves the development and study of a high-resolution vortex method. We outline the various Python scripts that enabled us to do routine tasks, drive CPU intensive simulations from Python, schedule several hundreds of runs and interactively explore and analyze the data. We also show how SWIG, Pypar and SciPy's Weave enabled the rapid and efficient implementation of an interesting parallel program.

1 Introduction

Computational Fluid Dynamics (CFD) involves the computational simulation of fluid flows. We are interested in a class of schemes known as vortex methods applied to the flow of viscous, incompressible fluids in a two-dimensional domain. The vortex method is a grid-free method and employs a Lagrangian method of solution. Computational points are introduced and tracked as the simulation evolves. The method is challenging and involves a fair amount of computational effort. In [17] we implement a random vortex method and obtain high-resolution fluid flow simulations. In this paper we explore how Python enabled many of the computations presented there. It is of interest to note how Python made many things easier to manage. In isolation, many of the developed Python scripts are of no particular interest. However, with respect to the goals of our research problem, the combination of the various tools developed and used proved to be

*Graduate Student, prabhu@aero.iitm.ernet.in

of immense benefit. Almost all of the number-crunching code was implemented in C++. However, the runs were scheduled using Python. Much of the analysis, and management of the data was also done using Python. In this paper we seek to bring out the following.

- Python makes for an excellent general purpose scripting language suitable for scientific computations.
- Python often makes routine and boring tasks efficient and fun.
- Through the use of SWIG [1, 2] and SciPy's[9] weave, it is also possible to rapidly develop computationally efficient programs. This will be brought out with a specific example of a parallel application that we developed.

Therefore, Python serves as an extremely powerful tool in the arsenal of any researcher interested in improving their productivity.

In the following, we first introduce the problem of interest. A very brief introduction to vortex methods is provided. Subsequently, we show how Python proved useful. Finally, we show how we rapidly developed a non-trivial, efficient, parallel application using different Python modules.

2 Vortex methods

Computational Fluid Dynamics (CFD) involves the computational simulation of fluid flow problems. A common approach of simulating complex fluid flows is to employ the Navier-Stokes (NS) equations. These are non-linear, second order partial differential equations. Solving these equations for general fluid flows requires the use of numerical schemes. Traditional CFD requires the use of a computational grid to discretize the physical space. The NS equations are discretized on this grid and then solved using an appropriate numerical scheme. Generating the grid for complex geometries can involve considerable effort and sometimes introduces difficulties in the numerical simulation of the governing equations.

Vortex methods offer a grid-free approach for the computational simulation of fluid flows. In this work we are concerned with vortex methods for the simulation of the flow of incompressible, viscous fluid flows in two-dimensional domains.

2.1 Governing equations

Vorticity, $\vec{\omega}$, is defined as the curl of the velocity field, $\vec{\omega} = \text{curl } \vec{V}$. Upon taking the curl of the NS equations, it can be seen that under the assumption of incompressibility, the entire flow is governed in terms of the vorticity alone. For the flow

past a body B, in two-dimensions, the governing differential equations along with the boundary conditions are given by,

$$\frac{\partial \omega}{\partial t} + \vec{V} \cdot \text{ grad } \omega = \nu \nabla^2 \omega, \qquad \text{(Vorticity transport)} \tag{1a}$$

$$\omega = \text{curl } \vec{V} \cdot \hat{k}, \qquad \text{(Vorticity)} \tag{1b}$$

$$\text{div } \vec{V} = 0, \qquad \text{(Mass conservation)} \tag{1c}$$

$$\omega(\vec{r}, 0) = \omega_0(\vec{r}), \qquad \text{(Initial condition)} \tag{1d}$$

$$\vec{V}(\vec{r}, t) = 0 \qquad \text{as } \vec{r} \to \infty, \qquad \text{(Infinity BC)} \tag{1e}$$

$$\vec{V}(\vec{r}, t) \cdot \hat{e}_n = \vec{V}_B \cdot \hat{e}_n \qquad \text{on } B, \qquad \text{(No-penetration BC)} \tag{1f}$$

$$\vec{V}(\vec{r}, t) \cdot \hat{e}_s = \vec{V}_B \cdot \hat{e}_s \qquad \text{on } B, \qquad \text{(No-slip BC)} \tag{1g}$$

where $\hat{e}_n$ and $\hat{e}_s$ are the normal and tangential unit vectors on B and $\hat{k}$ is the unit vector out of the plane of the flow.

Given the vorticity field, $\omega(\vec{r}, t)$ the velocity field satisfying the boundary conditions in equation (1f) and (1g) is to be found. Using this velocity field, the governing equation can be solved.

In the context of vortex methods, during each time-step, Δt, equation (1a) is solved in two steps.

$$\frac{D\omega}{Dt} = 0, \qquad \text{(Advection)} \tag{2a}$$

$$\frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega. \qquad \text{(Diffusion)} \tag{2b}$$

Equation (2a) is the advection equation. The advection equation implies that vorticity remains constant along a particle path. Equation (2b) is the diffusion equation.

In a vortex method, the vorticity field is first discretized into individual particles of vorticity called *vortex blobs*,

$$\omega(\vec{x}) = \sum_{j=0}^{N} f_\delta(\vec{x} - \vec{x}_j)\omega_j h^2, \tag{3}$$

where f_δ is a core or cutoff function (an approximation to the Dirac distribution), δ is a parameter called the core or cutoff radius, h is the grid spacing used for the initial discretization[1], N is the number of particles and $\vec{x}_j$, ω_j are the position and vorticity respectively of the j'th vortex blob.

[1]Note that this is only an initial discretization of the vorticity field and is not a fixed grid.

Given a vorticity field, $\omega(\vec{x}, t)$, the velocity field, $\vec{V}_\omega$ can be obtained as,

$$\vec{V}_\omega(\vec{x}, t) = \sum_{j=0}^{N} K_\delta(\vec{x} - \vec{x}_j)\Gamma_j. \tag{4}$$

K_δ is called a desingularized velocity kernel and is given as,

$$K_\delta = K * f_\delta. \tag{5}$$

where $*$ denotes convolution and

$$K(x, y) = \frac{(-y, x)}{2\pi r^2}, \tag{6}$$

where $r^2 = x^2 + y^2$.

Thus, the velocity field corresponding to a given vorticity field can be obtained depending on the nature of the discretization of the vorticity. Given a compact vorticity distribution or one that decays rapidly, the velocity field, $\vec{V}_\omega$ will satisfy the infinity boundary condition (eq. (1e)). In general, it will not satisfy the no-penetration and no-slip boundary conditions (eqs. (1f) and (1g)). The no-penetration boundary condition is satisfied by adding a suitable potential velocity field to $\vec{V}_\omega$. The no-slip boundary condition is satisfied by introducing vorticity along the surface of the body in a suitable manner.

Thus, given the vorticity field and the velocity field, the particles discretizing the vorticity can be convected by solving an ODE.

The diffusion equation can be solved in one of many ways suitable to particle methods. In the present work we use the random vortex method [5, 6]. In this method diffusion is simulated by making the individual vortex blobs undergo independent random walks with the displacement obtained from a Gaussian distribution having zero mean and variance $2\nu\Delta t$. Since the method is stochastic, the solution is noisy and some amount of averaging (ensemble, space or time averaging) is necessary to obtain smooth solutions.

More theoretical details along with a large number of references on vortex methods are available in a book by Cottet and Koumoutsakos [7]. Puckett [15] provides several details on the random vortex method employed in our work.

2.2 Numerical details

Algorithmically the vortex method usually proceeds in the following manner.

1. The slip velocity on the bounding solid surfaces is computed at a set of control points on the boundary.

2. The existing vortex particles are advected using an appropriate ODE integration scheme (usually a second order Runge-Kutta scheme is used).

3. Vortex particles are added just above the surface of the body to offset the slip velocity computed in step 1.

4. The vortex particles are diffused using the random vortex method.

5. The process repeats from step 1.

Thus, the computational method is physically intuitive. However, there are several computational challenges involved in the vortex method. The most important of these involves the efficient computation of the velocity field using an adaptive fast multipole method [4]. More details on these algorithms along with other algorithms used in vortex methods are discussed in [17].

3 Python for vortex methods

In the previous section we briefly discussed the mathematical details of vortex methods and also illustrated the general approach of the method. Implementing a vortex method is fairly challenging due to the complexity of the algorithms. High-resolution computations are also computationally expensive. Thus all the computationally intensive code developed has been written in C++. The library developed is called *VEBTIFS* (Vortex Element Based Two-Dimensional Incompressible Flow Solver). While the code is fairly efficient, it is a lot more convenient to use the library from a scripting language like Python.

Unlike many other scripting languages (save perhaps Lush [3]), Python is well suited to numerical computation. The existence of tools for numerical analysis, along with a powerful interactive interpreter make Python ideal for scientific computing. Thus we chose to use Python as our scripting language.

3.1 Building the library

We build and run our code on a cluster of Pentium®IV machines running Debian GNU/Linux. Initially we used GNU/Make to manage the builds. There are several different configurations of the library. The library has some rudimentary graphics support, using the Ygl [8] graphics library, that can be optionally turned on. We also needed to build debug, release and profiled versions of the library. Finally, we needed to build both shared and static libraries. While this was possible with makefiles, it required quite a bit of effort to get working. At this point we switched to using SCons [10] for the build. This was primarily because SCons build scripts are Python scripts, SCons handled dependencies very well, supports

parallel builds from the ground up and also handled multiple variants of the builds in a straightforward manner. The initial build system used over 20 makefiles with around 1000 lines of code (without support for shared libraries). The code was also not easy to read. Switching to SCons reduced this to two files with just 200 lines of code (with support for shared libraries). The SCons build scripts were Python scripts and as such were much easier to understand than the original makefiles. They were also powerful enough to handle all of the complications of our builds.

SCons supports parallel builds much better than make. In combination with distcc [14], SCons reduced compile times almost linearly with an increase in the number of computers on the network. Using distcc itself required very little modifications to the scripts. Thus, switching to SCons proved to be a very significant benefit.

3.2 Parsing text data files

Very often one needs to parse a simple text data file. In our work we found that commented text files were extremely useful. A simple commented text file is of the following form,

```
key = value # comment
# or
key1, key2 = value1, value2 (garbage text/doc string)
# or just
value1, value2
```

The keys, comments and garbage text/documentation strings are irrelevant and are just used for convenience. The values are alone of significance. It is easy to see that this type of file is both useful and commonly found. Most often the user is aware of the file format, i.e. what is to be expected in the file. Therefore, we required a simple parser that would allow us to obtain, in sequence, a particular value (an integer, float or string or alpha-numeric value) from the file.

A typical use case would be as shown below

```
p = DataParser()
p.parse_file('test.dat')
v = p.get_float()
a = p.get_float()
n = p.get_int()
for i in range(2):
    x, y = p.get_float(), p.get_float()
```

For example, when parsing for a float, all strings and comments are ignored until the float is found. Thus the text file is easily understood by a user and parsed by

code. Using XML for this seemed like overkill. Besides, it is clear that a lot of common data files are structured this way. For example, a data file that is plotted with gnuplot would be parse-able with our code, and not with an XML reader.

The data parser interface is listed in listing 1 The code for this class can be ob-

Listing 1 The DataParser interface

```
class DataParser:
    def parse_file ( self , file ):
        pass
    def get_float ( self ):
        pass
    def get_int ( self ):
        pass
    def get_string ( self ):
        pass
    def get_alpha( self ):
        pass
    def get_alnum(self):
        pass
```

tained from http://www.ae.iitm.ac.in/ prabhu/software/code/python/dp.py. This module is not efficient but works well and is remarkably useful.

3.3 Conversion of data formats and versioning

As our code evolved, the input and output file formats kept changing. We had data files in either ASCII or XDR formats. In addition, the files could also be optionally zipped. The main program that would perform the simulation would always use the latest data format. However, it was important to be able to change the data format from older data files. Further, manual inspection of data stored in the XDR format required conversion to ASCII. The file format itself was involved. Writing a converter in C++ to handle the four possible file types and the various versions of the data was quite out of the question. Getting XDR I/O support in VEBTIFS (in C++) involved a few days of effort. A shell scripting language (like bash) could not be used either since the task was too complex. Python was an obvious choice. The fact that the Python standard library had support for XDR (using the `xdrlib` module) was a big plus. Getting a basic converter working took a few hours of work. With a little more work, the code supported both XDR and ASCII I/O, supported zipped files, allowed one to read and write different formats and had a useful command line interface. The ASCII reader used the

data file parser discussed in section 3.2. The code was fun to write and much of the debugging was done on the interpreter. The final code was around 800 lines long.

An interface defining an abstract reader and writer is used for the I/O. Any object that writes and reads data derives from an abstract base class called `Thing` which stores a list indicating the type of the data elements the object will contain. Another list stores the actual data. A comment string is also stored and this is used to write out comments in the ASCII text output. The `Thing` class also has read and write methods that read and write the data using the format list and the data list. For each class in VEBTIFS that involves I/O, a class derived from `Thing` is created and handles changing of the versions. The read/write methods are overridden if necessary. This is a duplication of effort since the basic data format is implemented in two places (in VEBTIFS and in the converter written in Python). However, VEBTIFS only handles the latest version of the data format and it is quite easy to write code in Python to handle the changes. Thus, Python made the file converter possible. This allowed us to change the file formats more freely. Additionally, the code gave us the ability to read and inspect the data on an interpreter session.

3.4 A job scheduler

As discussed in section 2.1, the method of random walks is used to simulate diffusion of the vorticity. The method is stochastic and requires some amount of ensemble averaging for accurate results. While this requires more computational effort, the computations are also trivially parallelizable. The trouble is that several runs need to be made. In [17], we made close to 1000 runs in the span of a few months. The run-time for each computation would vary from anywhere near half-an-hour to twenty four. Our lab had a cluster of nine computers. A few of the machines were desktops, some were servers and most of the others were compute-servers. Many of the machines were being used by others for other tasks. Given these constraints we had to schedule almost 1000 individual runs which would take a long time to execute. Manually running these jobs was inefficient and error prone.

We decided to write a job scheduler to handle the task. The task was quite simple but once again something not easily achievable by writing a program in either C/C++ or using a shell script. Once again, Python was up to the task. What was required of the program was the following.

- Given a maximum load-level, run a job (specified as the full path to an executable program/script).
- If a submitted job cannot be run immediately, queue it.

- Allow the user to add and remove jobs in the queue.
- Allow the user to change the load-level and also the polling interval.
- Save a log of all the jobs run to date.

Since all the machines were running Linux, no cross-platform support was necessary. This simplified the task. Writing the code for the scheduler took about one day's work. The script had a complete command line interface, worked as a daemon and supported all we required. The program required about 500 lines of code.

In order to compute the current load level on a machine the the output of the `uptime` command is parsed. Similarly the output of the `ps` command is parsed to find the status of the running process based on its PID. The job is spawned using `os.spawnl` and the job's PID is the return value of the command. A `JobD` class manages the queue, the jobs and the history. A simplified view of the interface is shown in listing 2. Adding a job simply adds an entry into the queue in a host specific data file (this eliminates problems with NFS home directories). Every few minutes the daemon polls the queue and processes it. If the load level is acceptable it runs the first job in the queue. For each running job it checks (using the `ps` command) to see if the job is still running. If the job has terminated it makes an entry in the history (stored in the same file as the queue) along with the approximate start and stop time of the job.

The code was fun to write. Using this job daemon eased the task of making hundreds of runs efficiently.

In order to compute the ensemble average, several runs were necessary. This was done by using a different random number seed for each of the runs. This process was also automated using Python scripts. An input file determining the various parameters (including the random number generator's seed) is parsed. This information along with the number of different runs to be made was used to generate different input files for the runs along with the modified seed values. These files were placed in different directories. Once the files were generated, another file specifying a mapping between the directory and the machine where the files were to be transported and run was parsed. The data was then copied to all the relevant machines using the `commands` module from the standard library. The jobs were then scheduled on each of the target machines. Once the runs were all completed, the same script would be responsible to get back the data from the different machines. The launching of the runs and collection of the data were all done on interactive Python sessions using the enhanced interactive Python shell, IPython [12, 13]. Many of these features are part of freely available tools used to manage clusters. However, in our case there was a paucity of time to install and get these tools installed and running. A few days of Python coding achieved all we

Listing 2 The Job Scheduler interface

```
class JobD:
    def isServing(self):
        """Check if server is alive."""

    def run(self):
        """Process the queue and running jobs."""

    def start(self):
        """Start the server in daemon mode. Ideally the function
        never returns. Only one server per machine is allowed. If a
        server is already running this call returns."""

    def kill(self):
        """Kill the server."""

    def restart(self):
        """Restart the server after killing the running server."""

    def add(self, job):
        """Add a new job to the queue. The submitted job must be an
        executable file with full path or path implicit to the servers
        current working directory. Full path is preferred."""

    def remove(self, job):
        """Remove an existing job from queue."""

    def status(self, verbose=0):
        """Print status of server."""

    def history(self):
        """Print history of jobs run on server."""
```

wanted. Our scripts were simple and could also be used on other Linux clusters where we were unable to install new software easily.

Therefore, by using these Python scripts, it was possible to run several hundreds of simulations on different machines and also manage the data generated by these runs.

3.5 Driving C++ code using SWIG

It is extremely convenient to be able to drive simulations from an interactive and interpreted environment. The C++ code implementing the vortex method consisted of around 80 classes in over 35000 lines of code. These were wrapped to Python using SWIG[1, 2]. This required writing about 500 lines worth of SWIG interface files. SWIG handles bulk of the wrapper generation. Writing the interface files and getting everything working satisfactorily took around a week's work[2]. SCons was used to build the wrappers.

Once the SWIG wrappers were generated it became possible to write Python scripts to make runs, graphically inspect the data and do a host of other things. One such helper script allows us to view the data interactively, generate vorticity contours, streamlines and compute the velocity on points specified. This is used to compare our results with those of other researchers.

3.6 Data analysis

In order to analyze the data from our simulations we used the Python interpreter. We used IPython [12, 13] for this purpose.

Common tasks are abstracted into useful functions or classes that can be reused several times. Data from different simulations can be ensembled using a Python script. The script reads the output from the different runs and ensembles them and stores these results in a separate directory. The mean and standard deviation for various quantities are also easily computed using the data. The data parser is used to parse text files and Numeric is used for the computations.

Some of the data requires special processing. Due to the stochastic nature of the simulation some of the output can be noisy, these are processed using either simple averaging techniques (using the `Numeric` module) or more sophisticated schemes like fitting a piecewise polynomial to the data. For these purposes the various modules provided by SciPy [9] are used. Specifically, the `optimize` and `interpolate` modules are used.

Two-dimensional line plots are generated using Grace [18] and the gracePlot module (available here). For more sophisticated 2D and 3D visualization, MayaVi [16] is used.

[2]This includes reading the SWIG documentation and experimenting with SWIG.

4 Building a parallel application

The previous section demonstrated how Python proved useful in various ways. We use Python to build our code, schedule hundreds of runs, manage data and analyze data interactively.

In this section we demonstrate how we used Python to build an efficient parallel program. Given a cluster of processors (Pentium®IV machines running at 1.7GHz), the program performs computations for several time steps on each processor. Every n_{sync} time steps the data from each of the processors is assembled on one of the processors called the master processor. The data is then processed on the master and sent back to the slave processors. The processors resume their work from this point. Bulk of the time is spent in performing the computations on each processor. However, large quantities of data are transferred. The entire simulation could take anywhere between 4-24 hours.

For one particular run, the average time taken per iteration on an individual processor was 1 minute and the communication occurred every 10 iterations. On the average, each processor communicated about 3MB of data to the master processor and the master in turn communicated the same amount back to all the processors.

Since the program was embarrassingly parallel, we decided to try and implement it in Python. The salient features of the program we implemented are the following.

1. The computations on individual processors are run using the SWIG wrapped version of the VEBTIFS library.

2. The communication of the data is done using Pypar [11]. Numeric arrays of the data are communicated.

3. The data for communication is obtained from from various vortex particles (sometimes running into a quarter of a million particles on each processor), stored in Numeric arrays and transferred using Pypar. This data is then converted back into particle data on the master, processed at the C++ level, transformed back into a Numeric array and sent back to the slave processors.

Step 3 in the above is the most time-consuming, if the problem is to be solved in Python. However, at this point, we only had an idea of what we needed to do and no concrete information on how best to do things. Consequently, prototyping the idea in Python seemed like a good idea. Performance could be addressed later. The basic program was written in half a day. Debugging and getting everything working correctly took the rest of the day. The outline of the main class developed for the program is given in listing 3.

Listing 3 The parallel program interface

```
class Solver:
    def initialize(self, inp):
        """ Initializes solver from an input file."""
        ...
    def iterate(self):
        """ Performs self.n_sync iterations of the solver."""
        ...
    def readData(self, id):
        """Read data from the processor having ID, id."""
        ...
    def sendData(self, id):
        """Send data to the processor having ID, id."""
        ...
    def processData(self):
        ...
    def master(self):
        while self.fs.getCurrentTime() < self.stop_time:
            self.iterate()
            for i in range(1, self.n_proc):
                self.readData(i)
            self.processData()
            for i in range(1, self.n_proc):
                self.sendData(i)

    def slave(self):
        while self.fs.getCurrentTime() < self.stop_time:
            self.iterate()
            self.sendData(0) # send data to master
            self.readData(0) # get back processed data from master

    def run(self):
        if self.id == 0:
            self.master()
        else:
            self.slave()
```

Pypar initializes the MPI library on import. In this case LAM version 6.5.6 was used. The rest of the code is mostly straightforward. Each slave processor manages hundreds of thousands of particles (vortex blobs actually). The master processor also has its own set. After completing the computations, the slave processor needs to send back the master processor all of its particles. The master reads these from each processor and then processes them. It then sends the data back to each processor. The blobs on each processor are stored inside a `BlobManager` object. This is part of the VEBTIFS library and is wrapped to Python using SWIG. Pypar lets one receive and send Numeric arrays. Thus, the data inside the `BlobManager` object needs to be converted into a Numeric array. Similarly when reading data from the master processor, the data in the form of Numeric arrays needs to be converted back into `Blob` objects stored in a `BlobManager`. Doing this in Python is very slow. In listing 4, one example of a typical function used to convert data from numeric arrays to blobs is shown. The function `data2Blob_slow` is the way the function was first implemented. Needless to say, the implementation was very slow. However, it worked and was useful to get the code working correctly. In order to accelerate these function we resorted to using SciPy's `weave` module. Python wrappers for the C++ library, VEBTIFS, are made using the latest development of SWIG (version 1.3.22) called SWIG2. Unfortunately, `weave` did not support SWIG2 wrapped objects. In a day's (night actually) effort we were able to add SWIG2 support to `weave` [3]. The equivalent code with `weave` is shown in the function `data2Blob` in listing 4. As seen the code is easy to read and quite similar to the Python version. Using `weave` resulted in a 300-400 fold speedup!

The final version of the code is 570 lines long. It has a useful command line interface (7 different options), supports some primitive graphics [4] and is efficient. The overall performance of the code was excellent. Developing the code took about 3 days of work from concept to a working version. This includes the time spent on adding SWIG2 support for `weave`. The resulting code helped us produce simulations of unprecedented resolution [17] for the flow past an impulsively started circular cylinder using the random vortex method.

5 Conclusions

We have demonstrated here that Python is extremely well suited for scientific computations for several reasons.

- It is easy to learn and very versatile. The language is well designed and lets one focus on the problem at hand.

[3] This is available in `weave` from CVS.

[4] The graphical output is part of VEBTIFS. The important thing is that the output of all the processors can be graphically inspected.

Listing 4 Inter-conversion of data

```
def data2Blob_slow(blb_data, bm):
    """ Given numeric arrays containing blob data and a BlobManager,
    populate the BlobManager with blobs."""

    z, gam, core = blb_data['z'], blb_data['strength'], \
                   blb_data['core']
    bf = vebtifs.BlobFactory()
    for i in xrange(len(z)):
        bm.addElement(bf.create(z[i], gam[i], core[i]))

def data2Blob(blb_data, bm):
    z, gam, core = blb_data['z'], blb_data['strength'], \
                   blb_data['core']
    nb = len(z)
    bf = vebtifs.BlobFactory()
    code = """
    for (long i=0; i<nb; ++i) {
        bm->addElement(bf->create(z[i], gam[i], core[i]));
    }
    """
    weave.inline(code, ['bm', 'bf', 'z', 'gam', 'core', 'nb'],
                 headers=['"blob.H"'])
```

- The standard library along with other open source modules make Python extremely powerful.
- It is relatively easy to interface to C/C++ or Fortran libraries.
- Thanks to its excellent interactive interpreter (via IPython) and modules like Numeric/numarray, SciPy, gracePlot etc., Python is well suited to perform interactive numerical analysis and exploration of data.
- With packages like `weave` it is possible to build highly efficient Python modules relatively easily.

As demonstrated, non-trivial parallel applications can also be developed very rapidly using Python. Thus, it is clear that Python as a programming language is an excellent tool in the armory of any scientist or engineer. It would be of immense benefit to anyone pursuing serious computational work to be familiar with a high-performance language like C/C++ or Fortran *and* Python. As demonstrated in this paper, with the knowledge of both C/C++ and Python the possibilities are immense.

6 Acknowledgements

I gratefully acknowledge the encouragement and guidance of my advisors, Professors S. C. Rajan and M. Ramakrishna of the Department of Aerospace Engineering, IIT-Madras.

References

[1] David Beazley et al. SWIG: Simplified wrapper and interface generator, 1995–. URL `http://www.swig.org`.

[2] David M. Beazley. SWIG : An easy to use tool for integrating scripting languages with C and C++. Presented at the 4th Annual Tcl/Tk Workshop, Monterey, CA, July 1996.

[3] Leon Bottou, Yann LeCun, et al. The lush programming language, 2002–. URL `http://lush.sf.net`.

[4] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Stat. Comput.*, 9(4):669–686, 1988.

[5] A. J. Chorin. Numerical study of slightly viscous flow. *Journal of Fluid Mechanics*, 57(4):785–796, 1973.

[6] A. J. Chorin. Vortex sheet approximation of boundary layers. *Journal of Computational Physics*, 27(3):428–442, 1978.

[7] G.-H. Cottet and P. Koumoutsakos. *Vortex methods: theory and practice*. Cambridge University Press, March 2000.

[8] Fred Hucht. The Ygl graphics library, 1993–. URL `http://www.thp.uni-duisburg.de/Ygl/`.

[9] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[10] Steven Knight et al. SCons: A software construction tool, 2001–. URL `http://www.scons.org`.

[11] Ole M. Nielsen. Pypar: parallel programming in the spirit of Python, 2001–. URL `http://datamining.anu.edu.au/~ole/pypar/`.

[12] Fernando Pérez. IPython: an enhanced interactive Python. In *SciPy'03 – Python for Scientific Computing Workshop*, CalTech, Pasadena, CA, USA, September 11th.–12th. 2003.

[13] Fernando Pérez et al. IPython: an enhanced interactive Python shell, 2001–. URL `http://ipython.scipy.org`.

[14] Martin Pool et al. Distcc: a fast, free distributed c/c++ compiler, 2002–. URL `http://distcc.samba.org`.

[15] E. G. Puckett. Vortex methods: An introduction and survey of selected research topics. In R. A. Nicolaides and M. D. Gunzburger, editors, *Incompressible Computational Fluid Dynamics — Trends and Advances*, page 335. Cambridge University Press, 1991.

[16] Prabhu Ramachandran. MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium*. Aeronautical Society of India, August 2001. Software available at: http://mayavi.sf.net.

[17] Prabhu Ramachandran. *Development and study of a high-resolution two-dimensional random vortex method*. PhD thesis, Department of Aerospace Engineering, IIT-Madras, 2004.

[18] Paul J Turner, Evgeny Stambulchik, et al. Grace: a 2d plotting tool, 1998–. URL `http://plasma-gate.weizmann.ac.il/Grace/`.