Project report

INF5660, V2004, Kim Andre Sand (kimsa@ifi.uio.no)

Table of Contents

Part 1 - Finding a project

I started out with lots of loose ideas and interests in several areas and had quite a hard time trying to find some common factor to bind all those ideas together into one good project. Much time was actually spent doing research in several directions at once, learning, testing and looking for ideas. Eventually, time started getting scarce and I had to make a choice. As I had already spent so much time on research I decided to make my project both a brief introduction to the various tools and a short comparison of them regarding convenience and efficiency. I needed some sort of tool for comparing the running times of different implementations of a function, and decided to create my own; TimerSuite. This tool ended up becoming a large part of the project, and is documented properly in the source with doc-strings. My project thus consists of three parts: An introductory report, a tool for performing tests, and a few tests implemented using that tool. The results of those tests, and my observations, are also in this report.

Please note that I have not supplied any HTML documentation, as I saw little use in duplicating this report in another format. I couldn't quite figure out which parts to extract either, as the parts don't make much sense by themselves.

Part 2 - Tools and environment

Windows as a base for scientific computing:

Windows XP is my home operating system, and I'm more comfortable working under it than under Unix and its descendants. I therefore decided to do my project under Windows. Since both this course (INF5660) and its predecessor (INF3330) is focused on Unix as an environment, I also thought it might be useful to document the process of setting up the necessary tools for Windows. My goal was to get everything up and running using only free software (Windows itself being the exception, of course). Further, I tried to find packages with precompiled binaries and automatic installers, as Windows users generally dislike spending time on compiling and configuring their software.

Required software components:

First of all one needs Python. Though Python can be installed by itself, ActivePython [1] from ActiveState is a better alternative for Windows users. It comes bundled with a few essential libraries, Windows bindings, and full documentation in the handy, searchable Windows Compiled HTML format. It's also bundled with the free PythonWin IDE, which is a nice, simple IDE with basic debugging capabilities. More than good enough for my purpose. ;)

Finding a good compiler:

When building extensions one needs compilers for the extensions' target languages. For my project I needed C/C++ and Fortran. Unfortunately there are not many worthy, free compilers to choose from under Windows. C++ developers tend to use Microsoft Visual C++ 6 or .NET, both rather costly and extensive environments. The aging VC6 is especially popular with hobby programmers and smaller businesses. The default Python distribution binaries are compiled under it, making it a good candidate for building extensions. Unfortunately its both highly outdated, commercial and has very limited support for C++ STL and templates.

A free Microsoft compiler?:

Microsoft has recently released their free Visual C++ Toolkit 2003, which is basically just the compiler and libraries from their latest Visual C++ .NET incarnation. I decided to try it out, as it would hopefully make building extensions a breeze, being VC6 link compatible. Unfortunately, I also had to download and install the massive Microsoft Platform SDK to get anywhere. Not exactly ideal when developing platform independent software! Many hours later, after struggling with cryptic error messages and unsuccessful builds, I finally learned that the free toolkit wasn't complete after all. It was missing essential multithreaded libraries (needed by Boost especially) and a make tool (needed by most larger distributions). These missing components were, of course, not available for free. Pity.

Unix on Windows:

Abandoning bloated commercially developed software I started looking at ports of Unix tools instead. There are two major versions of GCC for Windows: Cygwin and MinGW. Cygwin tries to simulate a Unix environment running under Windows, thereby also requiring that compiled binaries are run under the same environment. Not good. MinGW, on the other hand, provides native Windows versions of the GCC tools and even compiles binaries that are link compatible with VC6! Very good! :)

When trying to download MinGW there are lots of different distributions available. The one to get is the one called just "MinGW". It contains the most essential tools bundled together. When newer versions of certain tools are available, just download them separately and unpack them to the installation directory. It's also a good idea to download MSYS which provides a minimal Unix environment useful for installing software based on Unix shell scripts. Compared to the earlier Microsoft attempt, the install is small, quick and painless. Not only does GCC fully support C++ STL and templates, but the suite also solves my problem of finding a Fortran compiler.

MinGW is not Visual C++:

When trying to build extension modules with MinGW, tools like SWIG and F2PY soon start complaining that Python was compiled with VC++ and extensions therefore need to be built with the same compiler. Searching the official Python documentation for help, it says one needs to compile a new library for Python using MinGW. This is not correct! All one needs is to let the tools know that one is using MinGW instead of VC++. (I'll specify how when I come to the specific tools below). This is because the MinGW compilers link with the standard Microsoft provided libraries, creating VC6 compatible binaries.

Going Psyco:

Psyco [3] is a JIT (Just-In-Time) compiler for Python, in the spirit of Java. It automatically compiles Python code on-the-fly and is very easy to set up and use, even for a complete novice. The results are pretty amazing, as I will show later. Unfortunately, it's only available for x86 architecture machines. If one is able to use it, however, there's hardly any quicker or easier way to optimize a program!

Interfacing with Fortran:

The best option for working with existing Fortran libraries is hands down F2PY [4]. It's a pleasure to work with. It's available as a click-install binary for Windows, and also installs the required SciPy distutils if not found. When running F2PY with MinGW, an additional argument is needed: *"--compiler=mingw32"*. Don't forget TWO dashes *"--"* as F2PY will happily ignore the argument otherwise. An easy detail to overlook when coming from the Windows world. ;)

Scientific computing with SciPy:

SciPy seems to be one of the most promising up-and-coming tools for scientific computing. It builds on Numeric and wraps a large number of Fortran scientific libraries using F2PY. Of particular interest to me is the 'weave' package which allows one to write snippets of code in C++, inlined in the Python code. Far easier than SWIG when going from Python to C++ and not the other way around. Being based on Numeric, it also wraps Numeric arrays automatically back and forth. When using weave.inline with MinGW, an additional argument is needed: *"compiler='gcc'"*. See the code examples for details.
The weave package also provides weave.blitz, which converts Python vectorized expressions to the Blitz++ library. Other tools are weave.ext_tool for building extension modules, and a future tool called weave.accelerate. SciPy [5] and required libraries (including Blitz++) can be downloaded as a click-install binary too. Sadly, the Weave documentation is quite old and lacking, and skimming through Fernando's documents [5b] first can save lots of confusion and problems later.

Boost and Boost.Python:

Boost [6] is a promising tool when working with C++. It is part of a massive library, has commercial backing and promises to integrate Python with C++ in new, powerful ways. Its interface is also fully based on C++ template syntax, and does not require one to learn an additional interface semi-language (like SWIG). Built-in support for Numeric arrays is another huge bonus. Unfortunately, Boost quite a pain to work with in its current state. The custom build environment is tiresome to use at best, and the documentation is rough and incomplete. I'm fairly certain it will surpass SWIG, CXX and friends as a C++ wrapper when it has matured though. From my experience, SWIG and SCXX are currently better options.

Working with documents:

Finally, it's useful having some kind of office suite installed for working with documents, presentations and the like. 'OpenOffice.org' [7] is an excellent, fast and free alternative to Microsoft Office. It supports every major format and can export documents to PDF. Besides, Python can be used as the suite's scripting language, thereby granting full access to the suite's API for extension, automation and data exchange. ;) This report was written in OpenOffice.org.

List of websites for the products mentioned:

[1] ActivePython:
http://www.activestate.com/Products/ActivePython/

[2] MinGW:
http://www.mingw.org

[3] Psyco:
http://psyco.sourceforge.net/

[4] F2PY:
http://cens.ioc.ee/projects/f2py2e/

[5] SciPy:
http://www.scipy.org/

[5b] Fernando Perez on weave:
http://windom.colorado.edu/~fperez/python/python-c/

[6] Boost.Python:
http://www.boost.org/

[7] 'OpenOffice.org':
http://www.openoffice.org

Part 3 – Software and requirements

In addition to this report, the project includes a distutils distribution. As I've only been working under Windows, I've made a click-install binary distribution which is the best option for Windows users. The file is called: “Timing-0.1a.win32.exe”. I've also packaged the files in a .zip archive, called “Timing-0.1a.zip” in case Windows isn't available when reviewing the project. The files should install using the included 'setup.py' script, but I haven't tested this on Unix and friends.

The archive contains the following files:

setup.py	The script for installing the module

In the Timing folder:

Timing.py	My tool for timing algorithms
TimingTests.py	Unit tests for the timing tool

In the Examples folder:

Integral.py	The integration exercise from the course
Integral_F77.f	The supporting Fortran code
Integral_F77.pyd	A compiled DLL for Windows
Smooth.py	The timeseries smoothing exercise from the course
Smooth_F77.f	The supporting Fortran code
Smooth_F77.pyd	A compiled DLL for Windows
Smooth3D.py	The 3D smoothing exercise from the course
Grid2D.py	The 2D gridloop example from the course

To run the examples a C/C++ compiler and SciPy (with weave) is required. The “compiler” attribute in the top of the examples must be modified if a non-gcc compiler is to be used. To recompile the Fortran modules, F2PY and a Fortran compiler is needed. As I've been working on Windows, I've included the compiled Fortran binaries in the Windows DLL format for python (.pyd).

Please note that I have not included any regression tests as I couldn't figure out how to make them fit the TimerSuite. The results produced by the suite are totally dependent on the system running the script, and the logic behind producing those results are more easily verified by unit tests.

Part 4 - Results

First let's try running Integral.py, the integration exercise from the course, and compare running times for an ordinary loop containing mathematical expressions. The session is set up with 4 runs, with 'n' (the number of samples used to estimate the integral) from 10^4 to 10^8, step 10^x.

This is the output from TimerSuite:

```
Run #1:
Plain loop took: 0.065440161 CPU seconds
Psyco loop took: 0.015195507 CPU seconds
Weave loop took: 0.008417550 CPU seconds
Vectorized took: 0.013844776 CPU seconds
F77 loop   took: 0.000047771 CPU seconds

Run #2:
Plain loop took: 0.659567347 CPU seconds
Psyco loop took: 0.168675907 CPU seconds
Weave loop took: 0.083955363 CPU seconds
Vectorized took: 0.196982298 CPU seconds
F77 loop   took: 0.000278527 CPU seconds

Run #3:
Plain loop took: 6.677657635 CPU seconds
Psyco loop took: 1.513332560 CPU seconds
Weave loop took: 0.844072082 CPU seconds
Vectorized took: 1.927898099 CPU seconds
F77 loop   took: 0.002683581 CPU seconds

Run #4:
Plain loop took: 66.899459848 CPU seconds
Psyco loop took: 14.801240102 CPU seconds
Weave loop took: 8.318599431 CPU seconds
Vectorized took: 19.513419164 CPU seconds
F77 loop   took: 0.025765845 CPU seconds
```

As 'n' increases with a factor of 10, so does the running times roughly. The most interesting observation here is how much faster Fortran is than the other versions. Also note that Psyco is perfectly capable of keeping up with a vectorized Numeric expression in this case.

Let's use the last run to see how many times slower the other versions are:

```
F77 loop   :    1
Weave loop :  322
Psyco loop :  574
Vectorized :  757
Plain loop : 2596
```

The peak memory usage during the last run was about 9 Mb for the loop versions, but 321 Mb (!) for the vectorized version, showing the first bleak signs of the penalty inherent in vectorization. Let's see in the next example what happens once we start working more heavily with arrays.

Next up is Smooth.py, the smoothing of time-series exercise from the course. Here, two reads and one write is performed on a Numeric array each iteration, with 'n' (the nr of elements in the array) from 10^4 to 10^8, step 10^x.

```
Run #1:
Plain loop took: 0.013955126 CPU seconds
Psyco loop took: 0.004159188 CPU seconds
Blitz loop took: 0.000311213 CPU seconds
Weave loop took: 0.000300876 CPU seconds
Vectorized took: 0.000805410 CPU seconds
Blitz vec  took: 0.001629816 CPU seconds
F77 loop   took: 0.000380495 CPU seconds

Run #2:
Plain loop took: 0.137291751 CPU seconds
Psyco loop took: 0.046113682 CPU seconds
Blitz loop took: 0.005388394 CPU seconds
Weave loop took: 0.005694579 CPU seconds
Vectorized took: 0.017619558 CPU seconds
Blitz vec  took: 0.006641906 CPU seconds
F77 loop   took: 0.009894274 CPU seconds

Run #3:
Plain loop took: 1.413657322 CPU seconds
Psyco loop took: 0.460999932 CPU seconds
Blitz loop took: 0.043473123 CPU seconds
Weave loop took: 0.042326888 CPU seconds
Vectorized took: 0.170448758 CPU seconds
Blitz vec  took: 0.044447269 CPU seconds
F77 loop   took: 0.105451391 CPU seconds

Run #4:
Plain loop took: 14.139421910 CPU seconds
Psyco loop took: 4.734886696 CPU seconds
Blitz loop took: 0.437985833 CPU seconds
Weave loop took: 0.469044250 CPU seconds
Vectorized took: 1.776361800 CPU seconds
Blitz vec  took: 0.461332935 CPU seconds
F77 loop   took: 1.134036868 CPU seconds
```

When working extensively with arrays, Fortran loses its crown to weave. Here the plain weave.inline version is about equal to the blitz version, as the Blitz++ library probably doesn't find much to optimize. Also note that the weave loop versions are just as fast as the vectorized versions.

Let's use the last run again to compare performances:

```
Blitz loop :  1.00
Blitz vec  :  1.05
Weave loop :  1.07
F77 loop   :  2.59
Vectorized :  4.06
Psyco loop : 10.80
Plain loop : 32.28
```

This time the two allocated arrays (source and destination) had to be equally large for every implementation, and so the memory usage was about equal for all. It peaked at about 415 Mb.
As a side-note, I also tried writing each computation back to the same array, thus working with a "live" array that changed with each iteration. I found out that the Numeric vectorized version now produced the wrong results, due to working with temporary arrays and not reading fresh data each iteration. The weave versions (loop and vectorized) still produced the correct results and ran a bit faster too.

Let's move to Smooth3D.py and see what happens when we start working in three dimensions instead of just one. Here the memory requirements increase by a factor of x^3, and so we have to settle with small array sizes of 'n'. I chose 10, 100 and 300.

```
Run #1:
Plain loop took: 0.350594559 CPU seconds
Psyco loop took: 0.223513603 CPU seconds
Blitz loop took: 0.001887391 CPU seconds
Weave loop took: 0.000594489 CPU seconds
Vectorized took: 0.010651633 CPU seconds
Blitz vec  took: 0.004549461 CPU seconds

Run #2:
Plain loop took: 14.041083993 CPU seconds
Psyco loop took: 9.616482542 CPU seconds
Blitz loop took: 0.231582226 CPU seconds
Weave loop took: 0.038525287 CPU seconds
Vectorized took: 0.437093821 CPU seconds
Blitz vec  took: 0.086128544 CPU seconds

Run #3:
Plain loop took: 378.387533332 CPU seconds
Psyco loop took: 265.641324526 CPU seconds
Blitz loop took: 35.346939066 CPU seconds
Weave loop took: 3.245423930 CPU seconds
Vectorized took: 371.581749077 CPU seconds
Blitz vec  took: 7.527887762 CPU seconds
```

Here things really start to get interesting. The weave loop version outperforms everything else by a large margin. In the last run it really flies! It's actually much faster than the blitz loop, especially with large array sizes. The blitz vectorized version has a much easier time, but it's clear that the Blitz++ optimizations have little to offer this algorithm. Note also that the Numeric vectorized version in the last run is just as slow as the plain Python loop! These results must be taken with a healthy dose of scepticism though. Nearly all versions peak at 650 Mb, with the Numeric vectorized one at a whopping 850 Mb! As my computer only has 512 Mb of RAM, these steep memory requirements lead to a lot of pagefile swapping and harddisk thrashing. Vectorized is now practically useless. Given my environment the weave loop is the clear winner.

Let's use the last run again to compare performances:

```
Weave loop :   1
Blitz vec  :   2.3
Blitz loop :  11
Psyco loop :  82
Vectorized : 115
Plain loop : 117
```

The final example is Grid2D.py, or the infamous gridloop from the lecture slides. Again we have to settle for small array sizes of 'n'. I chose 600, 2000 and 6000.

```
Run #1:
Plain loop took: 3.403155150 CPU seconds
Psyco loop took: 1.885668151 CPU seconds
Weave loop took: 0.066904593 CPU seconds
Blitz loop took: 0.078993839 CPU seconds
Blitzloop2 took: 0.076480391 CPU seconds
Vectorized took: 0.126815838 CPU seconds

Run #2:
Plain loop took: 37.945229072 CPU seconds
Psyco loop took: 20.928570810 CPU seconds
Weave loop took: 0.789173764 CPU seconds
Blitz loop took: 0.899003924 CPU seconds
Blitzloop2 took: 0.877810727 CPU seconds
Vectorized took: 1.431783648 CPU seconds

Run #3:
Plain loop took: 340.326438975 CPU seconds
Psyco loop took: 189.093675593 CPU seconds
Weave loop took: 6.968770612 CPU seconds
Blitz loop took: 8.132215128 CPU seconds
Blitzloop2 took: 7.959644871 CPU seconds
Vectorized took: 112.687375910 CPU seconds
```

In this example I have provided two different ways of supplying a function to the blitz loop, hence the two versions. See the example code for details. All versions peak at about 290 Mb, except the Numeric vectorized one peaking at about 580 Mb. Again, when the arrays get large, the vectorized version is pretty useless due to the high memory requirements needed for vectorization. Note that it goes from being about 27 times faster in the first two runs to just about 3 times faster in the last. The weave loop is again the winner by a small margin, showing that the Blitz++ library has not been of any help in my examples. Not through weave at least. Hand-coding an extension module could show a different story. From what I've heard, at least, Blitz++ is amazing when applied to complex, numerical algorithms. Unfortunately, complex math is not exactly my domain.

Let's use the last run again to compare performances:

```
Weave loop :  1
Blitzloop2 :  1.14
Blitz loop :  1.16
Psyco loop : 27
Vectorized : 16
Plain loop : 49
```

Part 5 - Observations

Psyco is by far the quickest and easiest way to speed up your program. No code changes are necessary, just bind slow functions to Psyco and watch them fly. This is truly free performance! Psyco can even be set to optimize the whole program, but this needs quite a bit of memory and probably won't give much of a boost compared to identifying and applying it to just the bottlenecks. Some programs have even been known to run slower when Psyco has been applied carelessly, probably because compiling certain parts takes longer than just plain running them. Luckily, Psyco comes with a built-in profiler that actually tests which parts of your program will benefit from compilation. When a speed boost is needed, try Psyco first!

Weave is very nice for prototyping C/C++. It's a lot easier than SWIG when all one needs is to test a few snippets of code in C/C++. Weave also makes it trivial to work with Numeric arrays in C/C++ (unlike SWIG). The underlying array formats of Numeric and Blitz++ are surprisingly compatible, making it quite possible writing powerful, manual extensions if needed! The weave-generated code is a nice starting point. Debugging weave can be quite tough though, being based on several layers of libraries, and error messages are often a mess of SciPy, Weave and Blitz++. Weave can be used both for converting algorithms and loops to C++ (using 'weave.inline') and for converting vectorized Numeric expressions to Blitz++ (using 'weave.blitz'). In both cases the speed boost is massive, far outperforming their Python/Psyco and Numeric equivalents. Thanks to weave, the speed can rival, and even surpass, that of hand-written Fortran modules when it comes to working with arrays! Most impressive! Fortran still seems to have the upper-hand with intensive computations not heavily based on array manipulation though. Still, weave is definitely my tool of choice for smart optimizations of bottlenecks!

Weave works by compiling code and storing it in a repository in a temporary location on the host machine. Each time it needs to run a piece of code this repository is first checked in case the code is already compiled. When lots of testing is done it's a good idea to manually clean out this repository once in a while, freeing up space and reducing look up time. I also learned the hard way that when experimenting heavily on the same algorithms, moving bits of code around, changing minor details and trying out different parts of weave, it's also wise to delete the repository once in a while. Weave sometimes has the tendency to get confused and use an old, incorrect version of the compiled code. The result is strange, or just plain wrong, behavior. If this happens, clean the repository and try again.

When using 'weave.inline' and compiling with blitz types (see code for examples) it's very important to remember to access arrays by parentheses, not brackets; i.e. 'a(i)' not 'a[i]'. Only the former version will be translated to a blitz array, while the latter version becomes a "regular" (SCXX-based) array. Weave will happily accept both forms, thereby making the mistake difficult to detect, but the performance difference can be quite significant, depending on the complexity of the problem. Weave will even accept mixing the two forms, but when I accidentally tried this the performance hit was huge! Again, weave won't complain if you do this, so be consistent!

Numeric vectorized arrays are practically useless on very large values because of extreme memory usage. Not only because of the space needed for vectorization, but also because of all the temporary arrays created in between computations. Numarray was partly created to combat this exact problem, but unfortunately isn't widely supported yet. SciPy (and thus weave) converts Numarray arrays to Numeric internally and works with them instead. With weave one doesn't necessarily need to vectorize though, as writing plain loops instead is usually easier and seems to be equally fast, if not faster. It's nice to know one has both options though.

Part 6 - Afterthoughts

Though I've learned a lot during this course and its related project, I've had a hard time putting all the pieces together. Coming from another background than physics and mathematics, I envision using the knowledge gained from this course in quite different areas than what is probably intended from studying the teaching material. This knowledge will undoubtedly prove useful later, but didn't exactly help when trying to create a project relevant for the course. My lack of understanding and experience regarding scientific numerical computing became sorely apparent when working on this project. I found out several times that I did not have the necessary background for completing an idea, and had to throw away the work and start anew. A few libraries also turned out to be quite different from what I had expected, ending in blind alleys or requiring too much work to be able to realize my ideas. Because of these problems I have several unfinished pieces lying around that have no place in the final project, and the end result doesn't fairly reflect the work that went into it.

Due to fragmented and outdated documentation and quickly changing libraries, I spent much more time than I would have liked trying to figure out how things worked, or why they didn't. This left me with too little time available to go as deep into each area as I had wanted, and I had to settle for a broader, more shallow perspective. I had especially wanted to provide examples of how to code directly against Blitz++ using SWIG or Boost as a wrapper. I also wish I could have tested more algorithms before drawing any conclusions. Neither of the algorithms I had time to implement had complex enough parameters to show any particularly interesting developments. Finally, I had much more functionality planned for the TimerSuite tool. This includes more advanced presentation and analysis of the results (and plotting them), stepping through individual runs; rerunning and modifying them on the fly, and more flexible options for configuring sessions. I will most probably continue improving the tool as a personal project though, as it turned out to be pretty useful.